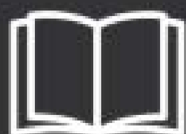
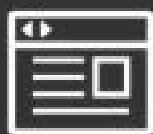
Prateek Joshi, John Hearty, Bastiaan Sjardin,
Luca Massaron, Alberto Boschetti

Python: Real World Machine Learning

Learning Path

Learn to solve challenging data science problems by building powerful machine learning models using Python





Packt>

Python: Real World Machine Learning

Table of Contents

Python: Real World Machine Learning
Python: Real World Machine Learning
Credits
Preface
- What this learning path covers
- What you need for this learning path
- Who this learning path is for
- Reader feedback
- Customer support
  - Downloading the example code
  - Errata
  - Piracy
  - Questions

I. Module 1
- 1. The Realm of Supervised Learning
  - Introduction
  - Preprocessing data using different techniques
    - Getting ready
    - How to do it…
      - Mean removal
      - Scaling
      - Normalization
      - Binarization
      - One Hot Encoding
  - Label encoding
    - How to do it…
  - Building a linear regressor
    - Getting ready
    - How to do it…
  - Computing regression accuracy
    - Getting ready
    - How to do it…
  - Achieving model persistence
    - How to do it…
  - Building a ridge regressor
    - Getting ready
    - How to do it…
  - Building a polynomial regressor
    - Getting ready
    - How to do it…
  - Estimating housing prices
    - Getting ready
    - How to do it…

- Computing the relative importance of features
  - How to do it…
- Estimating bicycle demand distribution
  - Getting ready
  - How to do it…
  - There's more…
- 2. Constructing a Classifier
  - Introduction
  - Building a simple classifier
    - How to do it…
    - There's more…
  - Building a logistic regression classifier
    - How to do it…
  - Building a Naive Bayes classifier
    - How to do it…
  - Splitting the dataset for training and testing
    - How to do it…
  - Evaluating the accuracy using cross-validation
    - Getting ready…
    - How to do it…
  - Visualizing the confusion matrix
    - How to do it…
  - Extracting the performance report
    - How to do it…
  - Evaluating cars based on their characteristics
    - Getting ready
    - How to do it…
  - Extracting validation curves
    - How to do it…
  - Extracting learning curves
    - How to do it…
  - Estimating the income bracket
    - How to do it…
- 3. Predictive Modeling
  - Introduction
  - Building a linear classifier using Support Vector Machine (SVMs)
    - Getting ready
    - How to do it…
  - Building a nonlinear classifier using SVMs
    - How to do it…
  - Tackling class imbalance
    - How to do it…
  - Extracting confidence measurements
    - How to do it…
  - Finding optimal hyperparameters
    - How to do it…
  - Building an event predictor

        - Getting ready
        - How to do it…
    - Estimating traffic
        - Getting ready
        - How to do it…
- 4. Clustering with Unsupervised Learning
    - Introduction
    - Clustering data using the k-means algorithm
        - How to do it…
    - Compressing an image using vector quantization
        - How to do it…
    - Building a Mean Shift clustering model
        - How to do it…
    - Grouping data using agglomerative clustering
        - How to do it…
    - Evaluating the performance of clustering algorithms
        - How to do it…
    - Automatically estimating the number of clusters using DBSCAN algorithm
        - How to do it…
    - Finding patterns in stock market data
        - How to do it…
    - Building a customer segmentation model
        - How to do it…
- 5. Building Recommendation Engines
    - Introduction
    - Building function compositions for data processing
        - How to do it…
    - Building machine learning pipelines
        - How to do it…
        - How it works…
    - Finding the nearest neighbors
        - How to do it…
    - Constructing a k-nearest neighbors classifier
        - How to do it…
        - How it works…
    - Constructing a k-nearest neighbors regressor
        - How to do it…
        - How it works…
    - Computing the Euclidean distance score
        - How to do it…
    - Computing the Pearson correlation score
        - How to do it…
    - Finding similar users in the dataset
        - How to do it…
    - Generating movie recommendations
        - How to do it…
- 6. Analyzing Text Data

- Introduction
- Preprocessing data using tokenization
    - How to do it…
- Stemming text data
    - How to do it…
    - How it works…
- Converting text to its base form using lemmatization
    - How to do it…
- Dividing text using chunking
    - How to do it…
- Building a bag-of-words model
    - How to do it…
    - How it works…
- Building a text classifier
    - How to do it…
    - How it works…
- Identifying the gender
    - How to do it…
- Analyzing the sentiment of a sentence
    - How to do it…
    - How it works…
- Identifying patterns in text using topic modeling
    - How to do it…
    - How it works…

7. Speech Recognition
    - Introduction
    - Reading and plotting audio data
        - How to do it…
    - Transforming audio signals into the frequency domain
        - How to do it…
    - Generating audio signals with custom parameters
        - How to do it…
    - Synthesizing music
        - How to do it…
    - Extracting frequency domain features
        - How to do it…
    - Building Hidden Markov Models
        - How to do it…
    - Building a speech recognizer
        - How to do it…
8. Dissecting Time Series and Sequential Data
    - Introduction
    - Transforming data into the time series format
        - How to do it…
    - Slicing time series data
        - How to do it…
    - Operating on time series data

- How to do it…
- Extracting statistics from time series data
  - How to do it…
- Building Hidden Markov Models for sequential data
  - Getting ready
  - How to do it…
- Building Conditional Random Fields for sequential text data
  - Getting ready
  - How to do it…
- Analyzing stock market data using Hidden Markov Models
  - How to do it…

9. Image Content Analysis
- Introduction
- Operating on images using OpenCV-Python
  - How to do it…
- Detecting edges
  - How to do it…
- Histogram equalization
  - How to do it…
- Detecting corners
  - How to do it…
- Detecting SIFT feature points
  - How to do it…
- Building a Star feature detector
  - How to do it…
- Creating features using visual codebook and vector quantization
  - How to do it…
- Training an image classifier using Extremely Random Forests
  - How to do it…
- Building an object recognizer
  - How to do it…

10. Biometric Face Recognition
- Introduction
- Capturing and processing video from a webcam
  - How to do it…
- Building a face detector using Haar cascades
  - How to do it…
- Building eye and nose detectors
  - How to do it…
- Performing Principal Components Analysis
  - How to do it…
- Performing Kernel Principal Components Analysis
  - How to do it…
- Performing blind source separation
  - How to do it…
- Building a face recognizer using Local Binary Patterns Histogram
  - How to do it…

- 11. Deep Neural Networks
  - Introduction
  - Building a perceptron
    - How to do it…
  - Building a single layer neural network
    - How to do it…
  - Building a deep neural network
    - How to do it…
  - Creating a vector quantizer
    - How to do it…
  - Building a recurrent neural network for sequential data analysis
    - How to do it…
  - Visualizing the characters in an optical character recognition database
    - How to do it…
  - Building an optical character recognizer using neural networks
    - How to do it…
- 12. Visualizing Data
  - Introduction
  - Plotting 3D scatter plots
    - How to do it…
  - Plotting bubble plots
    - How to do it…
  - Animating bubble plots
    - How to do it…
  - Drawing pie charts
    - How to do it…
  - Plotting date-formatted time series data
    - How to do it…
  - Plotting histograms
    - How to do it…
  - Visualizing heat maps
    - How to do it…
  - Animating dynamic signals
    - How to do it…

II. Module 2

- 1. Unsupervised Machine Learning
  - Principal component analysis
    - PCA – a primer
    - Employing PCA
  - Introducing k-means clustering
    - Clustering – a primer
    - Kick-starting clustering analysis
    - Tuning your clustering configurations
  - Self-organizing maps
    - SOM – a primer
    - Employing SOM
  - Further reading

- Summary
- 2. Deep Belief Networks
  - Neural networks – a primer
    - The composition of a neural network
    - Network topologies
  - Restricted Boltzmann Machine
    - Introducing the RBM
      - Topology
      - Training
    - Applications of the RBM
    - Further applications of the RBM
  - Deep belief networks
    - Training a DBN
    - Applying the DBN
    - Validating the DBN
  - Further reading
  - Summary
- 3. Stacked Denoising Autoencoders
  - Autoencoders
    - Introducing the autoencoder
      - Topology
      - Training
    - Denoising autoencoders
    - Applying a dA
  - Stacked Denoising Autoencoders
    - Applying the SdA
    - Assessing SdA performance
  - Further reading
  - Summary
- 4. Convolutional Neural Networks
  - Introducing the CNN
    - Understanding the convnet topology
      - Understanding convolution layers
      - Understanding pooling layers
      - Training a convnet
      - Putting it all together
    - Applying a CNN
  - Further Reading
  - Summary
- 5. Semi-Supervised Learning
  - Introduction
  - Understanding semi-supervised learning
  - Semi-supervised algorithms in action
    - Self-training
      - Implementing self-training
      - Finessing your self-training implementation
        - Improving the selection process

- Contrastive Pessimistic Likelihood Estimation
- Further reading
- Summary
- 6. Text Feature Engineering
  - Introduction
  - Text feature engineering
    - Cleaning text data
      - Text cleaning with BeautifulSoup
      - Managing punctuation and tokenizing
      - Tagging and categorising words
        - Tagging with NLTK
        - Sequential tagging
        - Backoff tagging
    - Creating features from text data
      - Stemming
      - Bagging and random forests
    - Testing our prepared data
  - Further reading
  - Summary
- 7. Feature Engineering Part II
  - Introduction
  - Creating a feature set
    - Engineering features for ML applications
      - Using rescaling techniques to improve the learnability of features
      - Creating effective derived variables
      - Reinterpreting non-numeric features
    - Using feature selection techniques
      - Performing feature selection
        - Correlation
        - LASSO
        - Recursive Feature Elimination
        - Genetic models
  - Feature engineering in practice
    - Acquiring data via RESTful APIs
      - Testing the performance of our model
      - Twitter
        - Translink Twitter
        - Consumer comments
        - The Bing Traffic API
      - Deriving and selecting variables using feature engineering techniques
        - The weather API
  - Further reading
  - Summary
- 8. Ensemble Methods
  - Introducing ensembles
    - Understanding averaging ensembles
      - Using bagging algorithms

- Using random forests
- Applying boosting methods
- Using XGBoost
- Using stacking ensembles
- Applying ensembles in practice
- Using models in dynamic applications
- Understanding model robustness
- Identifying modeling risk factors
- Strategies to managing model robustness
- Further reading
- Summary
- 9. Additional Python Machine Learning Tools
- Alternative development tools
- Introduction to Lasagne
- Getting to know Lasagne
- Introduction to TensorFlow
- Getting to know TensorFlow
- Using TensorFlow to iteratively improve our models
- Knowing when to use these libraries
- Further reading
- Summary
- A. Chapter Code Requirements
- III. Module 3
- 1. First Steps to Scalability
- Explaining scalability in detail
- Making large scale examples
- Introducing Python
- Scale up with Python
- Scale out with Python
- Python for large scale machine learning
- Choosing between Python 2 and Python 3
- Package upgrades
- Scientific distributions
- Introducing Jupyter/IPython
- Python packages
- NumPy
- SciPy
- Pandas
- Scikit-learn
- The matplotlib package
- Gensim
- H2O
- XGBoost
- Theano
- TensorFlow
- The sknn library
- Theanets

- Keras
- Other useful packages to install on your system
- Summary
- 2. Scalable Learning in Scikit-learn
  - Out-of-core learning
    - Subsampling as a viable option
    - Optimizing one instance at a time
    - Building an out-of-core learning system
  - Streaming data from sources
    - Datasets to try the real thing yourself
    - The first example – streaming the bike-sharing dataset
    - Using pandas I/O tools
    - Working with databases
    - Paying attention to the ordering of instances
  - Stochastic learning
    - Batch gradient descent
    - Stochastic gradient descent
    - The Scikit-learn SGD implementation
    - Defining SGD learning parameters
  - Feature management with data streams
    - Describing the target
    - The hashing trick
    - Other basic transformations
    - Testing and validation in a stream
    - Trying SGD in action
  - Summary
- 3. Fast SVM Implementations
  - Datasets to experiment with on your own
    - The bike-sharing dataset
    - The covertype dataset
  - Support Vector Machines
    - Hinge loss and its variants
    - Understanding the Scikit-learn SVM implementation
    - Pursuing nonlinear SVMs by subsampling
    - Achieving SVM at scale with SGD
  - Feature selection by regularization
  - Including non-linearity in SGD
    - Trying explicit high-dimensional mappings
  - Hyperparameter tuning
    - Other alternatives for SVM fast learning
      - Nonlinear and faster with Vowpal Wabbit
      - Installing VW
      - Understanding the VW data format
      - Python integration
      - A few examples using reductions for SVM and neural nets
      - Faster bike-sharing
      - The covertype dataset crunched by VW

- Summary
- 4. Neural Networks and Deep Learning
  - The neural network architecture
    - What and how neural networks learn
    - Choosing the right architecture
      - The input layer
      - The hidden layer
      - The output layer
    - Neural networks in action
    - Parallelization for sknn
  - Neural networks and regularization
  - Neural networks and hyperparameter optimization
  - Neural networks and decision boundaries
  - Deep learning at scale with H2O
    - Large scale deep learning with H2O
    - Gridsearch on H2O
  - Deep learning and unsupervised pretraining
  - Deep learning with theanets
  - Autoencoders and unsupervised learning
    - Autoencoders
  - Summary
- 5. Deep Learning with TensorFlow
  - TensorFlow installation
    - TensorFlow operations
      - GPU computing
      - Linear regression with SGD
      - A neural network from scratch in TensorFlow
  - Machine learning on TensorFlow with SkFlow
    - Deep learning with large files – incremental learning
  - Keras and TensorFlow installation
  - Convolutional Neural Networks in TensorFlow through Keras
    - The convolution layer
    - The pooling layer
    - The fully connected layer
  - CNN's with an incremental approach
  - GPU Computing
  - Summary
- 6. Classification and Regression Trees at Scale
  - Bootstrap aggregation
  - Random forest and extremely randomized forest
  - Fast parameter optimization with randomized search
    - Extremely randomized trees and large datasets
  - CART and boosting
    - Gradient Boosting Machines
      - max_depth
      - learning_rate
      - Subsample

            - Faster GBM with warm_start
                - Speeding up GBM with warm_start
            - Training and storing GBM models
    - XGBoost
        - XGBoost regression
            - XGBoost and variable importance
        - XGBoost streaming large datasets
        - XGBoost model persistence
    - Out-of-core CART with H2O
        - Random forest and gridsearch on H2O
        - Stochastic gradient boosting and gridsearch on H2O
    - Summary
- 7. Unsupervised Learning at Scale
    - Unsupervised methods
    - Feature decomposition – PCA
        - Randomized PCA
        - Incremental PCA
        - Sparse PCA
    - PCA with H2O
    - Clustering – K-means
        - Initialization methods
        - K-means assumptions
        - Selection of the best K
        - Scaling K-means – mini-batch
    - K-means with H2O
    - LDA
        - Scaling LDA – memory, CPUs, and machines
    - Summary
- 8. Distributed Environments – Hadoop and Spark
    - From a standalone machine to a bunch of nodes
        - Why do we need a distributed framework?
    - Setting up the VM
        - VirtualBox
        - Vagrant
        - Using the VM
    - The Hadoop ecosystem
        - Architecture
        - HDFS
        - MapReduce
        - YARN
    - Spark
        - pySpark
    - Summary
- 9. Practical Machine Learning with Spark
    - Setting up the VM for this chapter
    - Sharing variables across cluster nodes
        - Broadcast read-only variables

- Accumulators write-only variables
- Broadcast and accumulators together – an example
- Data preprocessing in Spark
  - JSON files and Spark DataFrames
  - Dealing with missing data
  - Grouping and creating tables in-memory
  - Writing the preprocessed DataFrame or RDD to disk
  - Working with Spark DataFrames
- Machine learning with Spark
  - Spark on the KDD99 dataset
  - Reading the dataset
  - Feature engineering
  - Training a learner
  - Evaluating a learner's performance
  - The power of the ML pipeline
  - Manual tuning
  - Cross-validation
    - Final cleanup
- Summary
- A. Introduction to GPUs and Theano
  - GPU computing
  - Theano – parallel computing on the GPU
  - Installing Theano
- A. Bibliography
- Index

Python: Real World Machine Learning

Python: Real World Machine Learning

Learn to solve challenging data science problems by building powerful machine learning models using Python

A course in three modules

BIRMINGHAM - MUMBAI

Python: Real World Machine Learning

Copyright © 2016 Packt Publishing

All rights reserved. No part of this course may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this course to ensure the accuracy of the information presented. However, the information contained in this course is sold without warranty, either express or implied. Neither the authors, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this course.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this course by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

Published on: October 2016

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham B3 2PB, UK.

ISBN 978-1-78712-321-2

www.packtpub.com

Credits

Authors

Prateek Joshi

John Hearty

Bastiaan Sjardin

Luca Massaron

Alberto Boschetti

Reviewers

Dr. Vahid Mirjalili

Jared Huffman

Ashwin Pajankar

Oleg Okun

Kai Londenberg

Content Development Editor

Aishwarya Pandere

Production Coordinator

Nilesh Mohite

Preface

Machine learning is becoming increasingly pervasive in the modern data-driven world. It is used extensively across many fields, such as search engines, robotics, self-driving cars, and so on. In this course, you will explore various real-life scenarios where you can use machine learning. You will understand what algorithms you should use in a given context using this exciting recipe-based guide.

This course starts by talking about various realms in machine learning followed by practical examples.

What this learning path covers

Module 1, *Python Machine Learning Cookbook*, teaches you about the algorithms that we use to build recommendation engines. We will learn how to apply these algorithms to collaborative filtering and movie recommendations.

Module 2, *Advanced Machine Learning with Python*, explains how to apply several semi-supervised learning techniques including CPLE, self learning, and S3VM.

Module 3, *Large Scale Machine Learning with Python*, covers interesting deep learning techniques together with an online method for neural networks. Although TensorFlow is only in its infancy, the framework provides elegant machine learning solutions.

What you need for this learning path

Module 1: While we believe that the world is moving forward with better versions coming out, a lot of developers still enjoy using Python 2.x. A lot of operating systems have Python 2.x built into them. This course is focused on machine learning in Python as opposed to Python itself. It also helps in maintaining compatibility with libraries that haven't been ported to Python 3.x. Hence the code in the book is oriented towards Python 2.x. In that spirit, we have tried to keep all the code as agnostic as possible to the Python versions.

Module 2: The entirety of this course's content leverages openly available data and code,including open source Python libraries and frameworks. While each chapter's example code is accompanied by a README file documenting all the libraries required to run the code provided in that chapter's accompanying scripts, the content of these files is collated here for your convenience. It is recommended that some libraries required for earlier chapters be available when working with code from any later chapter. These requirements are identified using bold text. Particularly, it is important to set up the first chapter's required libraries for any content later in the book.

Module 3: The execution of the code examples provided in this book requires an installation of Python 2.7 or higher versions on macOS, Linux, or Microsoft Windows.

The examples throughout the book will make frequent use of Python's essential libraries, such as SciPy, NumPy, Scikit-learn, and StatsModels, and to a minor extent, matplotlib and pandas, for scientific and statistical computing. We will also make use of an out-of-core cloud computing application called H2O. This book is highly dependent on Jupyter and its Notebooks powered by the Python kernel. We will use its most recent version, 4.1, for this book. The first chapter will provide you with all the step-by-step instructions and some useful tips to set up your Python environment, these core libraries, and all the necessary tools.

Who this learning path is for

This Learning Path is for Python programmers who are looking to use machine learning algorithms to create real-world applications. Python professionals intending to work with large and complex datasets and Python developers and analysts or data scientists who are looking to add to their existing skills by accessing some of the most powerful recent trends in data science will find this Learning Path useful. Experience with Python, Jupyter Notebooks, and command-line execution together with a good level of mathematical knowledge to understand the concepts is expected. Machine learning basics is also expected.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this course—what you liked or disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of.

To send us general feedback, simply e-mail <`feedback@packtpub.com`>, and mention the course's title in the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide at www.packtpub.com/authors.

Customer support

Now that you are the proud owner of a Packt course, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for this course from your account at http://www.packtpub.com. If you purchased this course elsewhere, you can visit http://www.packtpub.com/support and register to have the files e-mailed directly to you.

You can download the code files by following these steps:

1. Log in or register to our website using your e-mail address and password.
2. Hover the mouse pointer on the **SUPPORT** tab at the top.
3. Click on **Code Downloads & Errata**.
4. Enter the name of the course in the **Search** box.
5. Select the course for which you're looking to download the code files.
6. Choose from the drop-down menu where you purchased this course from.
7. Click on **Code Download**.

You can also download the code files by clicking on the **Code Files** button on the course's webpage at the Packt Publishing website. This page can be accessed by entering the course's name in the **Search** box. Please note that you need to be logged in to your Packt account.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR / 7-Zip for Windows
- Zipeg / iZip / UnRarX for Mac
- 7-Zip / PeaZip for Linux

The code bundle for the course is also hosted on GitHub at https://github.com/PacktPublishing/Python-Real-World-Machine-Learning. We also have other code bundles from our rich catalog of books, videos, and courses available at https://github.com/PacktPublishing/. Check them out!

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our courses—maybe a mistake in the text or the code—we would be grateful if you could report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this course. If you find any errata, please report them by visiting http://www.packtpub.com/submit-errata, selecting your course, clicking on the **Errata Submission Form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website or added to any list of existing errata under the Errata section of that title.

To view the previously submitted errata, go to https://www.packtpub.com/books/content/support and enter the name of the course in the search field. The required information will appear under the **Errata** section.

Piracy

Piracy of copyrighted material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works in any form on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at <`copyright@packtpub.com`> with a link to the suspected pirated material.

We appreciate your help in protecting our authors and our ability to bring you valuable content.

Questions

If you have a problem with any aspect of this course, you can contact us at <`questions@packtpub.com`>, and we will do our best to address the problem.

Part I. Module 1

Python Machine Learning Cookbook

100 recipes that teach you how to perform various machine learning tasks in the real world

Chapter 1. The Realm of Supervised Learning

In this chapter, we will cover the following recipes:

- Preprocessing data using different techniques
- Label encoding
- Building a linear regressor
- Computing regression accuracy
- Achieving model persistence
- Building a ridge regressor
- Building a polynomial regressor
- Estimating housing prices
- Computing the relative importance of features
- Estimating bicycle demand distribution

Introduction

If you are familiar with the basics of machine learning, you will certainly know what supervised learning is all about. To give you a quick refresher, supervised learning refers to building a machine learning model that is based on labeled samples. For example, if we build a system to estimate the price of a house based on various parameters, such as size, locality, and so on, we first need to create a database and label it. We need to tell our algorithm what parameters correspond to what prices. Based on this data, our algorithm will learn how to calculate the price of a house using the input parameters.

Unsupervised learning is the opposite of what we just discussed. There is no labeled data available here. Let's assume that we have a bunch of datapoints, and we just want to separate them into multiple groups. We don't exactly know what the criteria of separation would be. So, an unsupervised learning algorithm will try to separate the given dataset into a fixed number of groups in the best possible way. We will discuss unsupervised learning in the upcoming chapters.

We will use various Python packages, such as **NumPy**, **SciPy**, **scikit-learn**, and **matplotlib**, during the course of this book to build various things. If you use Windows, it is recommended that you use a SciPy-stack compatible version of Python. You can check the list of compatible versions at http://www.scipy.org/install.html. These distributions come with all the necessary packages already installed. If you use Mac OS X or Ubuntu, installing these packages is fairly straightforward. Here are some useful links for installation and documentation:

- **NumPy**: http://docs.scipy.org/doc/numpy-1.10.1/user/install.html
- **SciPy**: http://www.scipy.org/install.html
- **scikit-learn**: http://scikit-learn.org/stable/install.html
- **matplotlib**: http://matplotlib.org/1.4.2/users/installing.html

Make sure that you have these packages installed on your machine before you proceed.

Preprocessing data using different techniques

In the real world, we usually have to deal with a lot of raw data. This raw data is not readily ingestible by machine learning algorithms. To prepare the data for machine learning, we have to preprocess it before we feed it into various algorithms.

Getting ready

Let's see how to preprocess data in Python. To start off, open a file with a `.py` extension, for example, `preprocessor.py`, in your favorite text editor. Add the following lines to this file:

```
import numpy as np
from sklearn import preprocessing
```

We just imported a couple of necessary packages. Let's create some sample data. Add the following line to this file:

```
data = np.array([[3, -1.5,  2, -5.4], [0,  4,  -0.3, 2.1], [1,  3.3,
-1.9, -4.3]])
```

We are now ready to operate on this data.

How to do it…

Data can be preprocessed in many ways. We will discuss a few of the most commonly-used preprocessing techniques.

Mean removal

It's usually beneficial to remove the mean from each feature so that it's centered on zero. This helps us in removing any bias from the features. Add the following lines to the file that we opened earlier:

```
data_standardized = preprocessing.scale(data)
print "\nMean =", data_standardized.mean(axis=0)
print "Std deviation =", data_standardized.std(axis=0)
```

We are now ready to run the code. To do this, run the following command on your Terminal:

```
$ python preprocessor.py
```

You will see the following output on your Terminal:

```
Mean = [  5.55111512e-17  -1.11022302e-16  -7.40148683e-17
-7.40148683e-17]
Std deviation = [ 1.  1.  1.  1.]
```

You can see that the mean is almost `0` and the standard deviation is `1`.

Scaling

The values of each feature in a datapoint can vary between random values. So, sometimes it is important to scale them so that this becomes a level playing field. Add the following lines to the file and run the code:

```
data_scaler = preprocessing.MinMaxScaler(feature_range=(0, 1))
data_scaled = data_scaler.fit_transform(data)
print "\nMin max scaled data =", data_scaled
```

After scaling, all the feature values range between the specified values. The output will be displayed, as follows:

```
Min max scaled data:
[[ 1.          0.          1.          0.        ]
 [ 0.          1.          0.41025641  1.        ]
 [ 0.33333333  0.87272727  0.          0.14666667]]
```

Normalization

Data normalization is used when you want to adjust the values in the feature vector so that they can be measured on a common scale. One of the most common forms of normalization that is used in machine learning adjusts the values of a feature vector so that they sum up to 1. Add the following lines to the previous file:

```
data_normalized = preprocessing.normalize(data, norm='l1')
print "\nL1 normalized data =", data_normalized
```

If you run the Python file, you will get the following output:

```
L1 normalized data:
[[ 0.25210084 -0.12605042  0.16806723 -0.45378151]
 [ 0.          0.625      -0.046875    0.328125  ]
 [ 0.0952381   0.31428571 -0.18095238 -0.40952381]]
```

This is used a lot to make sure that datapoints don't get boosted artificially due to the fundamental nature of their features.

Binarization

Binarization is used when you want to convert your numerical feature vector into a Boolean vector. Add the following lines to the Python file:

```
data_binarized =
preprocessing.Binarizer(threshold=1.4).transform(data)
print "\nBinarized data =", data_binarized
```

Run the code again, and you will see the following output:

```
Binarized data:
[[ 1.  0.  1.  0.]
 [ 0.  1.  0.  1.]
 [ 0.  1.  0.  0.]]
```

This is a very useful technique that's usually used when we have some prior knowledge of the data.

One Hot Encoding

A lot of times, we deal with numerical values that are sparse and scattered all over the place. We don't really need to store these big values. This is where One Hot Encoding comes into picture. We can think of One Hot Encoding as a tool to *tighten* the feature vector. It looks at each feature and identifies the total number of distinct values. It uses a *one-of-k* scheme to encode the values. Each feature in the feature vector is encoded based on this. This helps us be more efficient in terms of space. For example, let's say we are dealing with 4-dimensional feature vectors. To encode the *n*-th feature in a feature vector, the encoder will go through the *n*-th feature in each feature vector and count the number of distinct values. If the number of distinct values is *k*, it will transform the feature into a *k*-dimensional vector where only one value is *1* and all other values are *0*. Add the following lines to the Python file:

```
encoder = preprocessing.OneHotEncoder()
encoder.fit([[0, 2, 1, 12], [1, 3, 5, 3], [2, 3, 2, 12], [1, 2, 4,
3]])
encoded_vector = encoder.transform([[2, 3, 5, 3]]).toarray()
print "\nEncoded vector =", encoded_vector
```

This is the expected output:

```
Encoded vector:
[[ 0.  0.  1.  0.  1.  0.  0.  0.  1.  1.  0.]]
```

In the above example, let's consider the third feature in each feature vector. The values are `1`, `5`, `2`, and `4`. There are four distinct values here, which means the one-hot encoded vector will be of length 4. If you want to encode the value `5`, it will be a vector `[0, 1, 0, 0]`. Only one value can be *1* in this vector. The second element is 1, which indicates that the value is `5`.

Label encoding

In supervised learning, we usually deal with a variety of labels. These can be in the form of numbers or words. If they are numbers, then the algorithm can use them directly. However, a lot of times, labels need to be in human readable form. So, people usually label the training data with words. Label encoding refers to transforming the word labels into numerical form so that the algorithms can understand how to operate on them. Let's take a look at how to do this.

How to do it…

1. Create a new Python file, and import the preprocessing package:

```
from sklearn import preprocessing
```

2. This package contains various functions that are needed for data preprocessing. Let's define the label encoder, as follows:

```
label_encoder = preprocessing.LabelEncoder()
```

3. The `label_encoder` object knows how to understand word labels. Let's create some labels:

```
input_classes = ['audi', 'ford', 'audi', 'toyota', 'ford',
'bmw']
```

4. We are now ready to encode these labels:

```
label_encoder.fit(input_classes)
print "\nClass mapping:"
for i, item in enumerate(label_encoder.classes_):
    print item, '-->', i
```

5. Run the code, and you will see the following output on your Terminal:

```
Class mapping:
audi --> 0
bmw --> 1
ford --> 2
toyota --> 3
```

6. As shown in the preceding output, the words have been transformed into 0-indexed numbers. Now, when you encounter a set of labels, you can simply transform them, as follows:

```
labels = ['toyota', 'ford', 'audi']
encoded_labels = label_encoder.transform(labels)
print "\nLabels =", labels
print "Encoded labels =", list(encoded_labels)
```

Here is the output that you'll see on your Terminal:

```
Labels = ['toyota', 'ford', 'audi']
Encoded labels = [3, 2, 0]
```

7. This is way easier than manually maintaining mapping between words and numbers. You can check the correctness by transforming numbers back to word labels:

```
encoded_labels = [2, 1, 0, 3, 1]
decoded_labels = label_encoder.inverse_transform(encoded_labels)
print "\nEncoded labels =", encoded_labels
print "Decoded labels =", list(decoded_labels)
```

Here is the output:

```
Encoded labels = [2, 1, 0, 3, 1]
Decoded labels = ['ford', 'bmw', 'audi', 'toyota', 'bmw']
```

As you can see, the mapping is preserved perfectly.

Building a linear regressor

Regression is the process of estimating the relationship between input data and the continuous-valued output data. This data is usually in the form of real numbers, and our goal is to estimate the underlying function that governs the mapping from the input to the output. Let's start with a very simple example. Consider the following mapping between input and output:

1 --> 2

3 --> 6

4.3 --> 8.6

7.1 --> 14.2

If I ask you to estimate the relationship between the inputs and the outputs, you can easily do this by analyzing the pattern. We can see that the output is twice the input value in each case, so the transformation would be as follows:

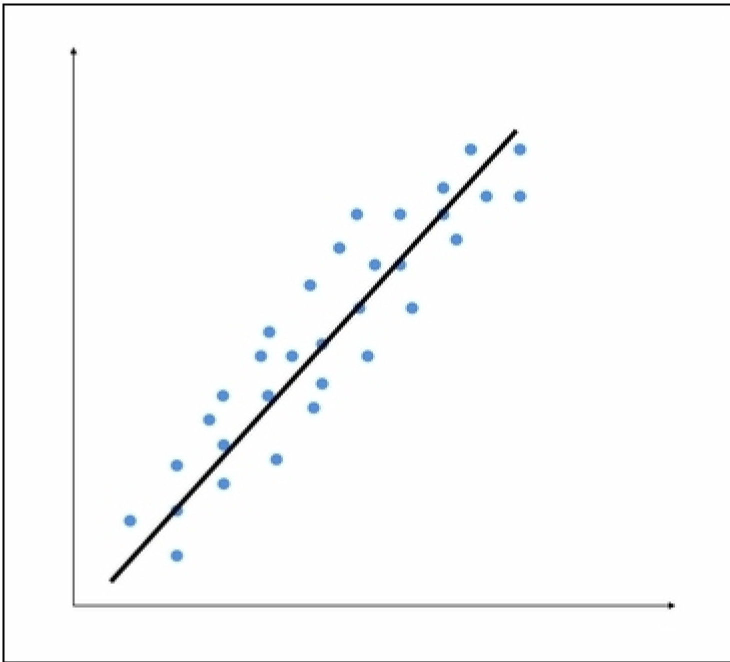
$f(x) = 2x$

This is a simple function, relating the input values with the output values. However, in the real world, this is usually not the case. Functions in the real world are not so straightforward!

Getting ready

Linear regression refers to estimating the underlying function using a linear combination of input variables. The preceding example was an example that consisted of one input variable and one output variable.

Consider the following figure:



The goal of linear regression is to extract the underlying linear model that relates the input variable to the output variable. This aims to minimize the sum of squares of differences between the actual output and the predicted output using a linear function. This method is called **Ordinary least squares**.

You might say that there might be a curvy line out there that fits these points better, but linear regression doesn't allow this. The main advantage of linear regression is that it's not complex. If you go into nonlinear regression, you may get more accurate models, but they will be slower. As shown in the preceding figure, the model tries to approximate the input datapoints using a straight line. Let's see how to build a linear regression model in Python.

How to do it…

You have been provided with a data file, called `data_singlevar.txt`. This contains comma-separated lines where the first element is the input value and the second element is the output value that corresponds to this input value. You should use this as the input argument:

1. Create a file called `regressor.py`, and add the following lines:

```
import sys
import numpy as np
filename = sys.argv[1]
X = []
y = []
with open(filename, 'r') as f:
    for line in f.readlines():
        xt, yt = [float(i) for i in line.split(',')]
```

```
        X.append(xt)
        y.append(yt)
```

We just loaded the input data into `X` and `y`, where `X` refers to data and `y` refers to labels. Inside the loop in the preceding code, we parse each line and split it based on the comma operator. We then convert it into floating point values and save it in `X` and `y`, respectively.

2. When we build a machine learning model, we need a way to validate our model and check whether the model is performing at a satisfactory level. To do this, we need to separate our data into two groups: a training dataset and a testing dataset. The training dataset will be used to build the model, and the testing dataset will be used to see how this trained model performs on unknown data. So, let's go ahead and split this data into training and testing datasets:

```
num_training = int(0.8 * len(X))
num_test = len(X) - num_training

# Training data
X_train = np.array(X[:num_training]).reshape((num_training,1))
y_train = np.array(y[:num_training])

# Test data
X_test = np.array(X[num_training:]).reshape((num_test,1))
y_test = np.array(y[num_training:])
```

Here, we will use 80% of the data for the training dataset and the remaining 20% for the testing dataset.

3. We are now ready to train the model. Let's create a regressor object, as follows:

```
from sklearn import linear_model

# Create linear regression object
linear_regressor = linear_model.LinearRegression()

# Train the model using the training sets
linear_regressor.fit(X_train, y_train)
```

4. We just trained the linear regressor, based on our training data. The fit method takes the input data and trains the model. Let's see how it fits:

```
import matplotlib.pyplot as plt

y_train_pred = linear_regressor.predict(X_train)
plt.figure()
plt.scatter(X_train, y_train, color='green')
plt.plot(X_train, y_train_pred, color='black', linewidth=4)
plt.title('Training data')
plt.show()
```

5. We are now ready to run the code using the following command:

```
$ python regressor.py data_singlevar.txt
```

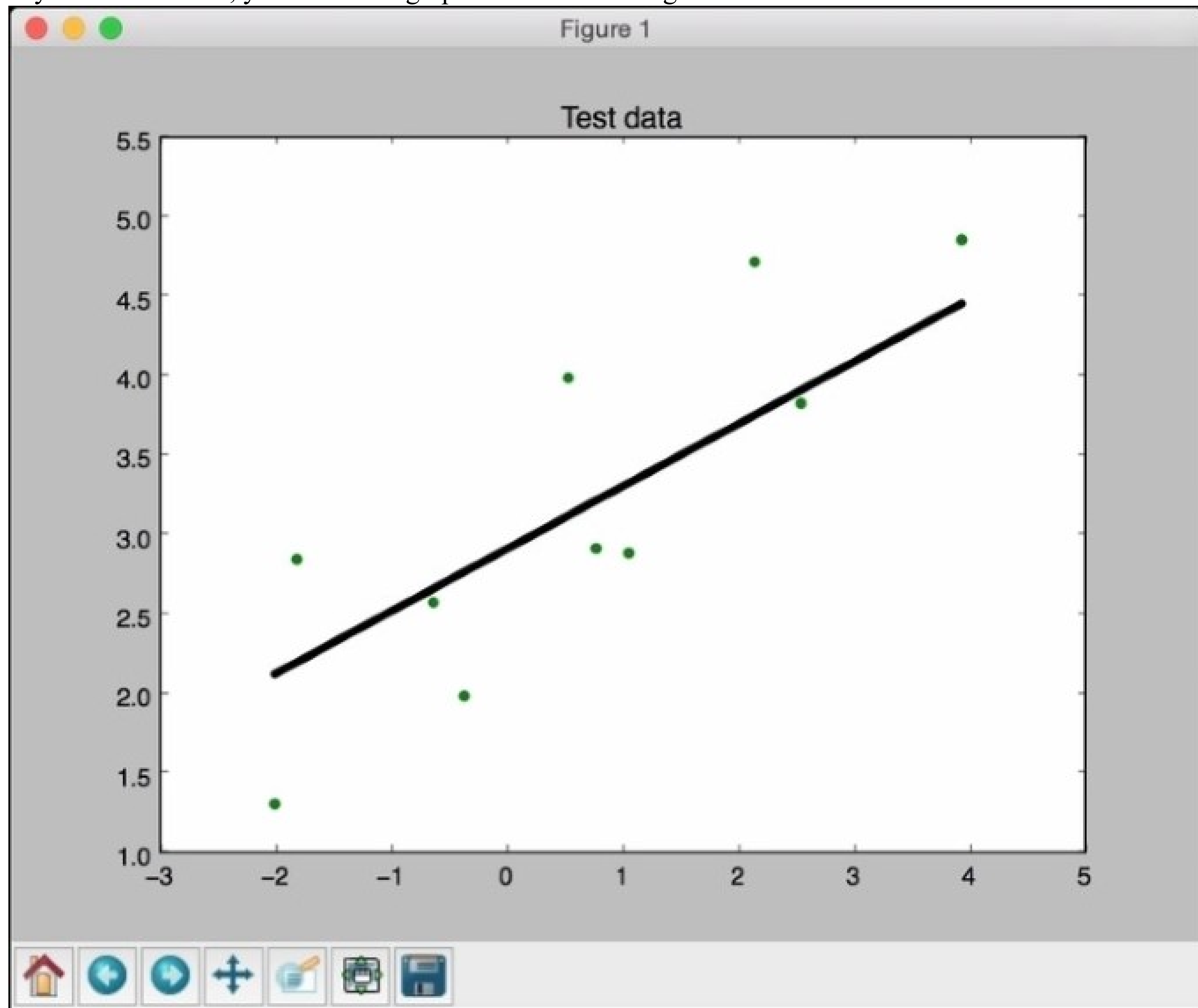
You should see the following figure:

6. In the preceding code, we used the trained model to predict the output for our training data. This wouldn't tell us how the model performs on unknown data because we are running it on training data itself. This just gives us an idea of how the model fits on training data. Looks like it's doing okay as you can see in the preceding figure!
7. Let's predict the test dataset output based on this model and plot it, as follows:

```
y_test_pred = linear_regressor.predict(X_test)

plt.scatter(X_test, y_test, color='green')
plt.plot(X_test, y_test_pred, color='black', linewidth=4)
```

```
plt.title('Test data')
plt.show()
```

If you run this code, you will see a graph like the following one:



Computing regression accuracy

Now that we know how to build a regressor, it's important to understand how to evaluate the quality of a regressor as well. In this context, an error is defined as the difference between the actual value and the value that is predicted by the regressor.

Getting ready

Let's quickly understand what metrics can be used to measure the quality of a regressor. A regressor can be evaluated using many different metrics, such as the following:

- **Mean absolute error**: This is the average of absolute errors of all the datapoints in the given dataset.
- **Mean squared error**: This is the average of the squares of the errors of all the datapoints in the given dataset. It is one of the most popular metrics out there!
- **Median absolute error**: This is the median of all the errors in the given dataset. The main advantage of this metric is that it's robust to outliers. A single bad point in the test dataset wouldn't skew the entire error metric, as opposed to a mean error metric.
- **Explained variance score**: This score measures how well our model can account for the variation in our dataset. A score of 1.0 indicates that our model is perfect.
- **R2 score**: This is pronounced as R-squared, and this score refers to the coefficient of determination. This tells us how well the unknown samples will be predicted by our model. The best possible score is 1.0, and the values can be negative as well.

How to do it…

There is a module in scikit-learn that provides functionalities to compute all the following metrics. Open a new Python file and add the following lines:

```
import sklearn.metrics as sm

print "Mean absolute error =", round(sm.mean_absolute_error(y_test,
y_test_pred), 2)
print "Mean squared error =", round(sm.mean_squared_error(y_test,
y_test_pred), 2)
print "Median absolute error =",
round(sm.median_absolute_error(y_test, y_test_pred), 2)
print "Explained variance score =",
round(sm.explained_variance_score(y_test, y_test_pred), 2)
print "R2 score =", round(sm.r2_score(y_test, y_test_pred), 2)
```

Keeping track of every single metric can get tedious, so we pick one or two metrics to evaluate our model. A good practice is to make sure that the mean squared error is low and the explained variance score is high.

Achieving model persistence

When we train a model, it would be nice if we could save it as a file so that it can be used later by simply loading it again.

How to do it…

Let's see how to achieve model persistence programmatically:

1. Add the following lines to `regressor.py`:

```
import cPickle as pickle

output_model_file = 'saved_model.pkl'
with open(output_model_file, 'w') as f:
    pickle.dump(linear_regressor, f)
```

2. The regressor object will be saved in the `saved_model.pkl` file. Let's look at how to load it and use it, as follows:

```
with open(output_model_file, 'r') as f:
    model_linregr = pickle.load(f)

y_test_pred_new = model_linregr.predict(X_test)
print "\nNew mean absolute error =",
round(sm.mean_absolute_error(y_test, y_test_pred_new), 2)
```

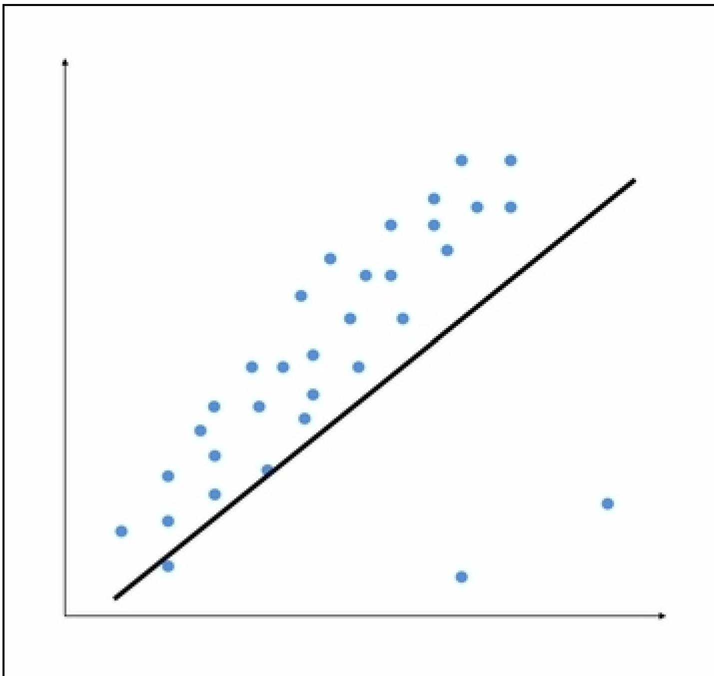
3. Here, we just loaded the regressor from the file into the `model_linregr` variable. You can compare the preceding result with the earlier result to confirm that it's the same.

Building a ridge regressor

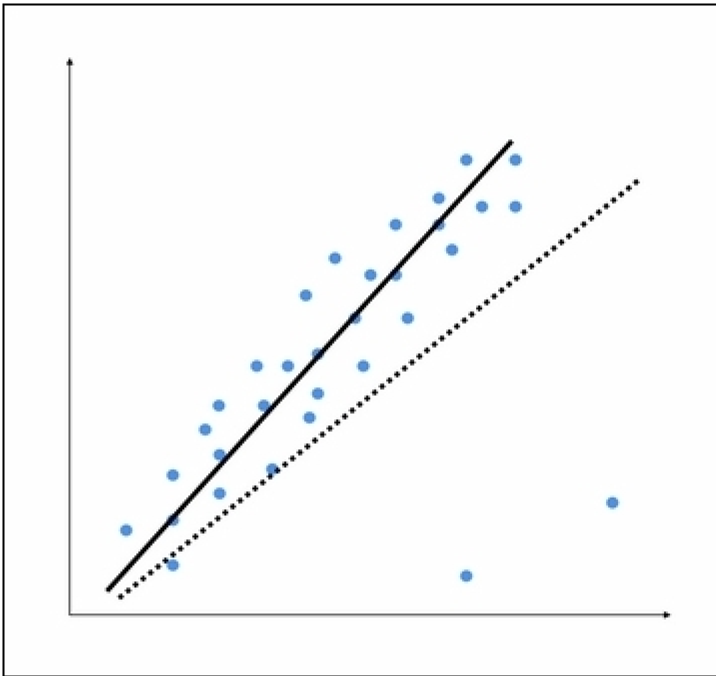
One of the main problems of linear regression is that it's sensitive to outliers. During data collection in the real world, it's quite common to wrongly measure the output. Linear regression uses ordinary least squares, which tries to minimize the squares of errors. The outliers tend to cause problems because they contribute a lot to the overall error. This tends to disrupt the entire model.

Getting ready

Let's consider the following figure:



The two points on the bottom are clearly outliers, but this model is trying to fit all the points. Hence, the overall model tends to be inaccurate. By visual inspection, we can see that the following figure is a better model:



Ordinary least squares considers every single datapoint when it's building the model. Hence, the actual model ends up looking like the dotted line as shown in the preceding figure. We can clearly see that this model is suboptimal. To avoid this, we use **regularization** where a penalty is imposed on the size of the coefficients. This method is called **Ridge Regression**.

How to do it…

Let's see how to build a ridge regressor in Python:

1. You can load the data from the `data_multi_variable.txt` file. This file contains multiple values in each line. All the values except the last value form the input feature vector.
2. Add the following lines to `regressor.py`. Let's initialize a ridge regressor with some parameters:

```
ridge_regressor = linear_model.Ridge(alpha=0.01,
fit_intercept=True, max_iter=10000)
```

3. The `alpha` parameter controls the complexity. As `alpha` gets closer to `0`, the ridge regressor tends to become more like a linear regressor with ordinary least squares. So, if you want to make it robust against outliers, you need to assign a higher value to `alpha`. We considered a value of `0.01`, which is moderate.
4. Let's train this regressor, as follows:

```
ridge_regressor.fit(X_train, y_train)
y_test_pred_ridge = ridge_regressor.predict(X_test)
print "Mean absolute error =",
```

```
round(sm.mean_absolute_error(y_test, y_test_pred_ridge), 2)
print "Mean squared error =",
round(sm.mean_squared_error(y_test, y_test_pred_ridge), 2)
print "Median absolute error =",
round(sm.median_absolute_error(y_test, y_test_pred_ridge), 2)
print "Explain variance score =",
round(sm.explained_variance_score(y_test, y_test_pred_ridge),
2)
print "R2 score =", round(sm.r2_score(y_test,
y_test_pred_ridge), 2)
```

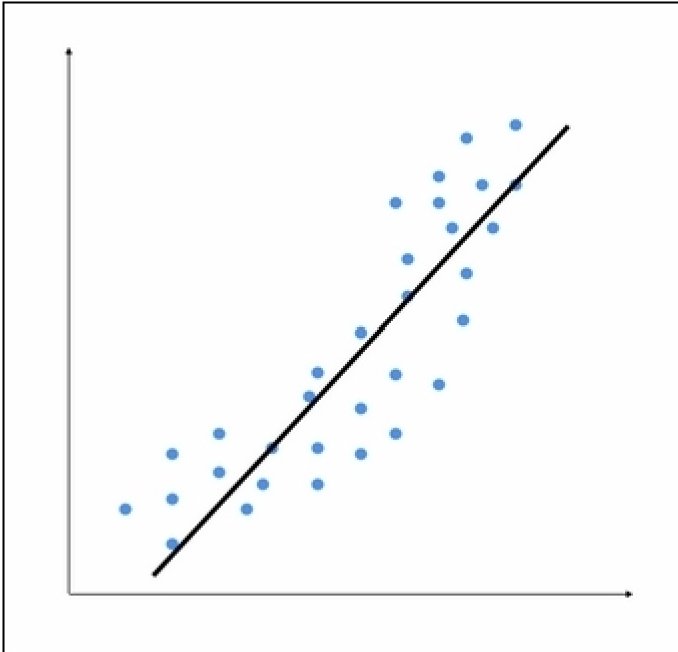
Run this code to view the error metrics. You can build a linear regressor to compare and contrast the results on the same data to see the effect of introducing regularization into the model.

Building a polynomial regressor

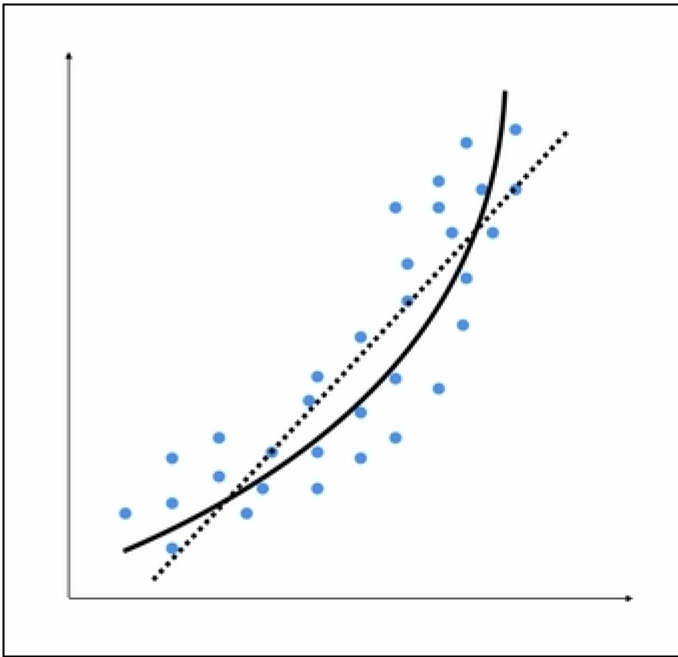
One of the main constraints of a linear regression model is the fact that it tries to fit a linear function to the input data. The polynomial regression model overcomes this issue by allowing the function to be a polynomial, thereby increasing the accuracy of the model.

Getting ready

Let's consider the following figure:



We can see that there is a natural curve to the pattern of datapoints. This linear model is unable to capture this. Let's see what a polynomial model would look like:



The dotted line represents the linear regression model, and the solid line represents the polynomial regression model. The curviness of this model is controlled by the degree of the polynomial. As the curviness of the model increases, it gets more accurate. However, curviness adds complexity to the model as well, hence, making it slower. This is a trade off where you have to decide between how accurate you want your model to be given the computational constraints.

How to do it…

1. Add the following lines to `regressor.py`:

   ```
   from sklearn.preprocessing import PolynomialFeatures

   polynomial = PolynomialFeatures(degree=3)
   ```

2. We initialized a polynomial of the degree `3` in the previous line. Now we have to represent the datapoints in terms of the coefficients of the polynomial:

   ```
   X_train_transformed = polynomial.fit_transform(X_train)
   ```

   Here, `X_train_transformed` represents the same input in the polynomial form.
3. Let's consider the first datapoint in our file and check whether it can predict the right output:

   ```
   datapoint = [0.39,2.78,7.11]
   poly_datapoint = polynomial.fit_transform(datapoint)

   poly_linear_model = linear_model.LinearRegression()
   poly_linear_model.fit(X_train_transformed, y_train)
   ```

```
print "\nLinear regression:",
linear_regressor.predict(datapoint)[0]
print "\nPolynomial regression:",
poly_linear_model.predict(poly_datapoint)[0]
```

The values in the variable datapoint are the values in the first line in the input data file. We are still fitting a linear regression model here. The only difference is in the way in which we represent the data. If you run this code, you will see the following output:

```
Linear regression: -11.0587294983
Polynomial regression: -10.9480782122
```

As you can see, this is close to the output value. If we want it to get closer, we need to increase the degree of the polynomial.

4. Let's make it `10` and see what happens:

```
polynomial = PolynomialFeatures(degree=10)
```

You should see something like the following:

```
Polynomial regression: -8.20472183853
```

Now, you can see that the predicted value is much closer to the actual output value.

Estimating housing prices

It's time to apply our knowledge to a real world problem. Let's apply all these principles to estimate the housing prices. This is one of the most popular examples that is used to understand regression, and it serves as a good entry point. This is intuitive and relatable, hence making it easier to understand concepts before we perform more complex things in machine learning. We will use a **decision tree regressor** with **AdaBoost** to solve this problem.

Getting ready

A decision tree is a tree where each node makes a simple decision that contributes to the final output. The leaf nodes represent the output values, and the branches represent the intermediate decisions that were made, based on input features. AdaBoost stands for Adaptive Boosting, and this is a technique that is used to boost the accuracy of the results from another system. This combines the outputs from different versions of the algorithms, called **weak learners**, using a weighted summation to get the final output. The information that's collected at each stage of the AdaBoost algorithm is fed back into the system so that the learners at the latter stages focus on training samples that are difficult to classify. This is the way it increases the accuracy of the system.

Using AdaBoost, we fit a regressor on the dataset. We compute the error and then fit the regressor on the same dataset again, based on this error estimate. We can think of this as fine-tuning of the regressor until the desired accuracy is achieved. You are given a dataset that contains various parameters that affect the price of a house. Our goal is to estimate the relationship between these parameters and the house price so that we can use this to estimate the price given unknown input parameters.

How to do it…

1. Create a new file called `housing.py`, and add the following lines:

    ```
    import numpy as np
    from sklearn.tree import DecisionTreeRegressor
    from sklearn.ensemble import AdaBoostRegressor
    from sklearn import datasets
    from sklearn.metrics import mean_squared_error,
    explained_variance_score
    from sklearn.utils import shuffle
    import matplotlib.pyplot as plt
    ```

2. There is a standard housing dataset that people tend to use to get started with machine learning. You can download it at https://archive.ics.uci.edu/ml/datasets/Housing. The good thing is that scikit-learn provides a function to directly load this dataset:

    ```
    housing_data = datasets.load_boston()
    ```

    Each datapoint has 13 input parameters that affect the price of the house. You can access the input data using `housing_data.data` and the corresponding price using `housing_data.target`.

3. Let's separate this into input and output. To make this independent of the ordering of the data, let's shuffle it as well:

```
X, y = shuffle(housing_data.data, housing_data.target,
random_state=7)
```

4. The `random_state` parameter controls how we shuffle the data so that we can have reproducible results. Let's divide the data into training and testing. We'll allocate 80% for training and 20% for testing:

```
num_training = int(0.8 * len(X))
X_train, y_train = X[:num_training], y[:num_training]
X_test, y_test = X[num_training:], y[num_training:]
```

5. We are now ready to fit a decision tree regression model. Let's pick a tree with a maximum depth of `4`, which means that we are not letting the tree become arbitrarily deep:

```
dt_regressor = DecisionTreeRegressor(max_depth=4)
dt_regressor.fit(X_train, y_train)
```

6. Let's also fit decision tree regression model with AdaBoost:

```
ab_regressor =
AdaBoostRegressor(DecisionTreeRegressor(max_depth=4),
n_estimators=400, random_state=7)
ab_regressor.fit(X_train, y_train)
```

   This will help us compare the results and see how AdaBoost really boosts the performance of a decision tree regressor.

7. Let's evaluate the performance of decision tree regressor:

```
y_pred_dt = dt_regressor.predict(X_test)
mse = mean_squared_error(y_test, y_pred_dt)
evs = explained_variance_score(y_test, y_pred_dt)
print "\n#### Decision Tree performance ####"
print "Mean squared error =", round(mse, 2)
print "Explained variance score =", round(evs, 2)
```

8. Now, let's evaluate the performance of AdaBoost:

```
y_pred_ab = ab_regressor.predict(X_test)
mse = mean_squared_error(y_test, y_pred_ab)
evs = explained_variance_score(y_test, y_pred_ab)
print "\n#### AdaBoost performance ####"
print "Mean squared error =", round(mse, 2)
print "Explained variance score =", round(evs, 2)
```

Here is the output on the Terminal:

```
#### Decision Tree performance ####
Mean squared error = 14.79
Explained variance score = 0.82

#### AdaBoost performance ####
Mean squared error = 7.54
Explained variance score = 0.91
```

The error is lower and the variance score is closer to 1 when we use AdaBoost as shown in the preceding output.

Computing the relative importance of features

Are all the features equally important? In this case, we used 13 input features, and they all contributed to the model. However, an important question here is, "How do we know which features are more important?" Obviously, all the features don't contribute equally to the output. In case we want to discard some of them later, we need to know which features are less important. We have this functionality available in scikit-learn.

How to do it…

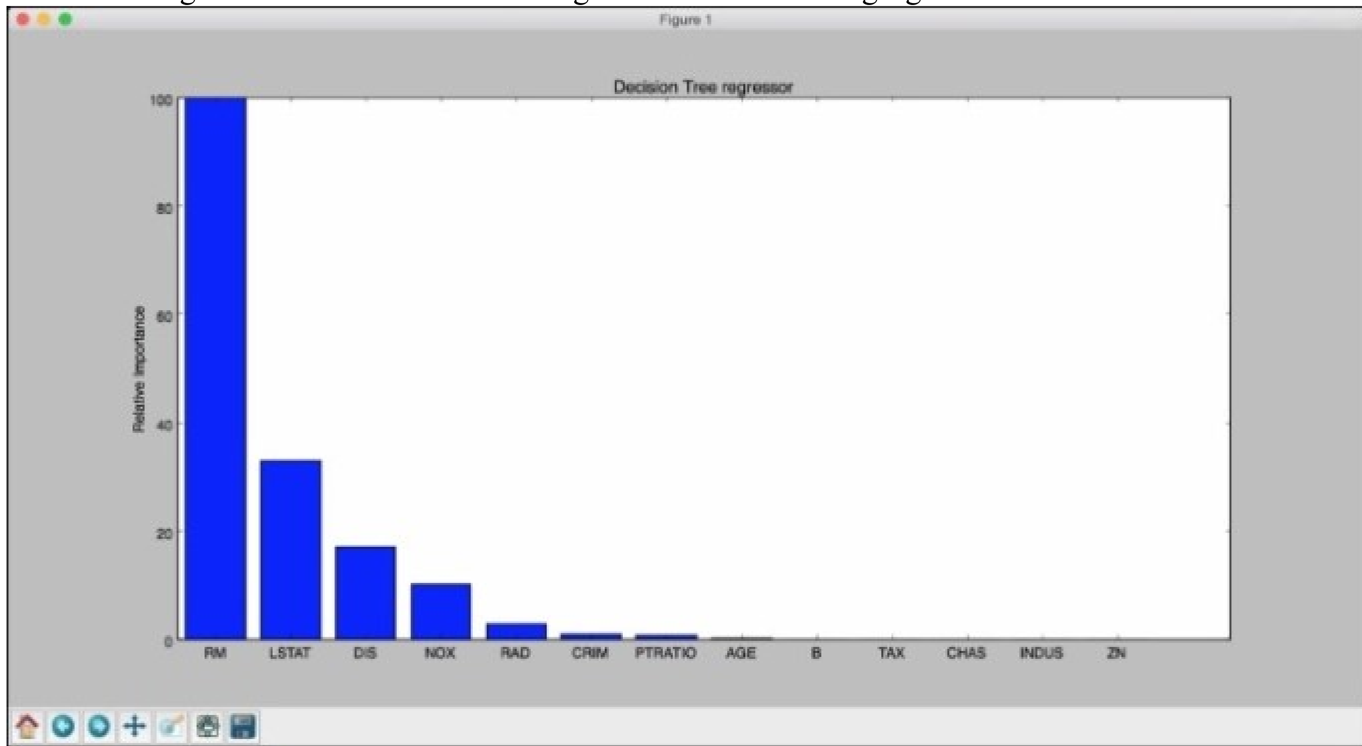
1. Let's plot the relative importance of the features. Add the following lines to `housing.py`:

   ```
   plot_feature_importances(dt_regressor.feature_importances_,
           'Decision Tree regressor', housing_data.feature_names)
   plot_feature_importances(ab_regressor.feature_importances_,
           'AdaBoost regressor', housing_data.feature_names)
   ```

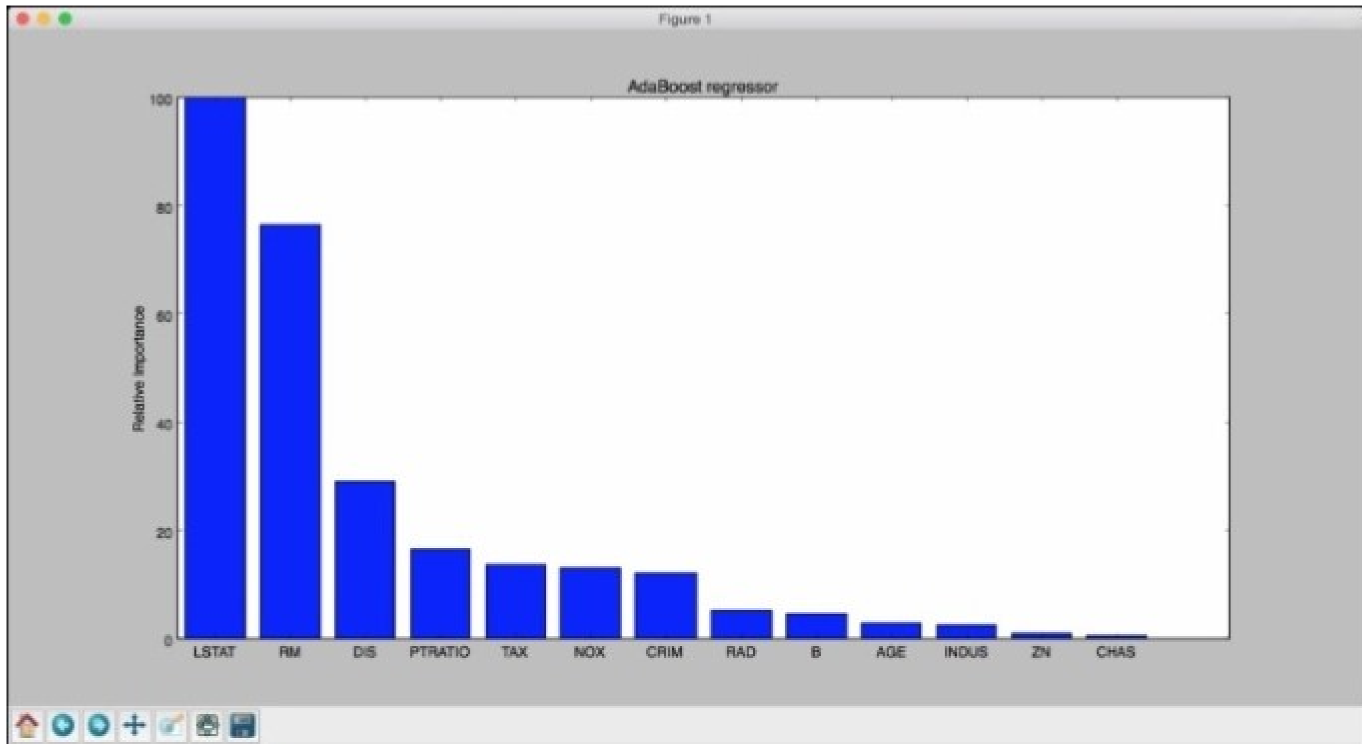
   The regressor object has a callable `feature_importances_` method that gives us the relative importance of each feature.
2. We actually need to define our `plot_feature_importances` function to plot the bar graphs:

   ```
   def plot_feature_importances(feature_importances, title, 
   feature_names):
       # Normalize the importance values 
       feature_importances = 100.0 * (feature_importances / 
   max(feature_importances))

       # Sort the index values and flip them so that they are 
   arranged in decreasing order of importance
       index_sorted = np.flipud(np.argsort(feature_importances))

       # Center the location of the labels on the X-axis (for 
   display purposes only)
       pos = np.arange(index_sorted.shape[0]) + 0.5

       # Plot the bar graph
       plt.figure()
       plt.bar(pos, feature_importances[index_sorted], 
   align='center')
       plt.xticks(pos, feature_names[index_sorted])
       plt.ylabel('Relative Importance')
       plt.title(title)
       plt.show()
   ```

3. We just take the values from the `feature_importances_` method and scale it so that it ranges between 0 and 100. If you run the preceding code, you will see two figures. Let's see what we will get for a decision tree-based regressor in the following figure:



4. So, the decision tree regressor says that the most important feature is RM. Let's take a look at what AdaBoost has to say in the following figure:



According to AdaBoost, the most important feature is LSTAT. In reality, if you build various regressors on this data, you will see that the most important feature is in fact LSTAT. This shows the advantage of using AdaBoost with a decision tree-based regressor.

Estimating bicycle demand distribution

Let's use a different regression method to solve the bicycle demand distribution problem. We will use the **random forest regressor** to estimate the output values. A random forest is a collection of decision trees. This basically uses a set of decision trees that are built using various subsets of the dataset, and then it uses averaging to improve the overall performance.

Getting ready

We will use the `bike_day.csv` file that is provided to you. This is also available at https://archive.ics.uci.edu/ml/datasets/Bike+Sharing+Dataset. There are 16 columns in this dataset. The first two columns correspond to the serial number and the actual date, so we won't use them for our analysis. The last three columns correspond to different types of outputs. The last column is just the sum of the values in the fourteenth and fifteenth columns, so we can leave these two out when we build our model.

How to do it…

Let's go ahead and see how to do this in Python. You have been provided with a file called `bike_sharing.py` that contains the full code. We will discuss the important parts of this, as follows:

1. We first need to import a couple of new packages, as follows:

```
import csv
from sklearn.ensemble import RandomForestRegressor
from housing import plot_feature_importances
```

2. We are processing a CSV file, so the CSV package is useful in handling these files. As it's a new dataset, we will have to define our own dataset loading function:

```
def load_dataset(filename):
    file_reader = csv.reader(open(filename, 'rb'),
delimiter=',')
    X, y = [], []
    for row in file_reader:
        X.append(row[2:13])
        y.append(row[-1])

    # Extract feature names
    feature_names = np.array(X[0])

    # Remove the first row because they are feature names
    return np.array(X[1:]).astype(np.float32),
np.array(y[1:]).astype(np.float32), feature_names
```

In this function, we just read all the data from the CSV file. The feature names are useful when we display it on a graph. We separate the data from the output values and return them.

3. Let's read the data and shuffle it to make it independent of the order in which the data is arranged in the file:

```
X, y, feature_names = load_dataset(sys.argv[1])
X, y = shuffle(X, y, random_state=7)
```

4. As we did earlier, we need to separate the data into training and testing. This time, let's use 90% of the data for training and the remaining 10% for testing:

```
num_training = int(0.9 * len(X))
X_train, y_train = X[:num_training], y[:num_training]
X_test, y_test = X[num_training:], y[num_training:]
```

5. Let's go ahead and train the regressor:

```
rf_regressor = RandomForestRegressor(n_estimators=1000,
max_depth=10, min_samples_split=1)
rf_regressor.fit(X_train, y_train)
```

   Here, `n_estimators` refers to the number of estimators, which is the number of decision trees that we want to use in our random forest. The `max_depth` parameter refers to the maximum depth of each tree, and the `min_samples_split` parameter refers to the number of data samples that are needed to split a node in the tree.

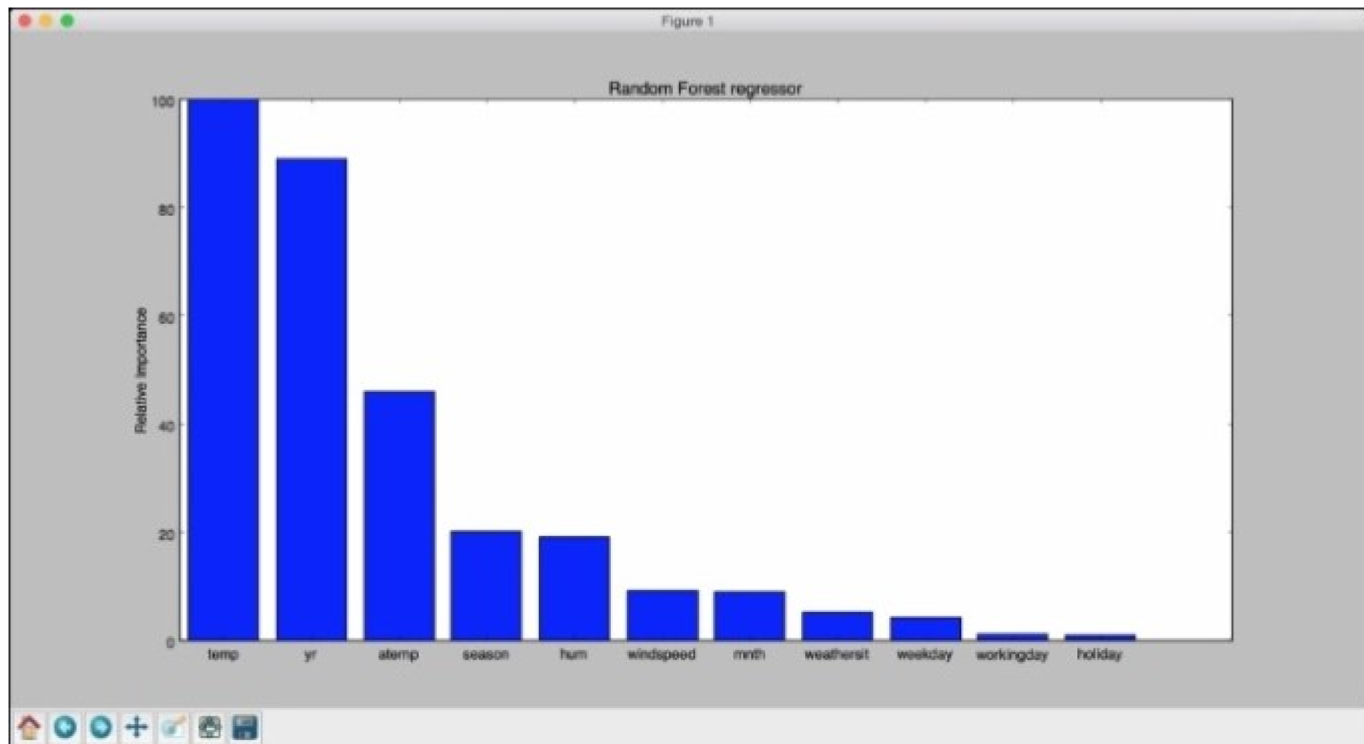
6. Let's evaluate performance of the random forest regressor:

```
y_pred = rf_regressor.predict(X_test)
mse = mean_squared_error(y_test, y_pred)
evs = explained_variance_score(y_test, y_pred)
print "\n#### Random Forest regressor performance ####"
print "Mean squared error =", round(mse, 2)
print "Explained variance score =", round(evs, 2)
```

7. As we already have the function to plot the `importances` feature, let's just call it directly:

```
plot_feature_importances(rf_regressor.feature_importances_,
'Random Forest regressor', feature_names)
```

   Once you run this code, you will see the following graph:



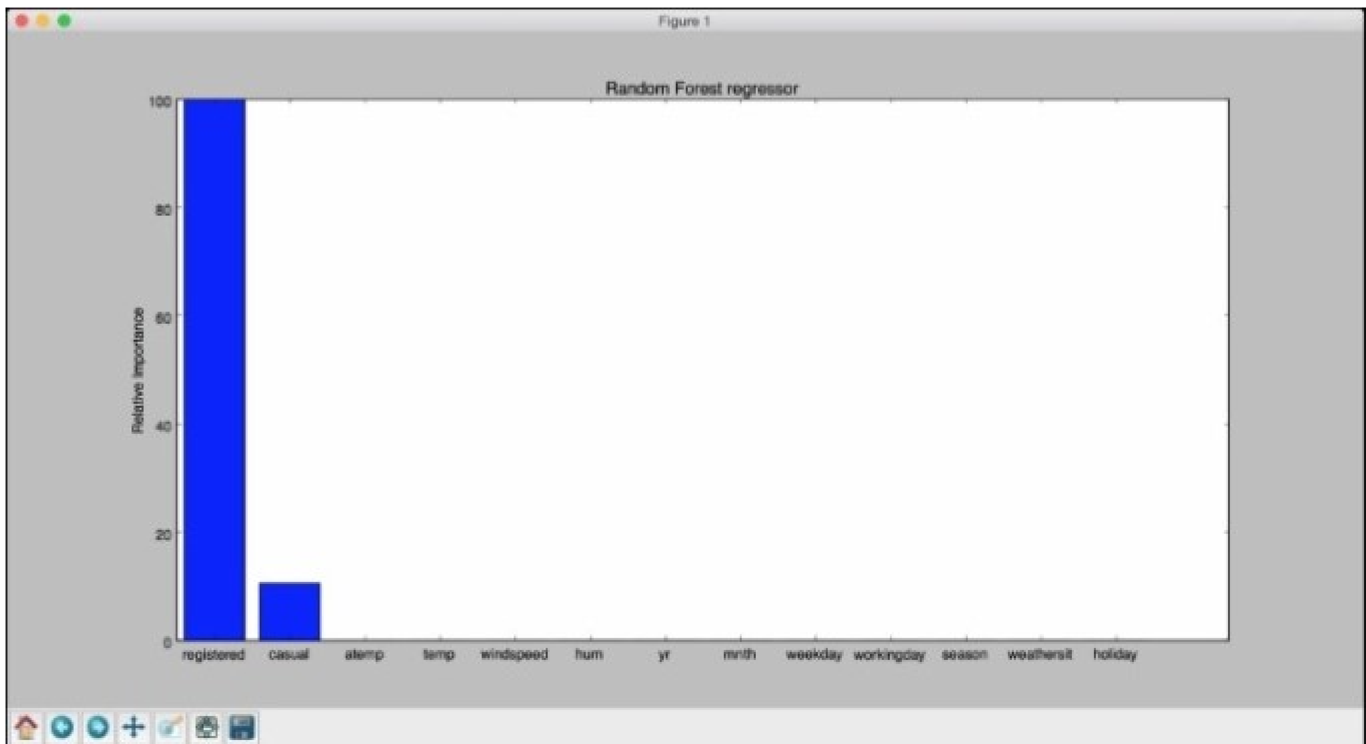
Looks like the temperature is the most important factor controlling the bicycle rentals.

There's more…

Let's see what happens when you include fourteenth and fifteenth columns in the dataset. In the feature importance graph, every feature other than these two has to go to zero. The reason is that the output can be obtained by simply summing up the fourteenth and fifteenth columns, so the algorithm doesn't need any other features to compute the output. In the `load_dataset` function, make the following change inside the for loop:

```
X.append(row[2:15])
```

If you plot the feature importance graph now, you will see the following:



As expected, it says that only these two features are important. This makes sense intuitively because the final output is a simple summation of these two features. So, there is a direct relationship between these two variables and the output value. Hence, the regressor says that it doesn't need any other variable to predict the output. This is an extremely useful tool to eliminate redundant variables in your dataset.

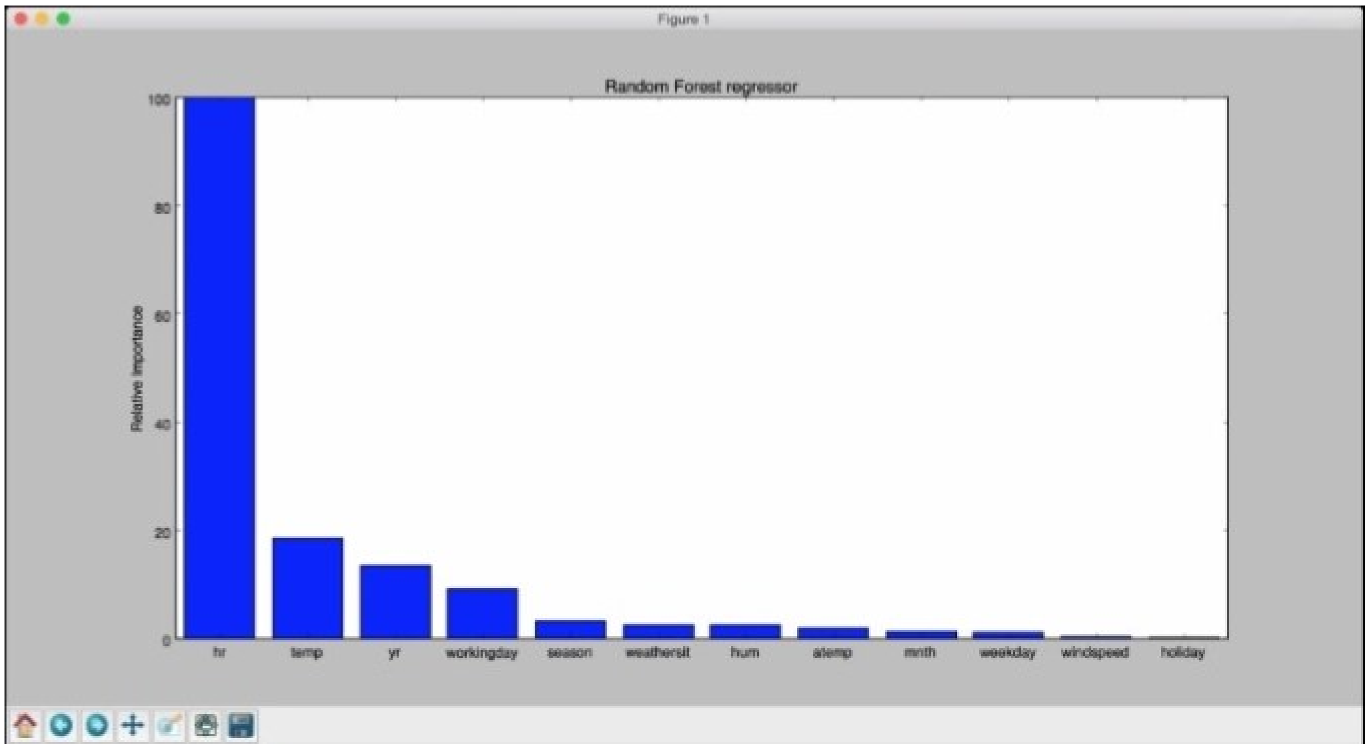
There is another file called `bike_hour.csv` that contains data about how the bicycles are shared hourly. We need to consider columns 3 to 14, so let's make this change inside the `load_dataset` function:

```
X.append(row[2:14])
```

If you run this, you will see the performance of the regressor displayed, as follows:

```
#### Random Forest regressor performance ####
Mean squared error = 2619.87
Explained variance score = 0.92
```

The feature importance graph will look like the following:



This shows that the hour of the day is the most important feature, which makes sense intuitively if you think about it! The next important feature is temperature, which is consistent with our earlier analysis.

Chapter 2. Constructing a Classifier

In this chapter, we will cover the following recipes:

- Building a simple classifier
- Building a logistic regression classifier
- Building a Naïve Bayes classifier
- Splitting the dataset for training and testing
- Evaluating the accuracy using cross-validation
- Visualizing the confusion matrix
- Extracting the performance report
- Evaluating cars based on their characteristics
- Extracting validation curves
- Extracting learning curves
- Estimating the income bracket

Introduction

In the field of machine learning, classification refers to the process of using the characteristics of data to separate it into a certain number of classes. This is different from regression that we discussed in the previous chapter where the output is a real number. A supervised learning classifier builds a model using labeled training data and then uses this model to classify unknown data.

A classifier can be any algorithm that implements classification. In simple cases, this classifier can be a straightforward mathematical function. In more real-world cases, this classifier can take very complex forms. In the course of study, we will see that classification can be either binary, where we separate data into two classes, or it can be multiclass, where we separate data into more than two classes. The mathematical techniques that are devised to deal with the classification problem tend to deal with two classes, so we extend them in different ways to deal with the multiclass problem as well.

Evaluating the accuracy of a classifier is an important step in the world machine learning. We need to learn how to use the available data to get an idea as to how this model will perform in the real world. In this chapter, we will look at recipes that deal with all these things.

Building a simple classifier

Let's see how to build a simple classifier using some training data.

How to do it…

1. We will use the `simple_classifier.py` file that is already provided to you as reference. Assuming that you imported the `numpy` and `matplotlib.pyplot` packages like we did in the last chapter, let's create some sample data:

   ```
   X = np.array([[3,1], [2,5], [1,8], [6,4], [5,2], [3,5], [4,7],
   [4,-1]])
   ```

2. Let's assign some labels to these points:

   ```
   y = [0, 1, 1, 0, 0, 1, 1, 0]
   ```

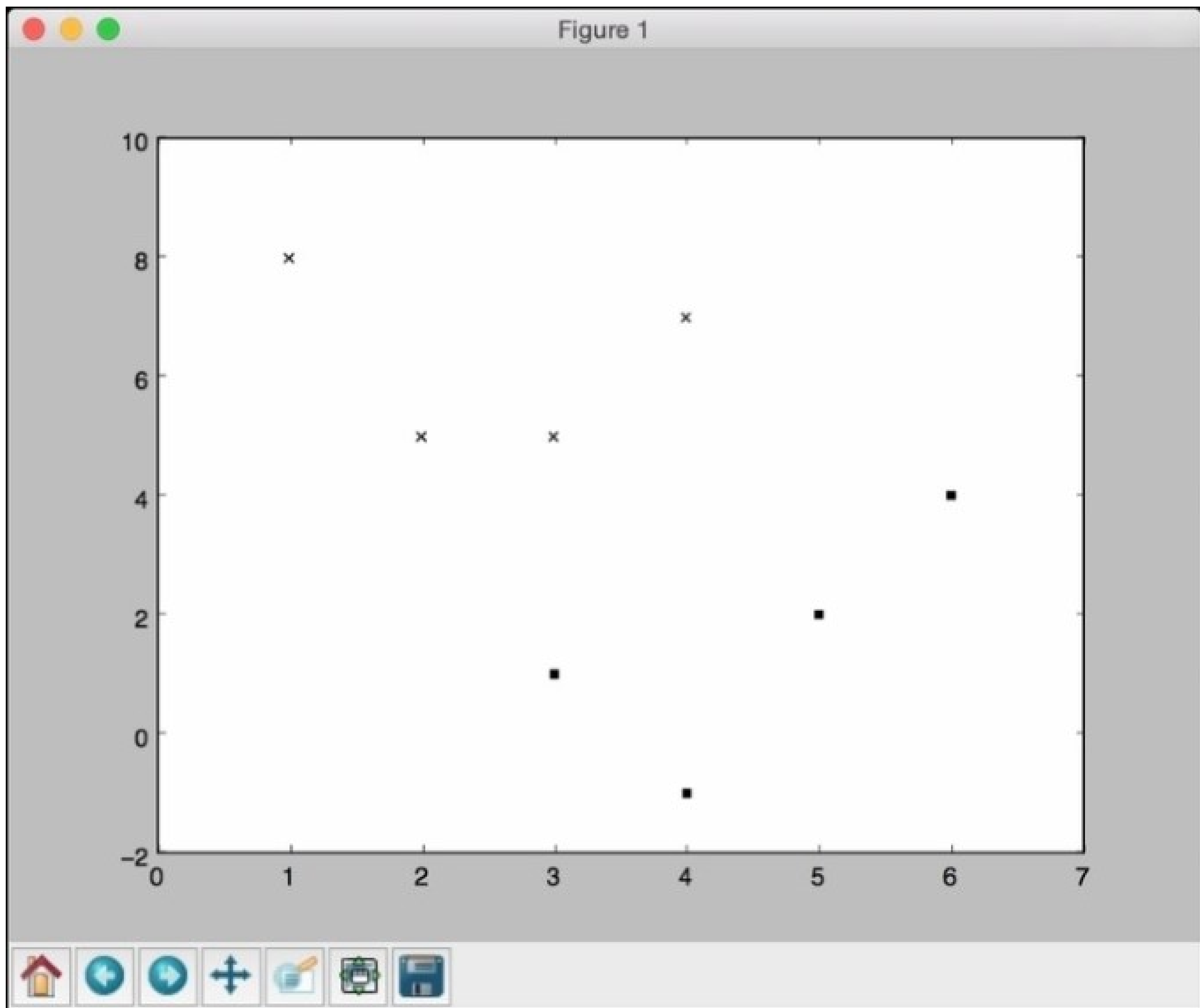
3. As we have only two classes, the `y` list contains 0s and 1s. In general, if you have *N* classes, then the values in `y` will range from 0 to *N*-1. Let's separate the data into classes based on the labels:

   ```
   class_0 = np.array([X[i] for i in range(len(X)) if y[i]==0])
   class_1 = np.array([X[i] for i in range(len(X)) if y[i]==1])
   ```

4. To get an idea about our data, let's plot it, as follows:

   ```
   plt.figure()
   plt.scatter(class_0[:,0], class_0[:,1], color='black',
   marker='s')
   plt.scatter(class_1[:,0], class_1[:,1], color='black',
   marker='x')
   ```

   This is a scatterplot, where we use squares and crosses to plot the points. In this context, the `marker` parameter specifies the shape you want to use. We use squares to denote points in `class_0` and crosses to denote points in `class_1`. If you run this code, you will see the following figure:



5. In the preceding two lines, we just use the mapping between `X` and `y` to create two lists. If you were asked to inspect the datapoints visually and draw a separating line, what would you do? You will simply draw a line in between them. Let's go ahead and do this:

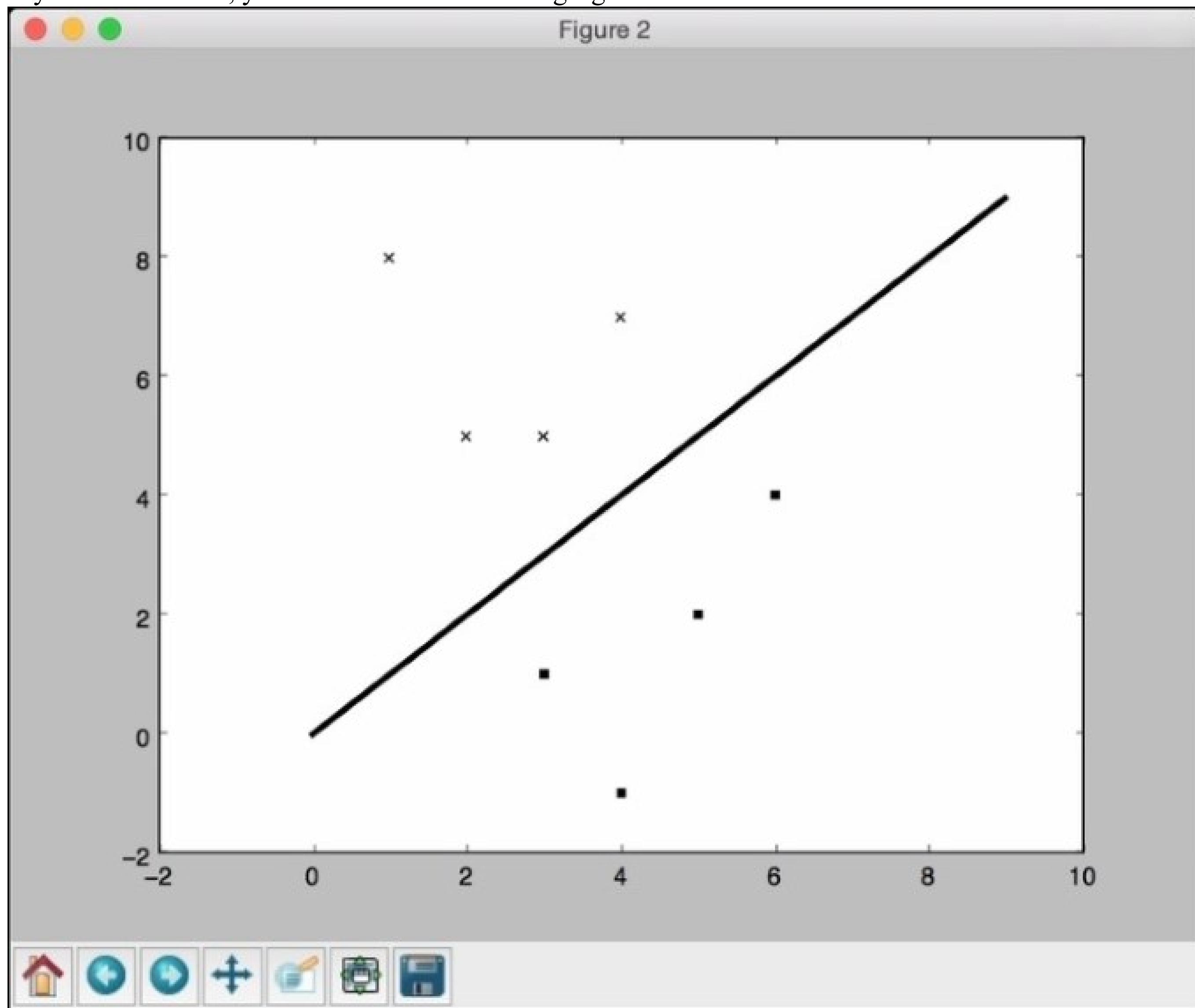
```
line_x = range(10)
line_y = line_x
```

6. We just created a line with the mathematical equation $y = x$. Let's plot it, as follows:

```
plt.figure()
plt.scatter(class_0[:,0], class_0[:,1], color='black',
marker='s')
plt.scatter(class_1[:,0], class_1[:,1], color='black',
marker='x')
```

```
plt.plot(line_x, line_y, color='black', linewidth=3)
plt.show()
```

7. If you run this code, you should see the following figure:



There's more…

We built a simple classifier using the following rule: the input point (`a`, `b`) belongs to `class_0` if `a` is greater than or equal to `b`; otherwise, it belongs to `class_1`. If you inspect the points one by one, you will see that this is, in fact, true. This is it! You just built a linear classifier that can classify unknown data. It's a linear classifier because the separating line is a straight line. If it's a curve, then it becomes a nonlinear classifier.

This formation worked fine because there were a limited number of points, and we could visually inspect them. What if there are thousands of points? How do we generalize this process? Let's discuss that in the next recipe.

Building a logistic regression classifier

Despite the word *regression* being present in the name, logistic regression is actually used for classification purposes. Given a set of datapoints, our goal is to build a model that can draw linear boundaries between our classes. It extracts these boundaries by solving a set of equations derived from the training data.

How to do it…

1. Let's see how to do this in Python. We will use the `logistic_regression.py` file that is provided to you as a reference. Assuming that you imported the necessary packages, let's create some sample data along with training labels:

   ```
   import numpy as np
   from sklearn import linear_model
   import matplotlib.pyplot as plt

   X = np.array([[4, 7], [3.5, 8], [3.1, 6.2], [0.5, 1], [1, 2],
   [1.2, 1.9], [6, 2], [5.7, 1.5], [5.4, 2.2]])
   y = np.array([0, 0, 0, 1, 1, 1, 2, 2, 2])
   ```

   Here, we assume that we have three classes.
2. Let's initialize the logistic regression classifier:

   ```
   classifier =
   linear_model.LogisticRegression(solver='liblinear', C=100)
   ```

   There are a number of input parameters that can be specified for the preceding function, but a couple of important ones are `solver` and `C`. The `solver` parameter specifies the type of solver that the algorithm will use to solve the system of equations. The `C` parameter controls the regularization strength. A lower value indicates higher regularization strength.
3. Let's train the classifier:

   ```
   classifier.fit(X, y)
   ```

4. Let's draw datapoints and boundaries:

   ```
   plot_classifier(classifier, X, y)
   ```

   We need to define this function, as follows:

   ```
   def plot_classifier(classifier, X, y):
       # define ranges to plot the figure
       x_min, x_max = min(X[:, 0]) - 1.0, max(X[:, 0]) + 1.0
       y_min, y_max = min(X[:, 1]) - 1.0, max(X[:, 1]) + 1.0
   ```

The preceding values indicate the range of values that we want to use in our figure. The values usually range from the minimum value to the maximum value present in our data. We add some buffers, such as 1.0 in the preceding lines, for clarity.

5. In order to plot the boundaries, we need to evaluate the function across a grid of points and plot it. Let's go ahead and define the grid:

```
    # denotes the step size that will be used in the mesh grid
    step_size = 0.01

    # define the mesh grid
    x_values, y_values = np.meshgrid(np.arange(x_min, x_max,
step_size), np.arange(y_min, y_max, step_size))
```

   The `x_values` and `y_values` variables contain the grid of points where the function will be evaluated.

6. Let's compute the output of the classifier for all these points:

```
    # compute the classifier output
    mesh_output = classifier.predict(np.c_[x_values.ravel(),
y_values.ravel()])

    # reshape the array
    mesh_output = mesh_output.reshape(x_values.shape)
```

7. Let's plot the boundaries using colored regions:

```
    # Plot the output using a colored plot
    plt.figure()

    # choose a color scheme
    plt.pcolormesh(x_values, y_values, mesh_output,
cmap=plt.cm.gray)
```

   This is basically a 3D plotter that takes the 2D points and the associated values to draw different regions using a color scheme. You can find all the color scheme options at http://matplotlib.org/examples/color/colormaps_reference.html.

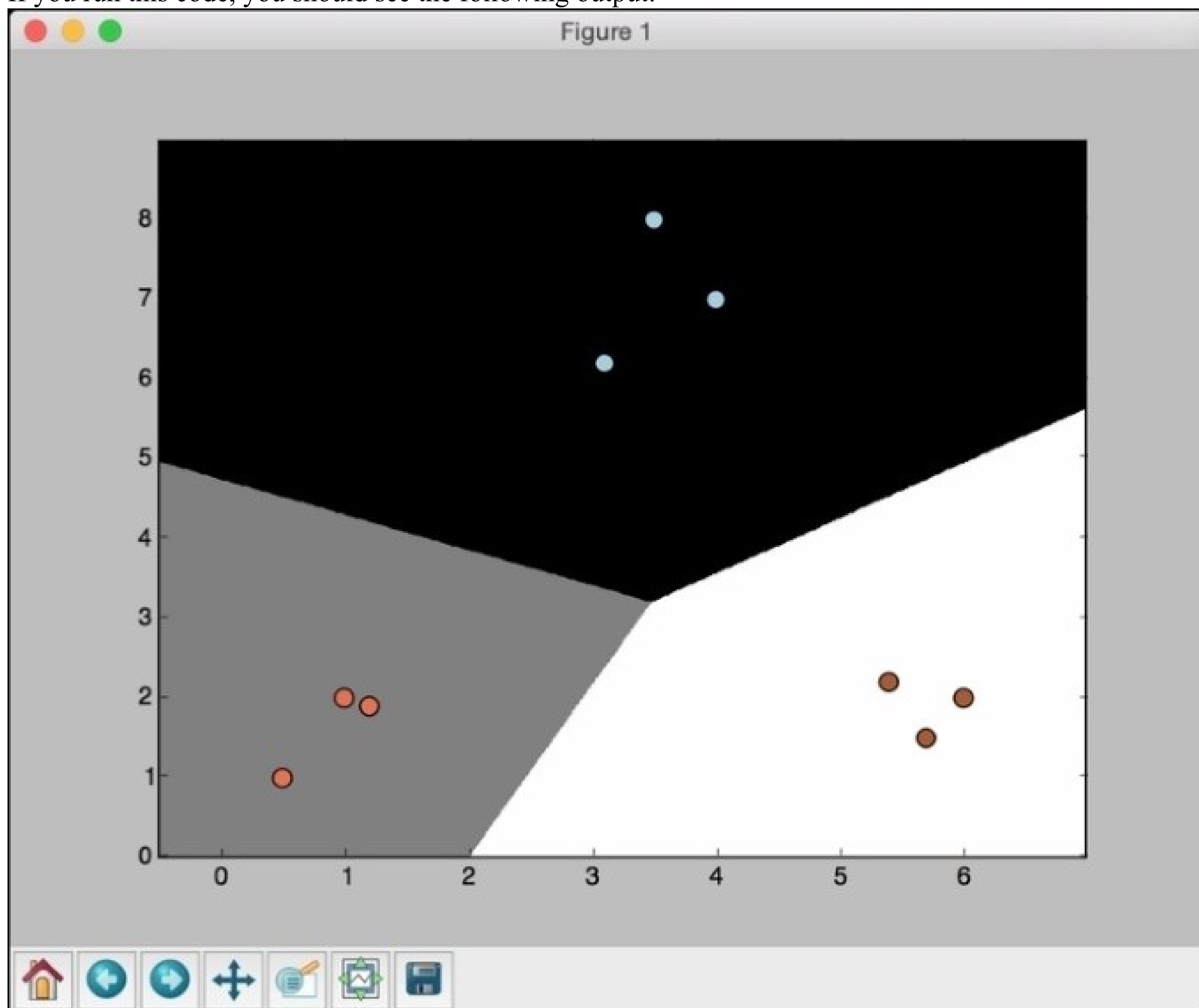
8. Let's overlay the training points on the plot:

```
    plt.scatter(X[:, 0], X[:, 1], c=y, s=80,
edgecolors='black', linewidth=1, cmap=plt.cm.Paired)

    # specify the boundaries of the figure
    plt.xlim(x_values.min(), x_values.max())
    plt.ylim(y_values.min(), y_values.max())

    # specify the ticks on the X and Y axes
    plt.xticks((np.arange(int(min(X[:, 0])-1), int(max(X[:,
0])+1), 1.0)))
```

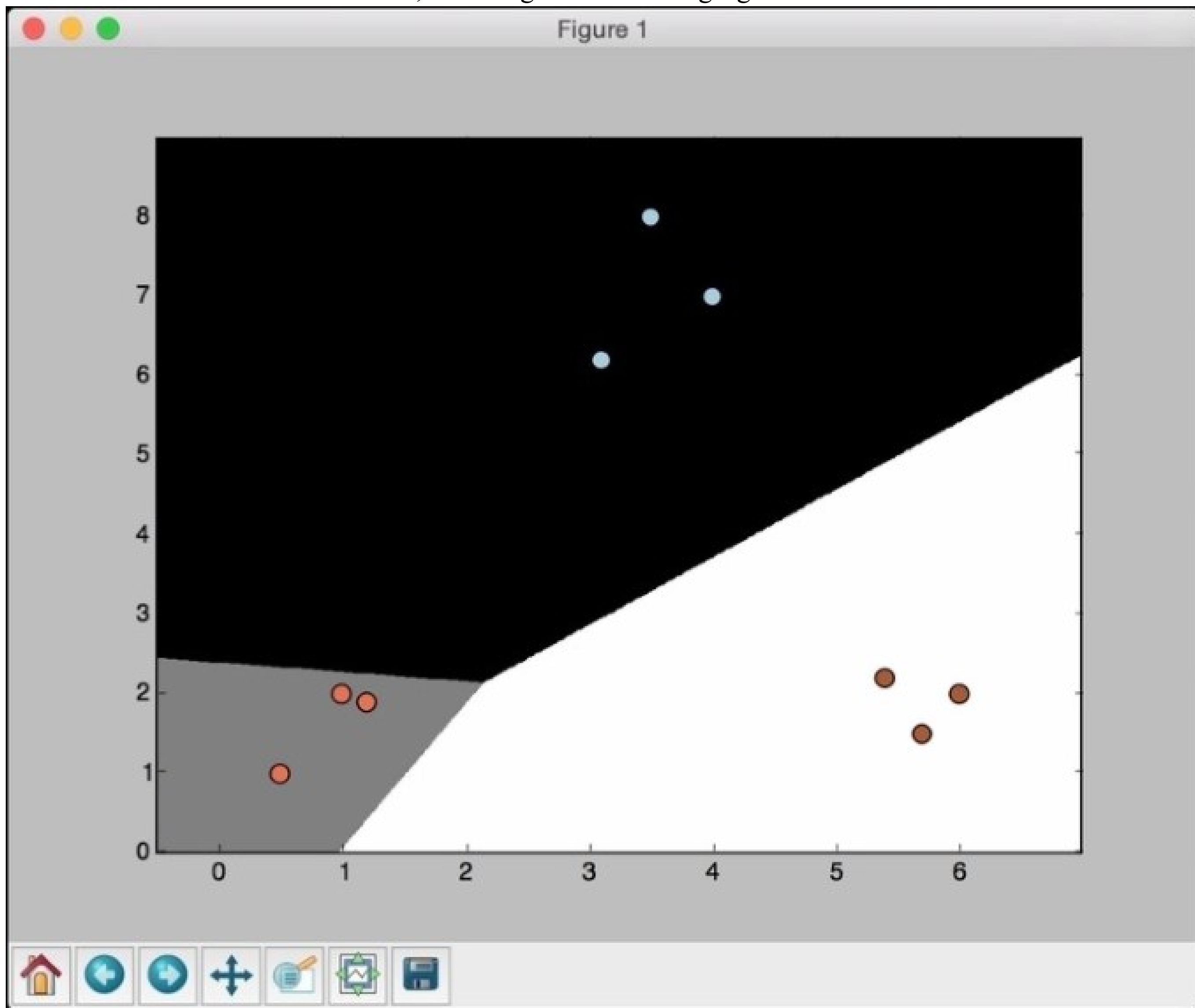
```
    plt.yticks((np.arange(int(min(X[:, 1])-1), int(max(X[:,
1])+1), 1.0)))

    plt.show()
```

Here, `plt.scatter` plots the points on the 2D graph. `X[:, 0]` specifies that we should take all the values along axis 0 (X-axis in our case) and `X[:, 1]` specifies axis 1 (Y-axis). The `c=y` parameter indicates the color sequence. We use the target labels to map to colors using `cmap`. Basically, we want different colors that are based on the target labels. Hence, we use `y` as the mapping. The limits of the display figure are set using `plt.xlim` and `plt.ylim`. In order to mark the axes with values, we need to use `plt.xticks` and `plt.yticks`. These functions mark the axes with values so that it's easier for us to see where the points are located. In the preceding code, we want the ticks to lie between the minimum and maximum values with a buffer of one unit. Also, we want these ticks to be integers. So, we use `int()` function to round off the values.

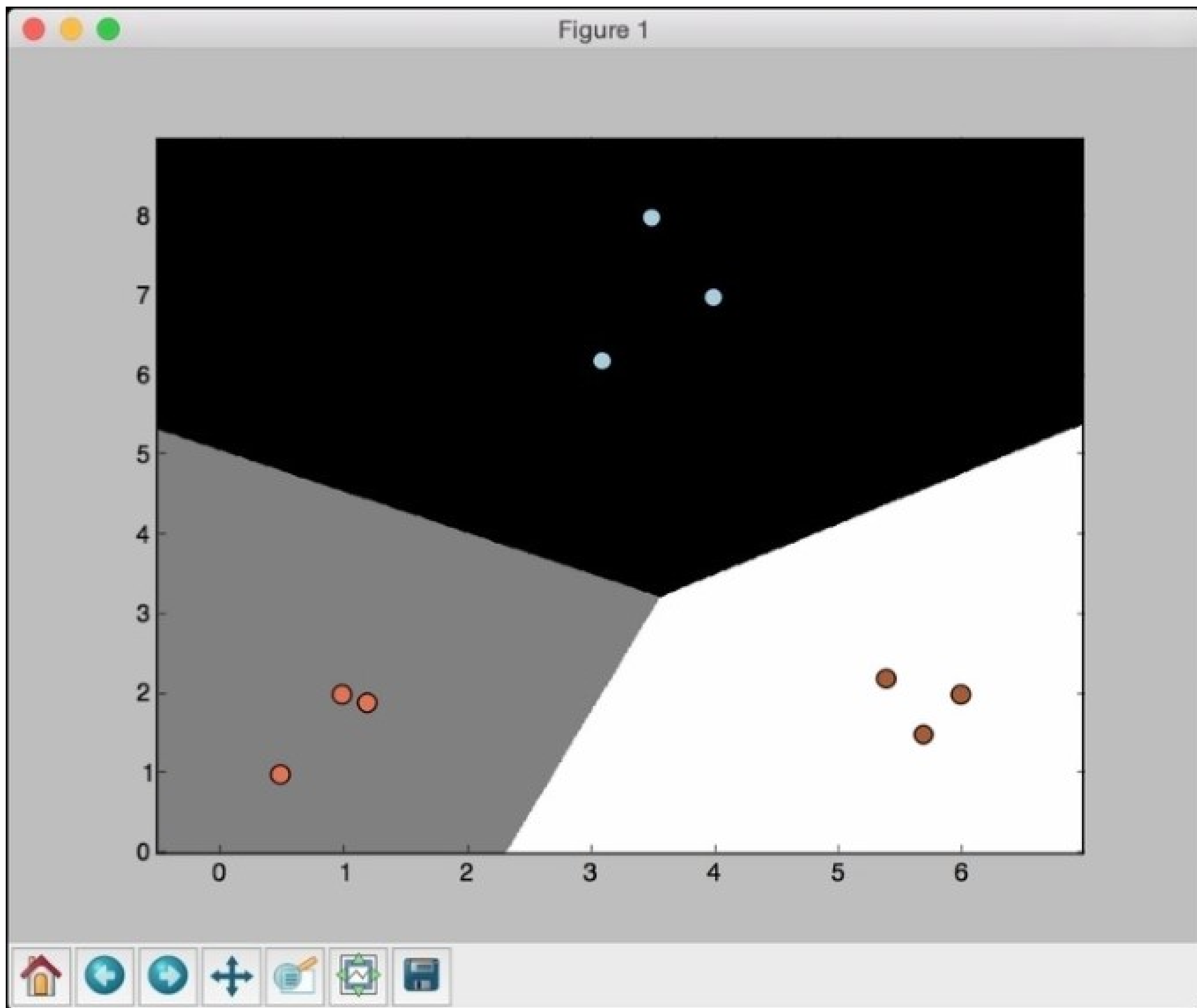
9. If you run this code, you should see the following output:



10. Let's see how the `C` parameter affects our model. The `C` parameter indicates the penalty for misclassification. If we set it to `1.0`, we will get the following figure:



11. If we set `C` to `10000`, we get the following figure:



As we increase C, there is a higher penalty for misclassification. Hence, the boundaries get more optimal.

Building a Naive Bayes classifier

A Naive Bayes classifier is a supervised learning classifier that uses Bayes' theorem to build the model. Let's go ahead and build a Naïve Bayes classifier.

How to do it…

1. We will use `naive_bayes.py` that is provided to you as reference. Let's import a couple of things:

   ```
   from sklearn.naive_bayes import GaussianNB
   from logistic_regression import plot_classifier
   ```

2. You were provided with a `data_multivar.txt` file. This contains data that we will use here. This contains comma-separated numerical data in each line. Let's load the data from this file:

   ```
   input_file = 'data_multivar.txt'

   X = []
   y = []
   with open(input_file, 'r') as f:
       for line in f.readlines():
           data = [float(x) for x in line.split(',')]
           X.append(data[:-1])
           y.append(data[-1])

   X = np.array(X)
   y = np.array(y)
   ```

   We have now loaded the input data into `X` and the labels into `y`.
3. Let's build the Naive Bayes classifier:

   ```
   classifier_gaussiannb = GaussianNB()
   classifier_gaussiannb.fit(X, y)
   y_pred = classifier_gaussiannb.predict(X)
   ```

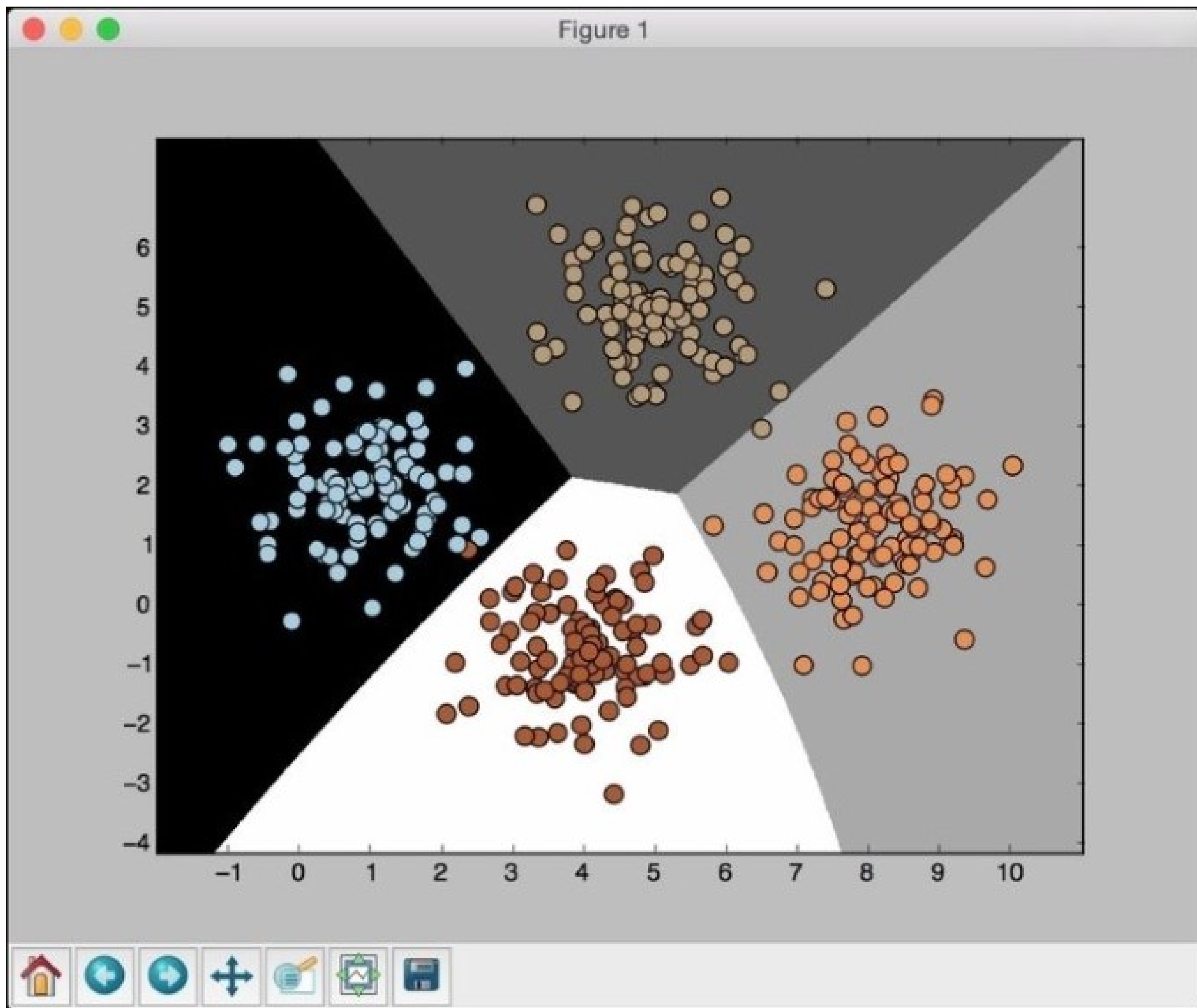
   The `GaussianNB` function specifies Gaussian Naive Bayes model.
4. Let's compute the accuracy of the classifier:

   ```
   accuracy = 100.0 * (y == y_pred).sum() / X.shape[0]
   print "Accuracy of the classifier =", round(accuracy, 2), "%"
   ```

5. Let's plot the data and the boundaries:

   ```
   plot_classifier(classifier_gaussiannb, X, y)
   ```

   You should see the following figure:



There is no restriction on the boundaries to be linear here. In the preceding example, we used up all the data for training. A good practice in machine learning is to have nonoverlapping data for training and testing. Ideally, we need some unused data for testing so that we can get an accurate estimate of how the model performs on unknown data. There is a provision in scikit-learn that handles this very well, as shown in the next recipe.

Splitting the dataset for training and testing

Let's see how to split our data properly into training and testing datasets.

How to do it…

1. Add the following code snippet into the same Python file as the previous recipe:

   ```
   from sklearn import cross_validation

   X_train, X_test, y_train, y_test =
   cross_validation.train_test_split(X, y, test_size=0.25,
   random_state=5)
   classifier_gaussiannb_new = GaussianNB()
   classifier_gaussiannb_new.fit(X_train, y_train)
   ```

   Here, we allocated 25% of the data for testing, as specified by the `test_size` parameter. The remaining 75% of the data will be used for training.
2. Let's evaluate the classifier on test data:

   ```
   y_test_pred = classifier_gaussiannb_new.predict(X_test)
   ```

3. Let's compute the accuracy of the classifier:

   ```
   accuracy = 100.0 * (y_test == y_test_pred).sum() /
   X_test.shape[0]
   print "Accuracy of the classifier =", round(accuracy, 2), "%"
   ```

4. Let's plot the datapoints and the boundaries on test data:

   ```
   plot_classifier(classifier_gaussiannb_new, X_test, y_test)
   ```

5. You should see the following figure:

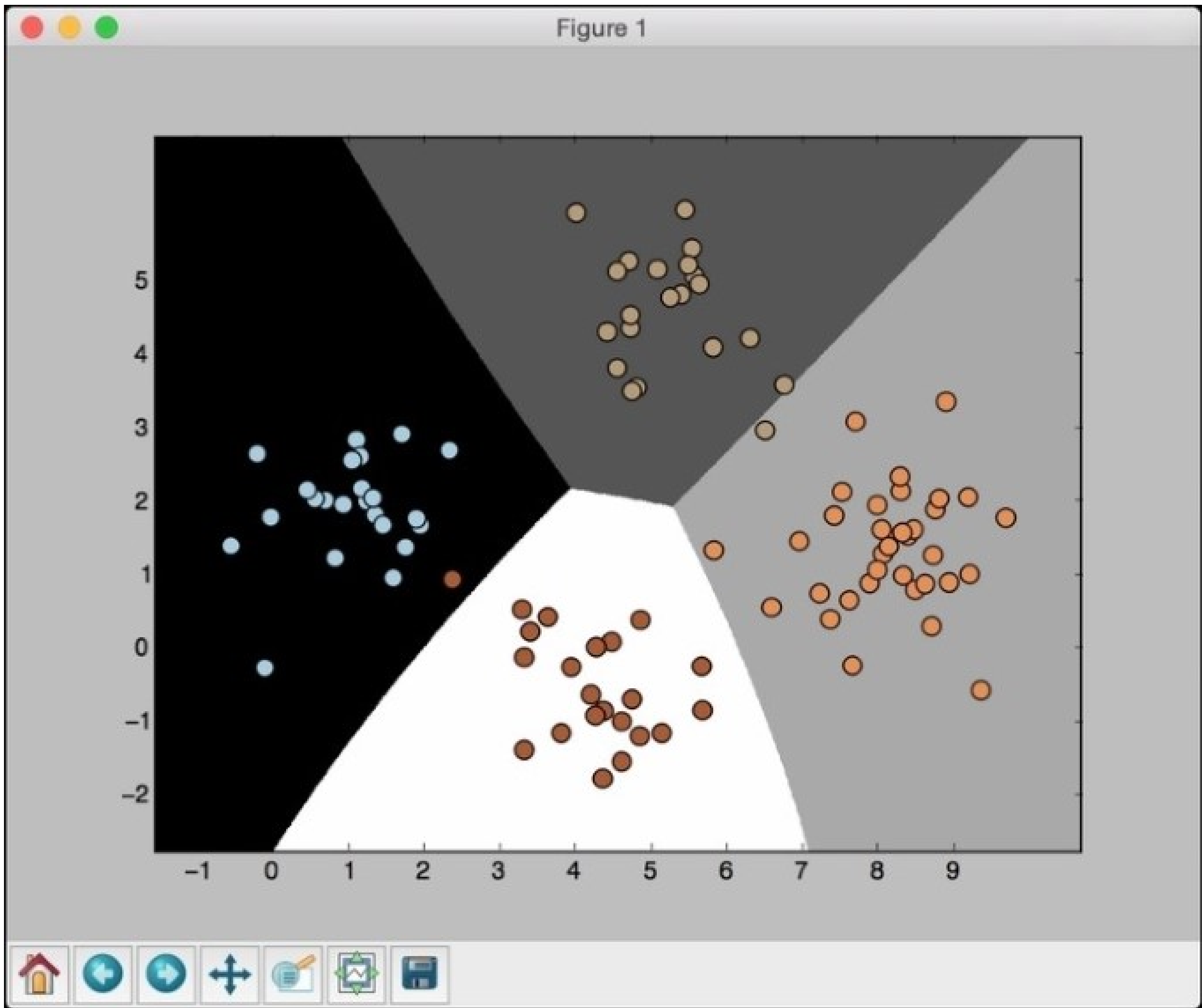


Figure 1
5
4
3
2
1
0
−1
−2
−1
0
1
2
3
4
5
6
7
8
9

Evaluating the accuracy using cross-validation

The **cross-validation** is an important concept in machine learning. In the previous recipe, we split the data into training and testing datasets. However, in order to make it more robust, we need to repeat this process with different subsets. If we just fine-tune it for a particular subset, we may end up overfitting the model. Overfitting refers to a situation where we fine-tune a model too much to a dataset and it fails to perform well on unknown data. We want our machine learning model to perform well on unknown data.

Getting ready…

Before we discuss how to perform cross-validation, let's talk about performance metrics. When we are dealing with machine learning models, we usually care about three things: precision, recall, and F1 score. We can get the required performance metric using the parameter scoring. Precision refers to the number of items that are correctly classified as a percentage of the overall number of items in the list. Recall refers to the number of items that are retrieved as a percentage of the overall number of items in the training list.

Let's consider a test dataset containing 100 items, out of which 82 are of interest to us. Now, we want our classifier to identify these 82 items for us. Our classifier picks out 73 items as the items of interest. Out of these 73 items, only 65 are actually the items of interest and the remaining eight are misclassified. We can compute precision in the following way:

- The number of correct identifications = 65
- The total number of identifications = 73
- Precision = 65 / 73 = 89.04%

To compute recall, we use the following:

- The total number of interesting items in the dataset = 82
- The number of items retrieved correctly = 65
- Recall = 65 / 82 = 79.26%

A good machine learning model needs to have good precision and good recall simultaneously. It's easy to get one of them to 100%, but the other metric suffers! We need to keep both the metrics high at the same time. To quantify this, we use an F1 score, which is a combination of precision and recall. This is actually the harmonic mean of precision and recall:

*F1 score = 2 * precision * recall / (precision + recall)*

In the preceding case, the F1 score will be as follows:

*F1 score = 2 * 0.89 * 0.79 / (0.89 + 0.79) = 0.8370*

How to do it…

1. Let's see how to perform cross-validation and extract performance metrics. We will start with the accuracy:

```
num_validations = 5
accuracy =
cross_validation.cross_val_score(classifier_gaussiannb,
        X, y, scoring='accuracy', cv=num_validations)
print "Accuracy: " + str(round(100*accuracy.mean(), 2)) + "%"
```

2. We will use the preceding function to compute precision, recall, and the F1 score as well:

```
f1 = cross_validation.cross_val_score(classifier_gaussiannb,
        X, y, scoring='f1_weighted', cv=num_validations)
print "F1: " + str(round(100*f1.mean(), 2)) + "%"

precision =
cross_validation.cross_val_score(classifier_gaussiannb,
        X, y, scoring='precision_weighted', cv=num_validations)
print "Precision: " + str(round(100*precision.mean(), 2)) + "%"

recall =
cross_validation.cross_val_score(classifier_gaussiannb,
        X, y, scoring='recall_weighted', cv=num_validations)
print "Recall: " + str(round(100*recall.mean(), 2)) + "%"
```

Visualizing the confusion matrix

A confusion matrix is a table that we use to understand the performance of a classification model. This helps us understand how we classify testing data into different classes. When we want to fine-tune our algorithms, we need to understand how the data gets misclassified before we make these changes. Some classes are worse than others, and the confusion matrix will help us understand this. Let's look at the following figure:

	Predicted class 0	Predicted class 1	Predicted class 2
True class 0	45	4	3
True class 1	11	56	2
True class 2	5	6	49

In the preceding chart, we can see how we categorize data into different classes. Ideally, we want all the nondiagonal elements to be 0. This would indicate perfect classification! Let's consider **class 0**. Overall, 52 items actually belong to **class 0**. We get 52 if we sum up the numbers in the first row. Now, 45 of these items are being predicted correctly, but our classifier says that four of them belong to **class 1** and three of them belong to **class 2**. We can apply the same analysis to the remaining two rows as well. An interesting thing to note is that 11 items from **class 1** are misclassified as **class 0**. This constitutes around 16% of the datapoints in this class. This is an insight that we can use to optimize our model.

How to do it…

1. We will use the `confusion_matrix.py` file that we already provided to you as a reference. Let's see how to extract the confusion matrix from our data:

```
from sklearn.metrics import confusion_matrix
y_true = [1, 0, 0, 2, 1, 0, 3, 3, 3]
y_pred = [1, 1, 0, 2, 1, 0, 1, 3, 3]
```

```
confusion_mat = confusion_matrix(y_true, y_pred)
plot_confusion_matrix(confusion_mat)
```

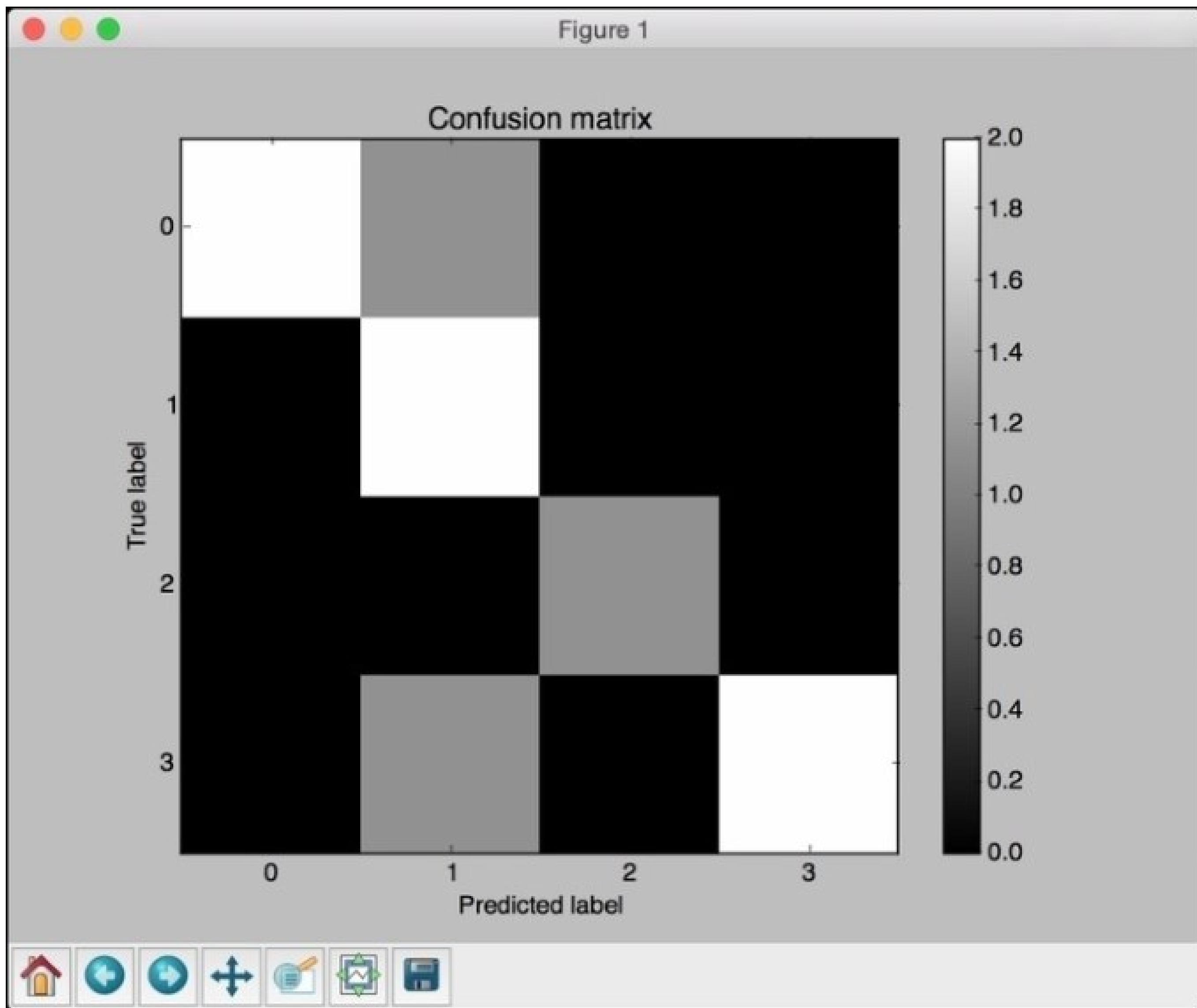
We use some sample data here. We have four classes with values ranging from 0 to 3. We have predicted labels as well. We use the `confusion_matrix` method to extract the confusion matrix and plot it.

2. Let's go ahead and define this function:

```
# Show confusion matrix
def plot_confusion_matrix(confusion_mat):
    plt.imshow(confusion_mat, interpolation='nearest',
cmap=plt.cm.Paired)
    plt.title('Confusion matrix')
    plt.colorbar()
    tick_marks = np.arange(4)
    plt.xticks(tick_marks, tick_marks)
    plt.yticks(tick_marks, tick_marks)
    plt.ylabel('True label')
    plt.xlabel('Predicted label')
    plt.show()
```

We use the `imshow` function to plot the confusion matrix. Everything else in the function is straightforward! We just set the title, color bar, ticks, and the labels using the relevant functions. The `tick_marks` argument range from 0 to 3 because we have four distinct labels in our dataset. The `np.arange` function gives us this `numpy` array.

3. If you run the preceding code, you will see the following figure:



The diagonal colors are strong, and we want them to be strong. The black color indicates zero. There are a couple of gray colors in the nondiagonal spaces, which indicate misclassification. For example, when the real label is 0, the predicted label is 1, as we can see in the first row. In fact, all the misclassifications belong to **class-1** in the sense that the second column contains three rows that are non-zero. It's easy to see this from the figure.

Extracting the performance report

We also have a function in scikit-learn that can directly print the precision, recall, and F1 scores for us. Let's see how to do this.

How to do it…

1. Add the following lines to a new Python file:

```
from sklearn.metrics import classification_report
y_true = [1, 0, 0, 2, 1, 0, 3, 3, 3]
y_pred = [1, 1, 0, 2, 1, 0, 1, 3, 3]
target_names = ['Class-0', 'Class-1', 'Class-2', 'Class-3']
print(classification_report(y_true, y_pred,
target_names=target_names))
```

2. If you run this code, you will see the following on your Terminal:

```
             precision    recall  f1-score   support

    Class-0       1.00      0.67      0.80         3
    Class-1       0.50      1.00      0.67         2
    Class-2       1.00      1.00      1.00         1
    Class-3       1.00      0.67      0.80         3

avg / total       0.89      0.78      0.79         9
```

Instead of computing these metrics separately, you can directly use this function to extract those statistics from your model.

Evaluating cars based on their characteristics

Let's see how we can apply classification techniques to a real-world problem. We will use a dataset that contains some details about cars, such as number of doors, boot space, maintenance costs, and so on. Our goal is to determine the quality of the car. For the purposes of classification, the quality can take four values: unacceptable, acceptable, good, and very good.

Getting ready

You can download the dataset at https://archive.ics.uci.edu/ml/datasets/Car+Evaluation.

You need to treat each value in the dataset as a string. We consider six attributes in the dataset. Here are the attributes along with the possible values they can take:

- `buying`: These will be `vhigh`, `high`, `med`, and `low`
- `maint`: These will be `vhigh`, `high`, `med`, and `low`
- `doors`: These will be `2`, `3`, `4`, `5`, and more
- `persons`: These will be `2`, `4`, more
- `lug_boot`: These will be `small`, `med`, and `big`
- `safety`: These will be `low`, `med`, and `high`

Given that each line contains strings, we need to assume that all the features are strings and design a classifier. In the previous chapter, we used random forests to build a regressor. In this recipe, we will use random forests as a classifier.

How to do it…

1. We will use the `car.py` file that we already provided to you as reference. Let's go ahead and import a couple of packages:

```
from sklearn import preprocessing
from sklearn.ensemble import RandomForestClassifier
```

2. Let's load the dataset:

```
input_file = 'path/to/dataset/car.data.txt'

# Reading the data
X = []
count = 0
with open(input_file, 'r') as f:
    for line in f.readlines():
        data = line[:-1].split(',')
        X.append(data)

X = np.array(X)
```

Each line contains a comma-separated list of words. Therefore, we parse the input file, split each line, and then append the list to the main data. We ignore the last character on each line because it's a newline character. The Python packages only work with numerical data, so we need to transform these attributes into something that those packages will understand.

3. In the previous chapter, we discussed label encoding. That is what we will use here to convert strings to numbers:

```
# Convert string data to numerical data
label_encoder = []
X_encoded = np.empty(X.shape)
for i,item in enumerate(X[0]):
    label_encoder.append(preprocessing.LabelEncoder())
    X_encoded[:, i] = label_encoder[-1].fit_transform(X[:, i])

X = X_encoded[:, :-1].astype(int)
y = X_encoded[:, -1].astype(int)
```

As each attribute can take a limited number of values, we can use the label encoder to transform them into numbers. We need to use different label encoders for each attribute. For example, the `lug_boot` attribute can take three distinct values, and we need a label encoder that knows how to encode this attribute. The last value on each line is the class, so we assign it to the `y` variable.

4. Let's train the classifier:

```
# Build a Random Forest classifier
params = {'n_estimators': 200, 'max_depth': 8, 'random_state':
7}
classifier = RandomForestClassifier(**params)
classifier.fit(X, y)
```

You can play around with the `n_estimators` and `max_depth` parameters to see how they affect the classification accuracy. We will actually do this soon in a standardized way.

5. Let's perform cross-validation:

```
# Cross validation
from sklearn import cross_validation

accuracy = cross_validation.cross_val_score(classifier,
        X, y, scoring='accuracy', cv=3)
print "Accuracy of the classifier: " +
str(round(100*accuracy.mean(), 2)) + "%"
```

Once we train the classifier, we need to see how it performs. We use three-fold cross-validation to calculate the accuracy here.

6. One of the main goals of building a classifier is to use it on isolated and unknown data instances. Let's use a single datapoint and see how we can use this classifier to categorize it:

```
# Testing encoding on single data instance
input_data = ['vhigh', 'vhigh', '2', '2', 'small', 'low']
input_data_encoded = [-1] * len(input_data)
for i,item in enumerate(input_data):
    input_data_encoded[i] =
int(label_encoder[i].transform(input_data[i]))

input_data_encoded = np.array(input_data_encoded)
```

The first step was to convert that data into numerical data. We need to use the label encoders that we used during training because we want it to be consistent. If there are unknown values in the input datapoint, the label encoder will complain because it doesn't know how to handle that data. For example, if you change the first value in the list from `vhigh` to `abcd`, then the label encoder won't work because it doesn't know how to interpret this string. This acts like an error check to see if the input datapoint is valid.

7. We are now ready to predict the output class for this datapoint:

```
# Predict and print output for a particular datapoint
output_class = classifier.predict(input_data_encoded)
print "Output class:",
label_encoder[-1].inverse_transform(output_class)[0]
```

We use the `predict` method to estimate the output class. If we output the encoded output label, it wouldn't mean anything to us. Therefore, we use the `inverse_transform` method to convert this label back to its original form and print out the output class.

Extracting validation curves

We used random forests to build a classifier in the previous recipe, but we don't exactly know how to define the parameters. In our case, we dealt with two parameters: `n_estimators` and `max_depth`. They are called **hyperparameters**, and the performance of the classifier depends on them. It would be nice to see how the performance gets affected as we change the hyperparameters. This is where validation curves come into picture. These curves help us understand how each hyperparameter influences the training score. Basically, all other parameters are kept constant and we vary the hyperparameter of interest according to our range. We will then be able to visualize how this affects the score.

How to do it…

1. Add the following code to the same Python file, as in the previous recipe:

   ```
   # Validation curves

   from sklearn.learning_curve import validation_curve

   classifier = RandomForestClassifier(max_depth=4, random_state=7)
   parameter_grid = np.linspace(25, 200, 8).astype(int)
   train_scores, validation_scores = validation_curve(classifier,
   X, y,
           "n_estimators", parameter_grid, cv=5)
   print "\n##### VALIDATION CURVES #####"
   print "\nParam: n_estimators\nTraining scores:\n", train_scores
   print "\nParam: n_estimators\nValidation scores:\n",
   validation_scores
   ```

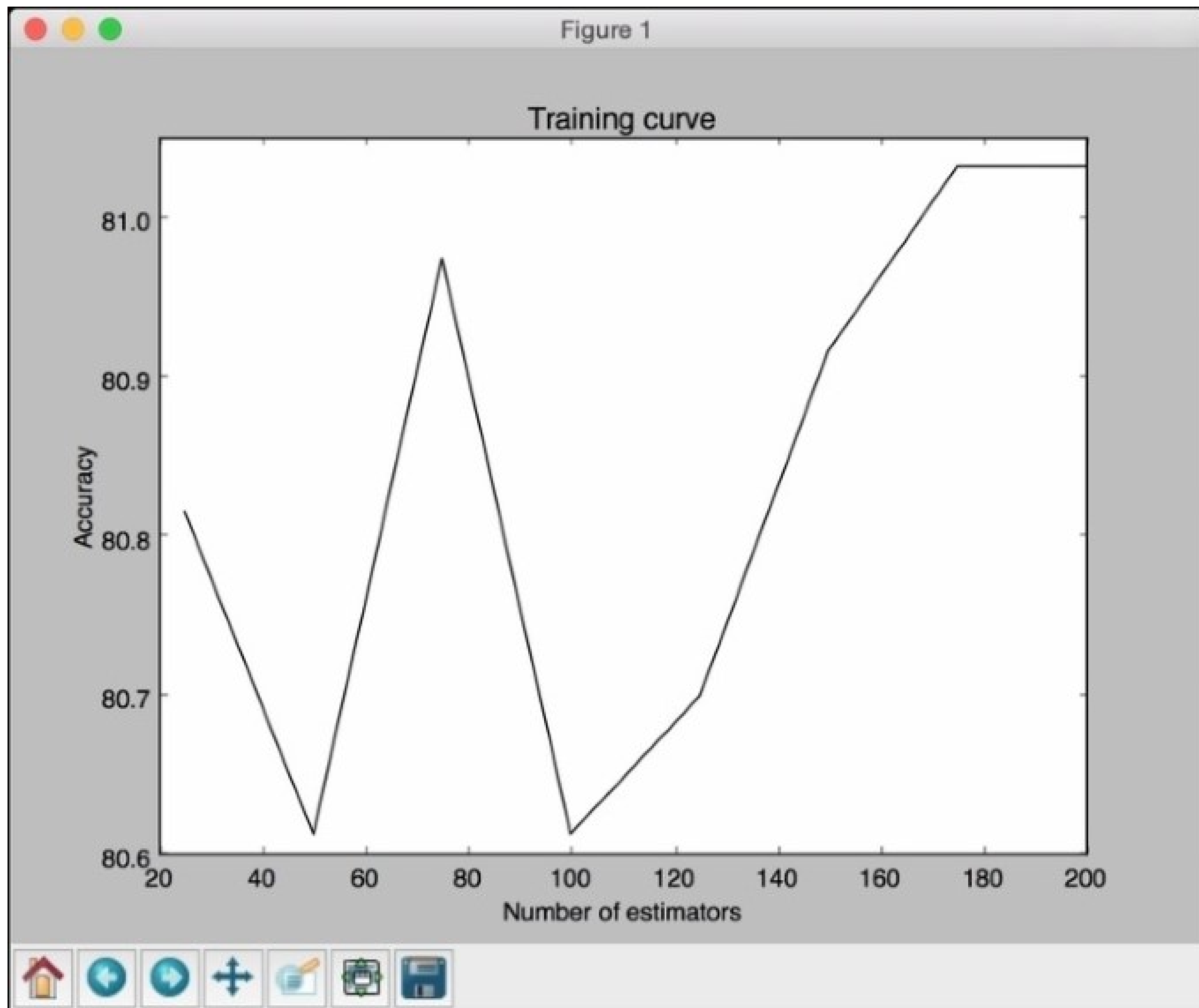
   In this case, we defined the classifier by fixing the `max_depth` parameter. We want to estimate the optimal number of estimators to use, and so have defined our search space using `parameter_grid`. It is going to extract training and validation scores by iterating from 25 to 200 in eight steps.
2. If you run it, you will see the following on your Terminal:

```
##### VALIDATION CURVES #####

Param: n_estimators
Training scores:
[[ 0.80680174  0.80824891  0.80752533  0.80463097  0.81358382]
 [ 0.79522431  0.80535456  0.81041968  0.8089725   0.81069364]
 [ 0.80101302  0.80680174  0.81114327  0.81476122  0.8150289 ]
 [ 0.8024602   0.80535456  0.81186686  0.80752533  0.80346821]
 [ 0.80028944  0.80463097  0.81114327  0.80824891  0.81069364]
 [ 0.80390738  0.80535456  0.81041968  0.80969609  0.81647399]
 [ 0.80390738  0.80463097  0.81114327  0.81476122  0.81719653]
 [ 0.80390738  0.80607815  0.81114327  0.81403763  0.81647399]]

Param: n_estimators
Validation scores:
[[ 0.71098266  0.76589595  0.72543353  0.76300578  0.75290698]
 [ 0.71098266  0.75433526  0.71965318  0.75722543  0.74127907]
 [ 0.71098266  0.72254335  0.71965318  0.75722543  0.74418605]
 [ 0.71098266  0.71387283  0.71965318  0.75722543  0.72674419]
 [ 0.71098266  0.74277457  0.71965318  0.75722543  0.74127907]
 [ 0.71098266  0.74277457  0.71965318  0.75722543  0.74127907]
 [ 0.71098266  0.74566474  0.71965318  0.75722543  0.74418605]
 [ 0.71098266  0.75144509  0.71965318  0.75722543  0.74127907]]
```

3. Let's plot it:

```
# Plot the curve
plt.figure()
plt.plot(parameter_grid, 100*np.average(train_scores, axis=1),
color='black')
plt.title('Training curve')
plt.xlabel('Number of estimators')
plt.ylabel('Accuracy')
plt.show()
```

4. Here is the figure that you'll get:



5. Let's do the same for the `max_depth` parameter:

```
classifier = RandomForestClassifier(n_estimators=20,
random_state=7)
parameter_grid = np.linspace(2, 10, 5).astype(int)
train_scores, valid_scores = validation_curve(classifier, X, y,
        "max_depth", parameter_grid, cv=5)
print "\nParam: max_depth\nTraining scores:\n", train_scores
print "\nParam: max_depth\nValidation scores:\n",
validation_scores
```

We fixed the `n_estimators` parameter at 20 to see how the performance varies with `max_depth`. Here is the output on the Terminal:

```
Param: max_depth
Training scores:
[[ 0.71852388  0.70043415  0.70043415  0.70043415  0.69942197]
 [ 0.80607815  0.80535456  0.80752533  0.79450072  0.81069364]
 [ 0.90665702  0.91027496  0.92836469  0.89797395  0.90679191]
 [ 0.97467438  0.96743849  0.97105644  0.97829233  0.96820809]
 [ 0.99421129  0.99782923  0.99782923  0.99855282  0.99421965]]

Param: max_depth
Validation scores:
[[ 0.71098266  0.76589595  0.72543353  0.76300578  0.75290698]
 [ 0.71098266  0.75433526  0.71965318  0.75722543  0.74127907]
 [ 0.71098266  0.72254335  0.71965318  0.75722543  0.74418605]
 [ 0.71098266  0.71387283  0.71965318  0.75722543  0.72674419]
 [ 0.71098266  0.74277457  0.71965318  0.75722543  0.74127907]
 [ 0.71098266  0.74277457  0.71965318  0.75722543  0.74127907]
 [ 0.71098266  0.74566474  0.71965318  0.75722543  0.74418605]
 [ 0.71098266  0.75144509  0.71965318  0.75722543  0.74127907]]
```

6. Let's plot it:

```
# Plot the curve
plt.figure()
plt.plot(parameter_grid, 100*np.average(train_scores, axis=1),
color='black')
plt.title('Validation curve')
plt.xlabel('Maximum depth of the tree')
plt.ylabel('Accuracy')
plt.show()
```

7. If you run this code, you will get the following figure:

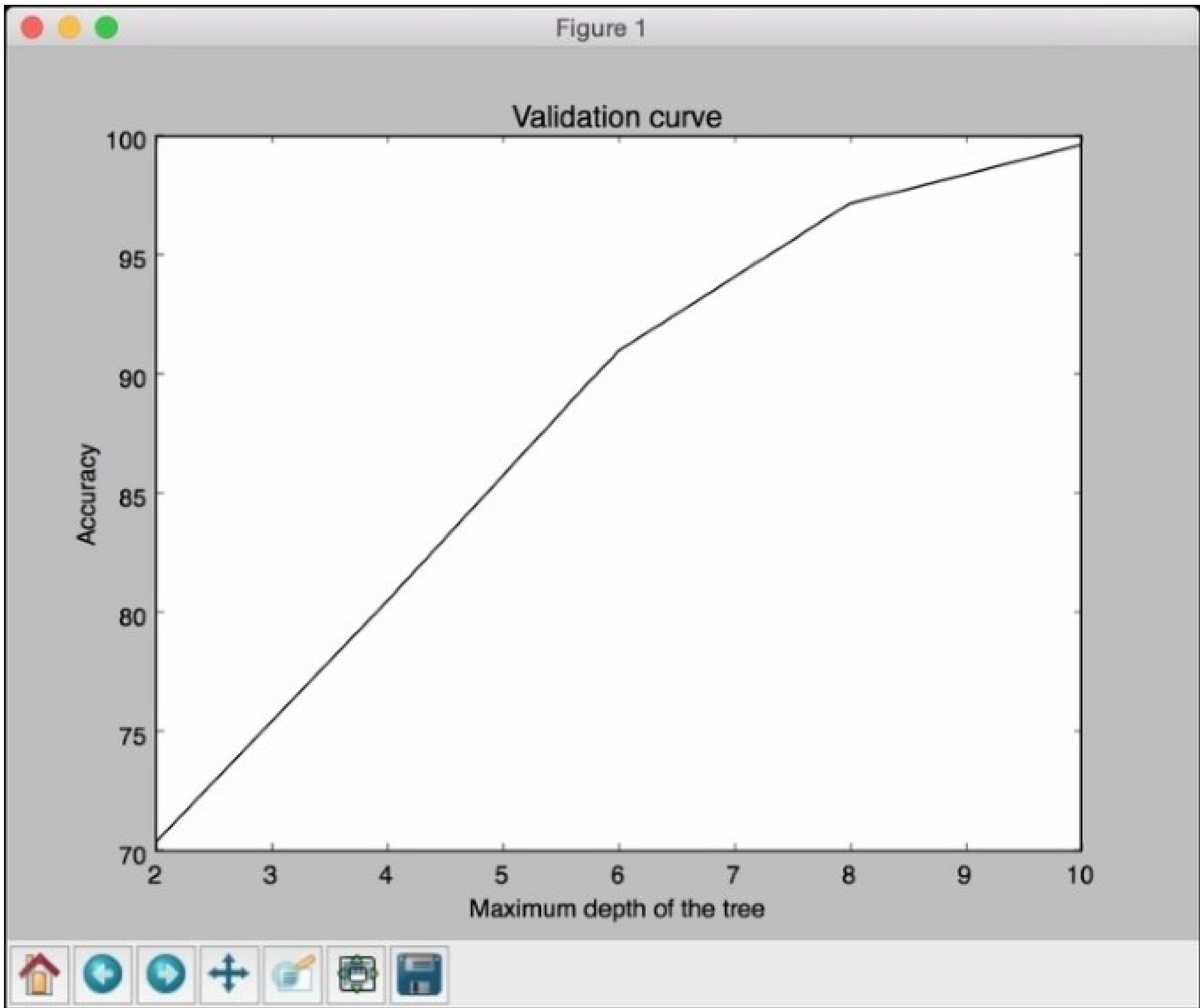


Figure 1
Validation curve
100
95
90
85
80
75
70
Accuracy
2
3
4
5
6
7
8
9
10
Maximum depth of the tree

Extracting learning curves

Learning curves help us understand how the size of our training dataset influences the machine learning model. This is very useful when you have to deal with computational constraints. Let's go ahead and plot the learning curves by varying the size of our training dataset.

How to do it…

1. Add the following code to the same Python file, as in the previous recipe:

```
# Learning curves

from sklearn.learning_curve import learning_curve

classifier = RandomForestClassifier(random_state=7)

parameter_grid = np.array([200, 500, 800, 1100])
train_sizes, train_scores, validation_scores =
learning_curve(classifier,
        X, y, train_sizes=parameter_grid, cv=5)
print "\n##### LEARNING CURVES #####"
print "\nTraining scores:\n", train_scores
print "\nValidation scores:\n", validation_scores
```

   We want to evaluate the performance metrics using training datasets of size 200, 500, 800, and 1100. We use five-fold cross-validation, as specified by the `cv` parameter in the `learning_curve` method.

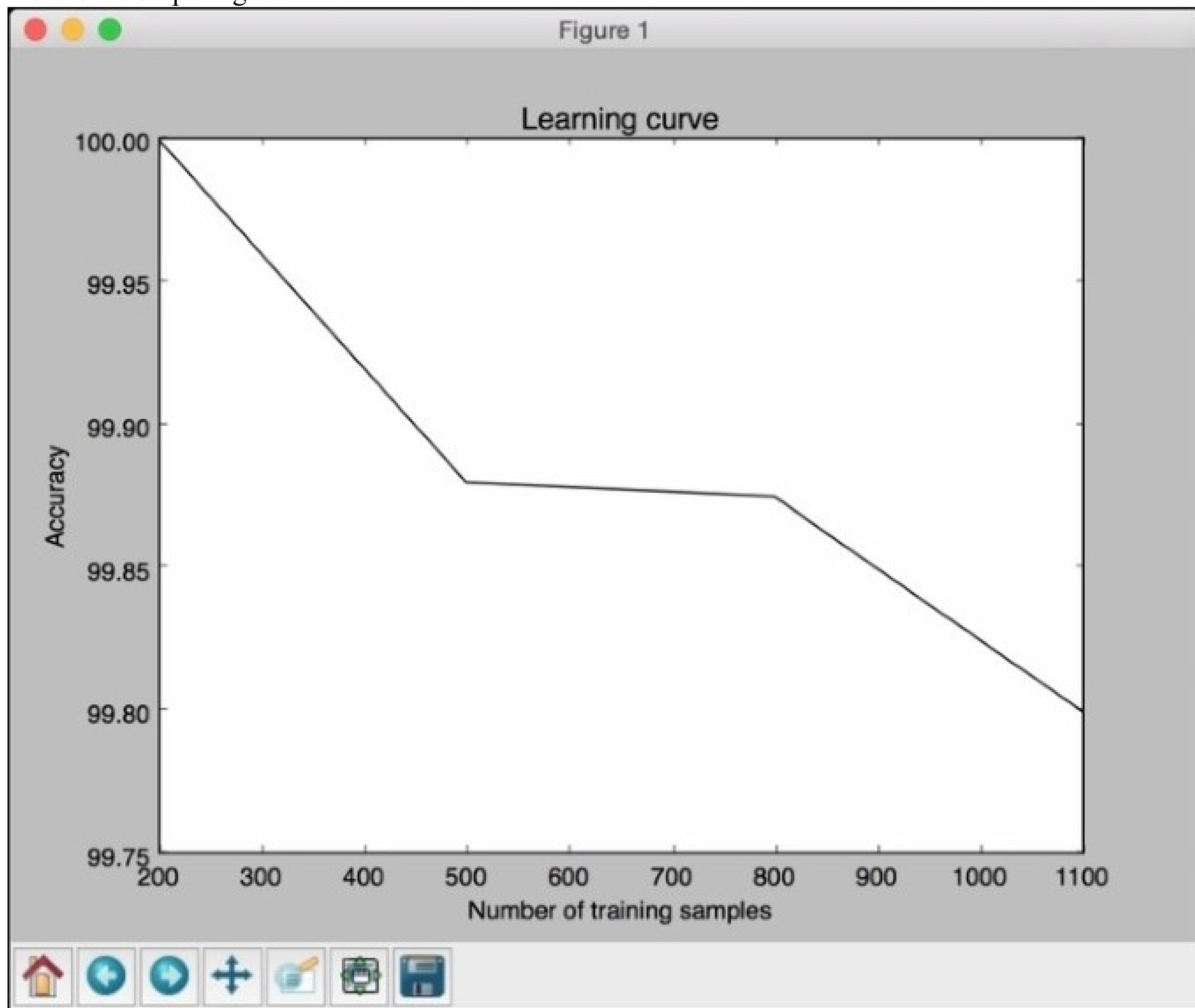
2. If you run this code, you will get the following output on the Terminal:

```
##### LEARNING CURVES #####

Training scores:
[[ 1.          1.          1.          1.          1.        ]
 [ 1.          1.          0.998       0.998       0.998     ]
 [ 0.99875     0.99875     0.99875     0.99875     0.99875   ]
 [ 0.99909091  0.99545455  0.99909091  0.99818182  0.99818182]]

Validation scores:
[[ 0.69942197  0.69942197  0.69942197  0.69942197  0.70348837]
 [ 0.75433526  0.65028902  0.76878613  0.76589595  0.70348837]
 [ 0.70520231  0.78612717  0.52312139  0.76878613  0.77034884]
 [ 0.6416185   0.75722543  0.64450867  0.75433526  0.76744186]]
```

3. Let's plot it:

```
# Plot the curve
plt.figure()
plt.plot(parameter_grid, 100*np.average(train_scores, axis=1),
color='black')
plt.title('Learning curve')
plt.xlabel('Number of training samples')
plt.ylabel('Accuracy')
plt.show()
```

4. Here is the output figure:



Although smaller training sets seem to give better accuracy, they are prone to overfitting. If we choose a bigger training dataset, it consumes more resources. Therefore, we need to make a trade-off here to pick the right size of the training dataset.

Estimating the income bracket

We will build a classifier to estimate the income bracket of a person based on 14 attributes. The possible output classes are *higher than 50K* or *lower than or equal to 50K*. There is a slight twist in this dataset in the sense that each datapoint is a mixture of numbers and strings. Numerical data is valuable, and we cannot use a label encoder in these situations. We need to design a system that can deal with numerical and non-numerical data at the same time. We will use the census income dataset available at https://archive.ics.uci.edu/ml/datasets/Census+Income.

How to do it…

1. We will use the `income.py` file already provided to you as a reference. We will use a Naive Bayes classifier to achieve this. Let's import a couple of packages:

```
from sklearn import preprocessing
from sklearn.naive_bayes import GaussianNB
```

2. Let's load the dataset:

```
input_file = 'path/to/adult.data.txt'

# Reading the data
X = []
y = []
count_lessthan50k = 0
count_morethan50k = 0
num_images_threshold = 10000
```

3. We will use 20,000 datapoints from the datasets—10,000 for each class to avoid class imbalance. During training, if you use many datapoints that belong to a single class, the classifier tends to get biased toward this class. Therefore, it's better to use the same number of datapoints for each class:

```
with open(input_file, 'r') as f:
    for line in f.readlines():
        if '?' in line:
            continue

        data = line[:-1].split(', ')

        if data[-1] == '<=50K' and count_lessthan50k <
num_images_threshold:
            X.append(data)
            count_lessthan50k = count_lessthan50k + 1

        elif data[-1] == '>50K' and count_morethan50k <
num_images_threshold:
            X.append(data)
```

```
                count_morethan50k = count_morethan50k + 1

        if count_lessthan50k >= num_images_threshold and
count_morethan50k >= num_images_threshold:
            break

X = np.array(X)
```

It's a comma-separated file again. We just loaded the data in the `X` variable just like before.

4. We need to convert string attributes to numerical data while leaving out the original numerical data:

```
# Convert string data to numerical data
label_encoder = []
X_encoded = np.empty(X.shape)
for i,item in enumerate(X[0]):
    if item.isdigit():
        X_encoded[:, i] = X[:, i]
    else:
        label_encoder.append(preprocessing.LabelEncoder())
        X_encoded[:, i] = label_encoder[-1].fit_transform(X[:,
i])

X = X_encoded[:, :-1].astype(int)
y = X_encoded[:, -1].astype(int)
```

The `isdigit()` function helps us in identifying numerical data. We converted string data to numerical data and stored all the label encoders in a list so that we can use it when we want to classify unknown data.

5. Let's train the classifier:

```
# Build a classifier
classifier_gaussiannb = GaussianNB()
classifier_gaussiannb.fit(X, y)
```

6. Let's split the data into training and testing to extract performance metrics:

```
# Cross validation
from sklearn import cross_validation

X_train, X_test, y_train, y_test =
cross_validation.train_test_split(X, y, test_size=0.25,
random_state=5)
classifier_gaussiannb = GaussianNB()
classifier_gaussiannb.fit(X_train, y_train)
y_test_pred = classifier_gaussiannb.predict(X_test)
```

7. Let's extract performance metrics:

```
# compute F1 score of the classifier
f1 = cross_validation.cross_val_score(classifier_gaussiannb,
        X, y, scoring='f1_weighted', cv=5)
print "F1 score: " + str(round(100*f1.mean(), 2)) + "%"
```

8. Let's see how to classify a single datapoint. We need to convert the datapoint into something that our classifier can understand:

```
# Testing encoding on single data instance
input_data = ['39', 'State-gov', '77516', 'Bachelors', '13',
'Never-married', 'Adm-clerical', 'Not-in-family', 'White',
'Male', '2174', '0', '40', 'United-States']
count = 0
input_data_encoded = [-1] * len(input_data)
for i,item in enumerate(input_data):
    if item.isdigit():
        input_data_encoded[i] = int(input_data[i])
    else:
        input_data_encoded[i] =
int(label_encoder[count].transform(input_data[i]))
        count = count + 1

input_data_encoded = np.array(input_data_encoded)
```

9. We are now ready to classify it:

```
# Predict and print output for a particular datapoint
output_class = classifier_gaussiannb.predict(input_data_encoded)
print label_encoder[-1].inverse_transform(output_class)[0]
```

Just like before, we use the `predict` method to get the output class and the `inverse_transform` method to convert this label back to its original form to print it out on the Terminal.

Chapter 3. Predictive Modeling

In this chapter, we will cover the following recipes:

- Building a linear classifier using Support Vector Machines (SVMs)
- Building a nonlinear classifier using SVMs
- Tackling class imbalance
- Extracting confidence measurements
- Finding optimal hyperparameters
- Building an event predictor
- Estimating traffic

Introduction

Predictive modeling is probably one of the most exciting fields in data analytics. It has gained a lot of attention in recent years due to massive amounts of data being available in many different verticals. It is very commonly used in areas concerning data mining to forecast future trends.

Predictive modeling is an analysis technique that is used to predict the future behavior of a system. It is a collection of algorithms that can identify the relationship between independent input variables and the target responses. We create a mathematical model, based on observations, and then use this model to estimate what's going to happen in the future.

In predictive modeling, we need to collect data with known responses to train our model. Once we create this model, we validate it using some metrics, and then use it to predict future values. We can use many different types of algorithms to create a predictive model. In this chapter, we will use Support Vector Machines to build linear and nonlinear models.

A predictive model is built using a number of features that are likely to influence the behavior of the system. For example, to estimate the weather conditions, we may use various types of data, such as temperature, barometric pressure, precipitation, and other atmospheric processes. Similarly, when we deal with other types of systems, we need to decide what factors are likely to influence its behavior and include them as part of the feature vector before training our model.

Building a linear classifier using Support Vector Machine (SVMs)

SVMs are supervised learning models that are used to build classifiers and regressors. An SVM finds the best separating boundary between the two sets of points by solving a system of mathematical equations. If you are not familiar with SVMs, here are a couple of good tutorials to get started:

- http://web.mit.edu/zoya/www/SVM.pdf
- http://www.support-vector.net/icml-tutorial.pdf
- http://www.svms.org/tutorials/Berwick2003.pdf

Let's see how to build a linear classifier using an SVM.

Getting ready

Let's visualize our data to understand the problem at hand. We will use `svm.py` that's already provided to you as a reference. Before we build the SVM, let's understand our data. We will use the `data_multivar.txt` file that's already provided to you. Let's see how to to visualize the data. Create a new Python file and add the following lines to it:

```
import numpy as np
import matplotlib.pyplot as plt

import utilities

# Load input data
input_file = 'data_multivar.txt'
X, y = utilities.load_data(input_file)
```

We just imported a couple of packages and named the input file. Let's look at the `load_data()` method:

```
# Load multivar data in the input file
def load_data(input_file):
    X = []
    y = []
    with open(input_file, 'r') as f:
        for line in f.readlines():
            data = [float(x) for x in line.split(',')]
            X.append(data[:-1])
            y.append(data[-1])

    X = np.array(X)
    y = np.array(y)

    return X, y
```

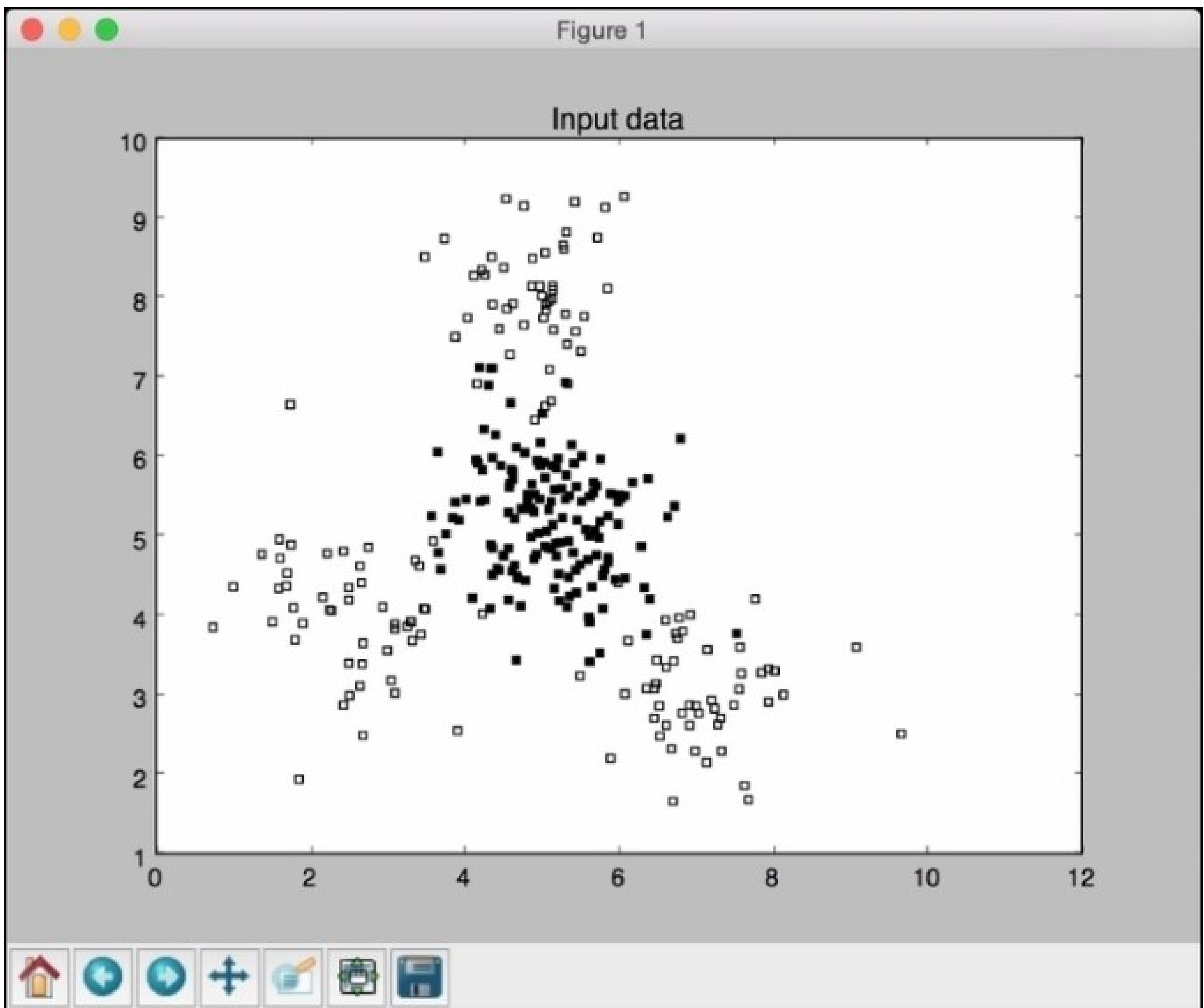
We need to separate the data into classes, as follows:

```
class_0 = np.array([X[i] for i in range(len(X)) if y[i]==0])
class_1 = np.array([X[i] for i in range(len(X)) if y[i]==1])
```

Now that we have separated the data, let's plot it:

```
plt.figure()
plt.scatter(class_0[:,0], class_0[:,1], facecolors='black',
edgecolors='black', marker='s')
plt.scatter(class_1[:,0], class_1[:,1], facecolors='None',
edgecolors='black', marker='s')
plt.title('Input data')
plt.show()
```

If you run this code, you will see the following figure:



The preceding figure consists of two types of points – **solid squares** and **empty squares**. In machine learning lingo, we say that our data consists of two classes. Our goal is to build a model that can separate the solid squares from the empty squares.

How to do it…

1. We need to split our dataset into training and testing datasets. Add the following lines to the same Python file:

```
# Train test split and SVM training
from sklearn import cross_validation
from sklearn.svm import SVC

X_train, X_test, y_train, y_test =
cross_validation.train_test_split(X, y, test_size=0.25,
random_state=5)
```

2. Let's initialize the SVM object using a linear kernel. If you don't know what kernels are, you can check out http://www.eric-kim.net/eric-kim-net/posts/1/kernel_trick.html. Add the following lines to the file:

```
params = {'kernel': 'linear'}
classifier = SVC(**params)
```

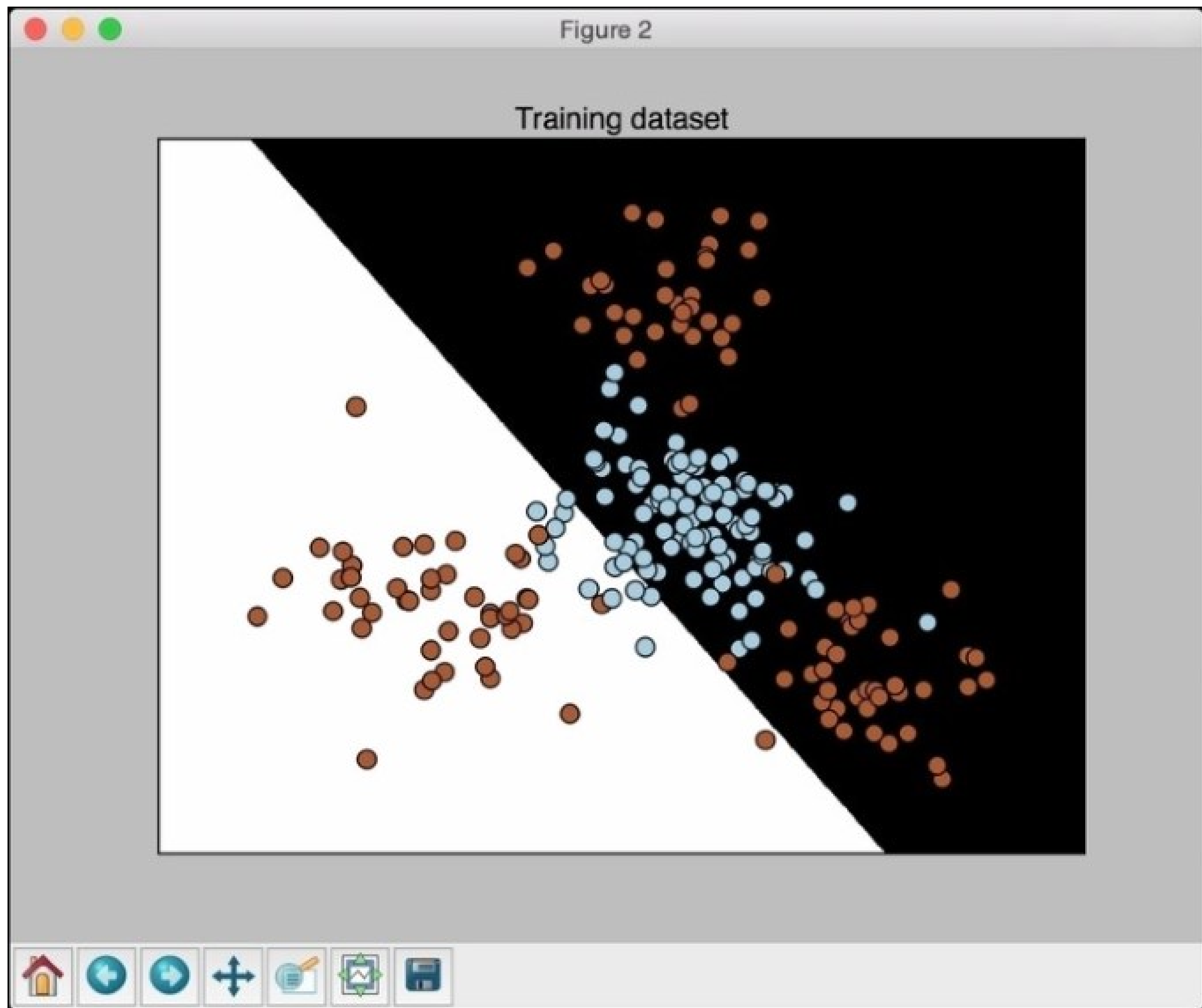
3. We are now ready to train the linear SVM classifier:

```
classifier.fit(X_train, y_train)
```

4. We can now see how the classifier performs:

```
utilities.plot_classifier(classifier, X_train, y_train,
'Training dataset')
plt.show()
```

5. If you run this code, you will get the following figure:

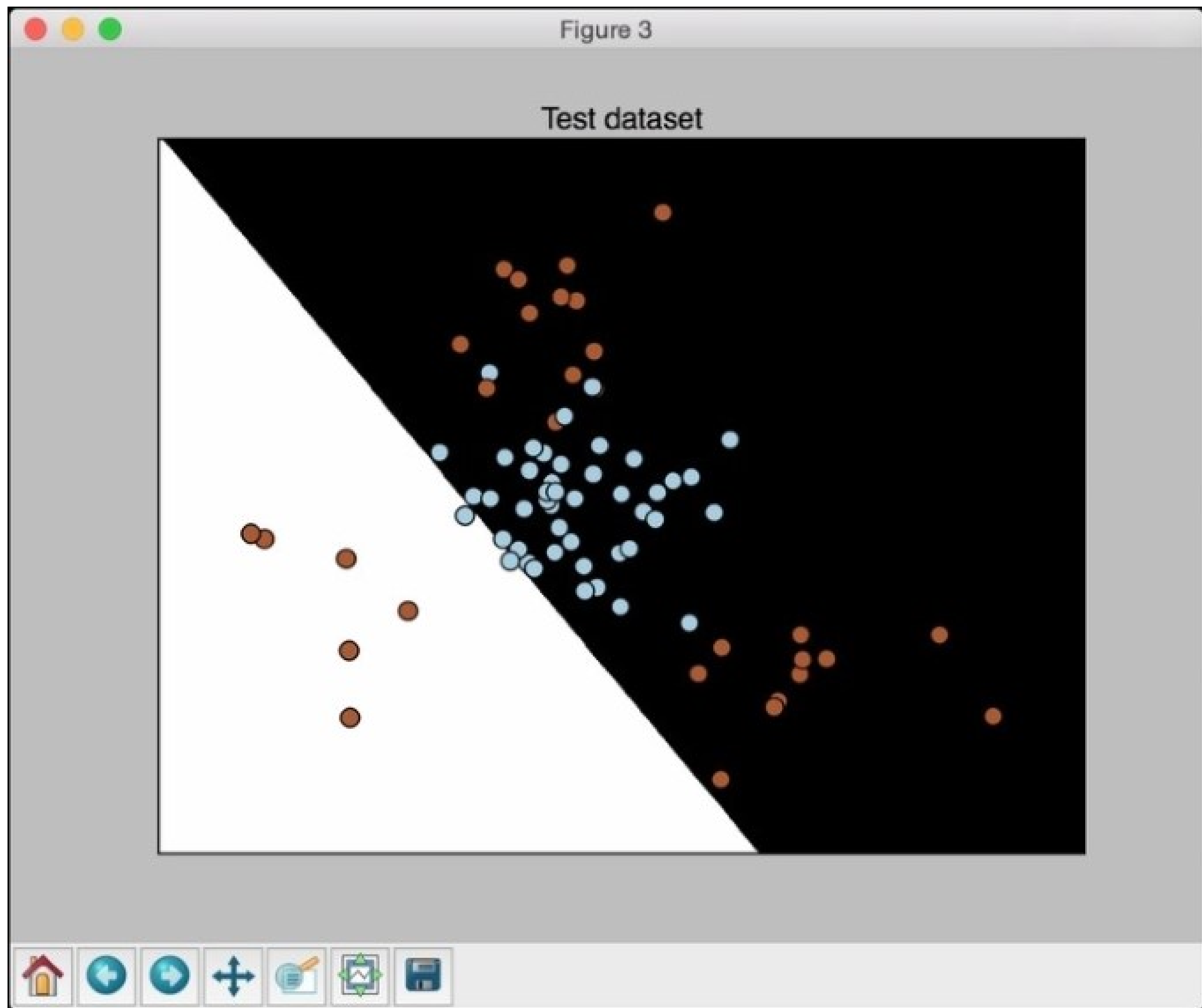


The `plot_classifier` function is the same that we discussed in the previous chapter. It has a couple of minor additions. You can check out the `utilities.py` file already provided to you for more details.

6. Let's see how this performs on the test dataset. Add the following lines to the same file:

```
y_test_pred = classifier.predict(X_test)
utilities.plot_classifier(classifier, X_test, y_test, 'Test
dataset')
plt.show()
```

7. If you run this code, you will see the following figure:



8. Let's compute the accuracy for the training set. Add the following lines to the same file:

```
from sklearn.metrics import classification_report

target_names = ['Class-' + str(int(i)) for i in set(y)]
print "\n" + "#"*30
print "\nClassifier performance on training dataset\n"
print classification_report(y_train,
classifier.predict(X_train), target_names=target_names)
print "#"*30 + "\n"
```

9. If you run this code, you will see the following on your Terminal:

```
##############################

Classifier performance on training dataset

             precision    recall  f1-score   support

    Class-0       0.55      0.88      0.68       105
    Class-1       0.78      0.38      0.51       120

avg / total       0.67      0.61      0.59       225

##############################
```

10. Finally, let's see the classification report for the testing dataset:

```
print "#"*30
print "\nClassification report on test dataset\n"
print classification_report(y_test, y_test_pred,
target_names=target_names)
print "#"*30 + "\n"
```

11. If you run this code, you will see the following on the Terminal:

```
##############################

Classification report on test dataset

             precision    recall  f1-score   support

    Class-0       0.64      0.96      0.77        45
    Class-1       0.75      0.20      0.32        30

avg / total       0.69      0.65      0.59        75

##############################
```

From the figure where we visualized the data, we can see that the solid squares are completely surrounded by empty squares. This means that the data is not linearly separable. We cannot draw a nice straight line to separate the two sets of points! Hence, we need a nonlinear classifier to separate these datapoints.

Building a nonlinear classifier using SVMs

An SVM provides a variety of options to build a nonlinear classifier. We need to build a nonlinear classifier using various kernels. For the sake of simplicity, let's consider two cases here. When we want to represent a curvy boundary between two sets of points, we can either do this using a polynomial function or a radial basis function.

How to do it…

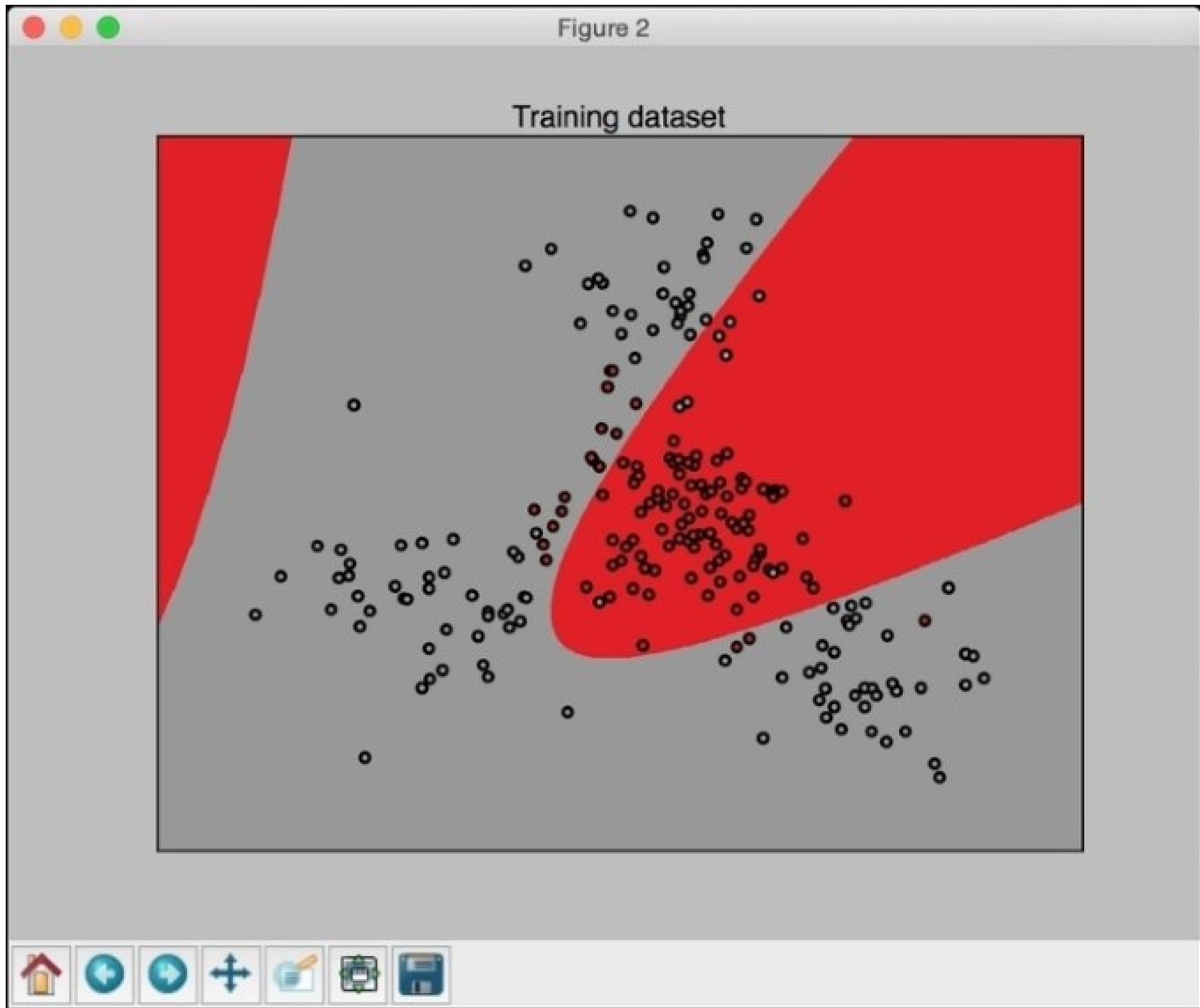
1. For the first case, let's use a polynomial kernel to build a nonlinear classifier. In the same Python file, search for the following line:

   ```
   params = {'kernel': 'linear'}
   ```

   Replace this line with the following:

   ```
   params = {'kernel': 'poly', 'degree': 3}
   ```

   This means that we use a polynomial function with degree 3. If you increase the degree, this means we allow the polynomial to be curvier. However, curviness comes at a cost in the sense that it will take more time to train because it's more computationally expensive.
2. If you run this code now, you will get the following figure:



3. You will also see the following classification report printed on your Terminal:

```
##############################

Classifier performance on training dataset

             precision    recall  f1-score   support

    Class-0       0.92      0.84      0.88       105
    Class-1       0.87      0.93      0.90       120

avg / total       0.89      0.89      0.89       225

##############################
```

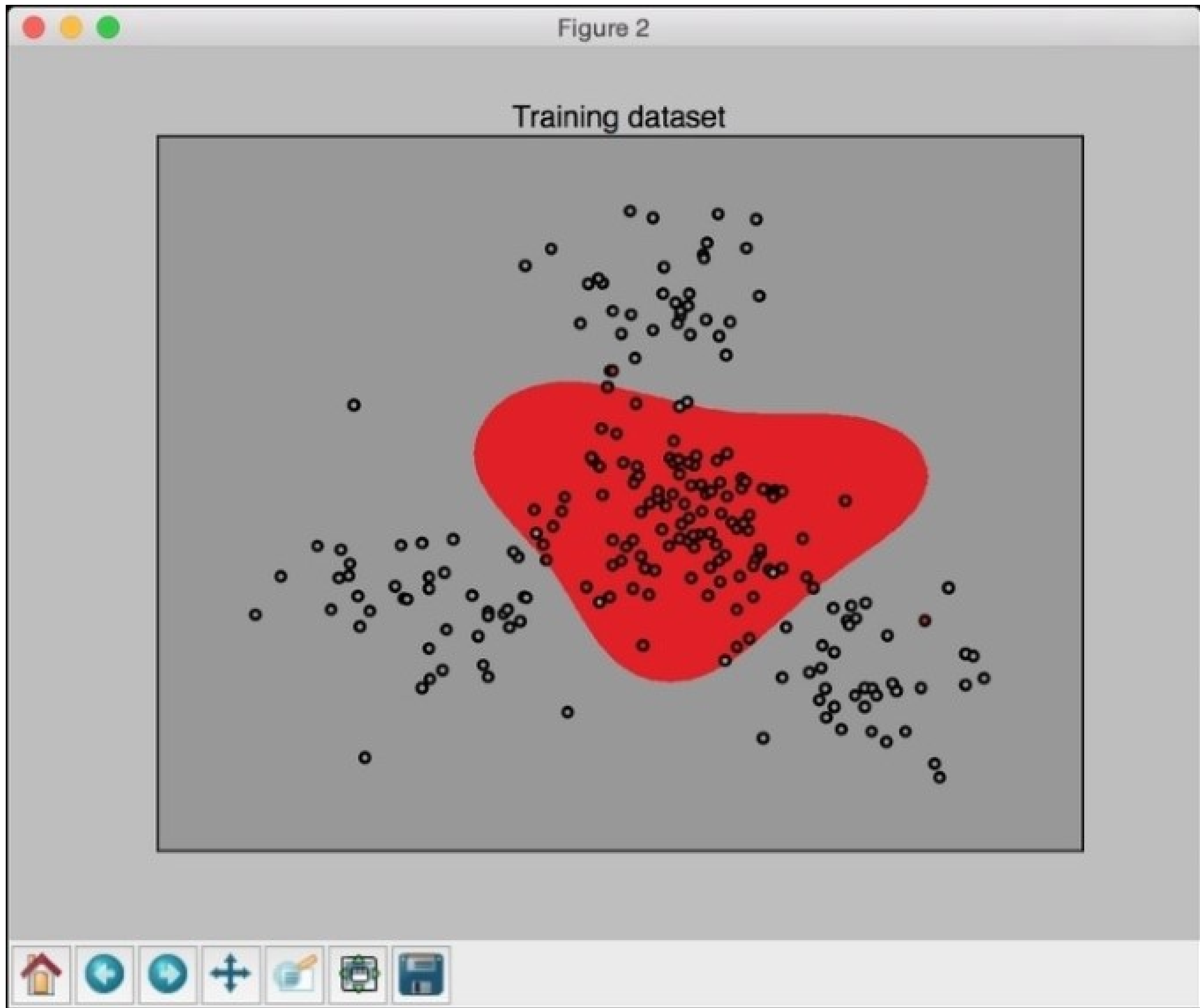
4. We can also use a radial basis function kernel to build a nonlinear classifier. In the same Python file, search for the following line:

   ```
   params = {'kernel': 'poly', 'degree': 3}
   ```

   Replace this line with the following one:

   ```
   params = {'kernel': 'rbf'}
   ```

5. If you run this code now, you will get the following figure:



6. You will also see the following classification report printed on your Terminal:

```
##############################

Classifier performance on training dataset

             precision    recall  f1-score   support

    Class-0       0.95      0.98      0.97       105
    Class-1       0.98      0.96      0.97       120

avg / total       0.97      0.97      0.97       225

##############################
```

Tackling class imbalance

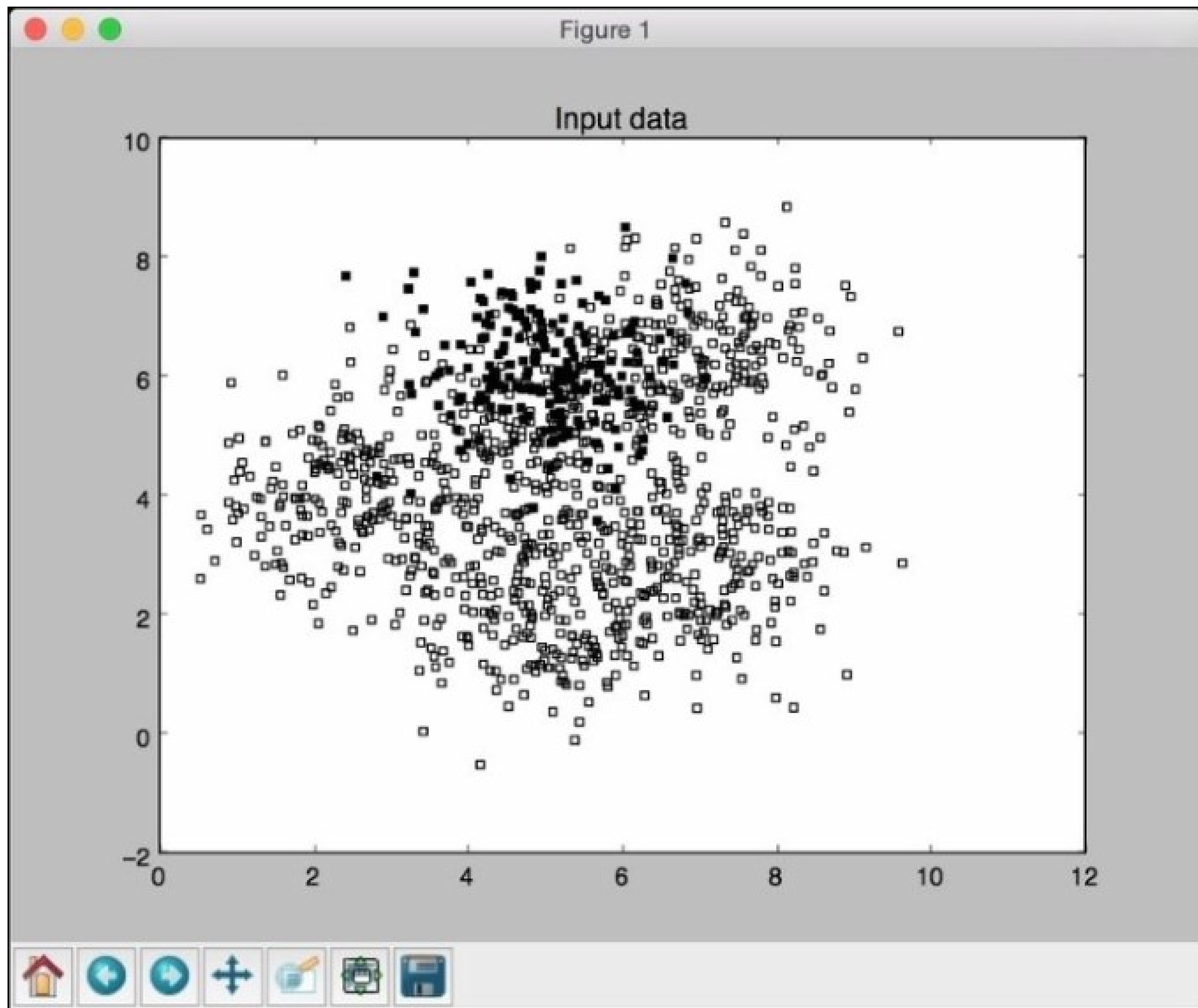
Until now, we dealt with problems where we had a similar number of datapoints in all our classes. In the real world, we might not be able to get data in such an orderly fashion. Sometimes, the number of datapoints in one class is a lot more than the number of datapoints in other classes. If this happens, then the classifier tends to get biased. The boundary won't reflect of the true nature of your data just because there is a big difference in the number of datapoints between the two classes. Therefore, it becomes important to account for this discrepancy and neutralize it so that our classifier remains impartial.

How to do it…

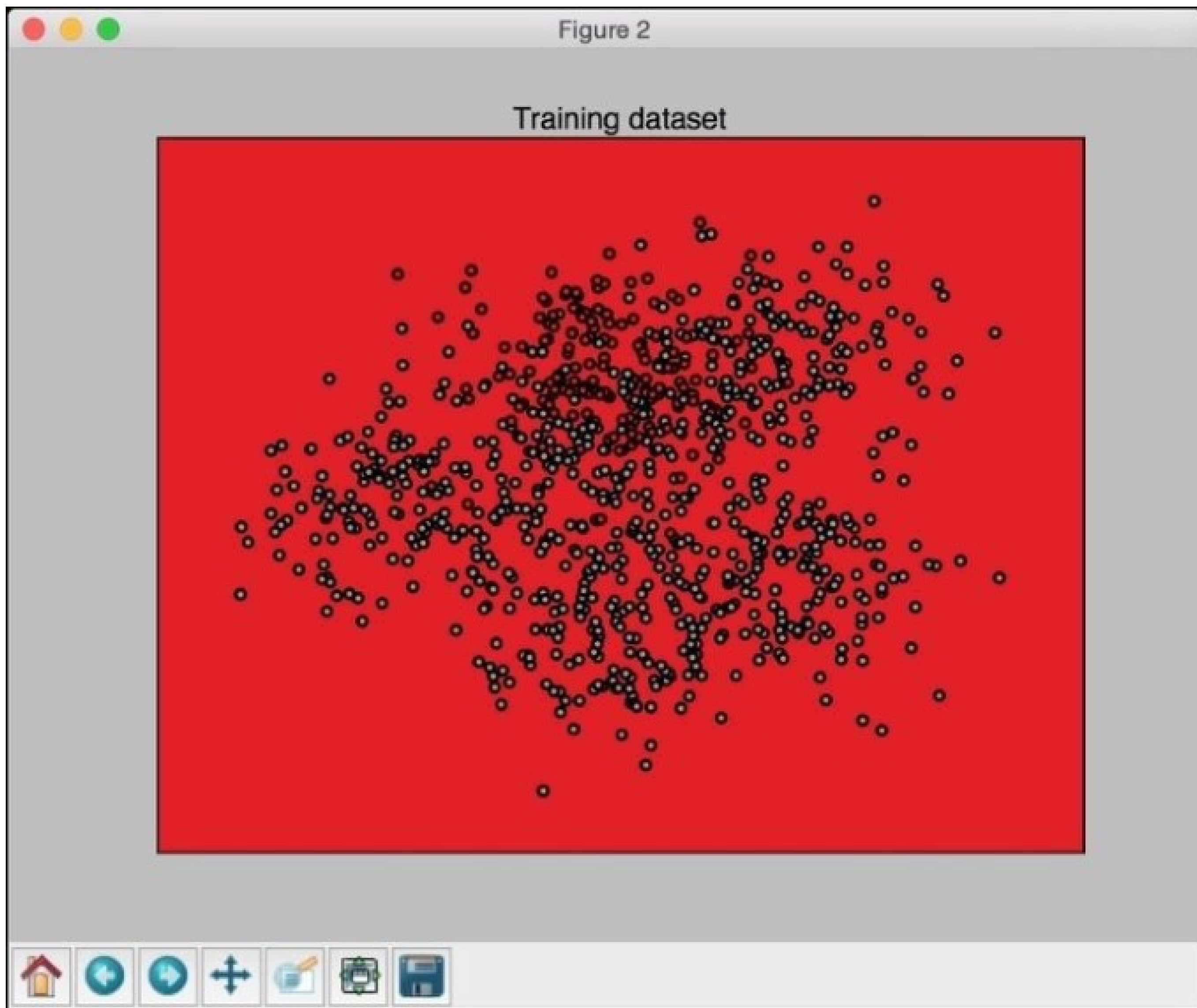
1. Let's load the data:

```
input_file = 'data_multivar_imbalance.txt'
X, y = utilities.load_data(input_file)
```

2. Let's visualize the data. The code for visualization is exactly the same as it was in the previous recipe. You can also find it in the file named `svm_imbalance.py` already provided to you. If you run it, you will see the following figure:



3. Let's build an SVM with a linear kernel. The code is the same as it was in the previous recipe. If you run it, you will see the following figure:



4. You might wonder why there's no boundary here! Well, this is because the classifier is unable to separate the two classes at all, resulting in 0% accuracy for `Class-0`. You will also see a classification report printed on your Terminal, as shown in the following screenshot:

```
##################################

Classification report on test dataset

              precision    recall  f1-score   support

    Class-0       0.00      0.00      0.00        42
    Class-1       0.86      1.00      0.92       258

avg / total       0.74      0.86      0.80       300

##################################
```

As we expected, `Class-0` has 0% precision.

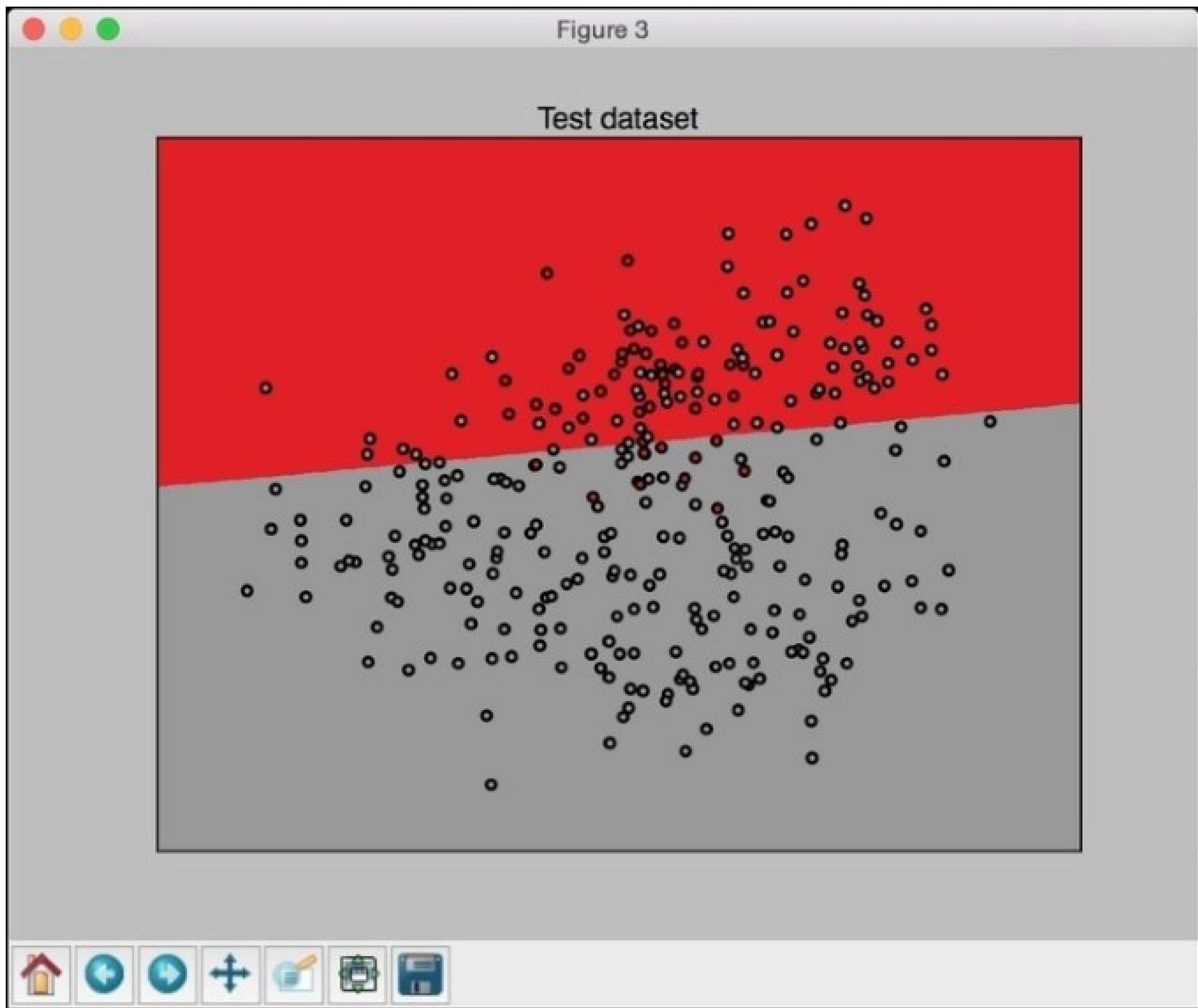
5. Let's go ahead and fix this! In the Python file, search for the following line:

   ```
   params = {'kernel': 'linear'}
   ```

   Replace the preceding line with the following:

   ```
   params = {'kernel': 'linear', 'class_weight': 'auto'}
   ```

   The `class_weight` parameter will count the number of datapoints in each class to adjust the weights so that the imbalance doesn't adversely affect the performance.
6. You will get the following figure once you run this code:



7. Let's look at the classification report, as follows:

```
##############################

Classification report on test dataset

             precision    recall  f1-score   support

    Class-0       0.29      0.76      0.42        42
    Class-1       0.95      0.70      0.80       258

avg / total       0.86      0.71      0.75       300

##############################
```

As we can see, `Class-0` is now detected with nonzero percentage accuracy.

Extracting confidence measurements

It would be nice to know the confidence with which we classify unknown data. When a new datapoint is classified into a known category, we can train the SVM to compute the confidence level of this output as well.

How to do it…

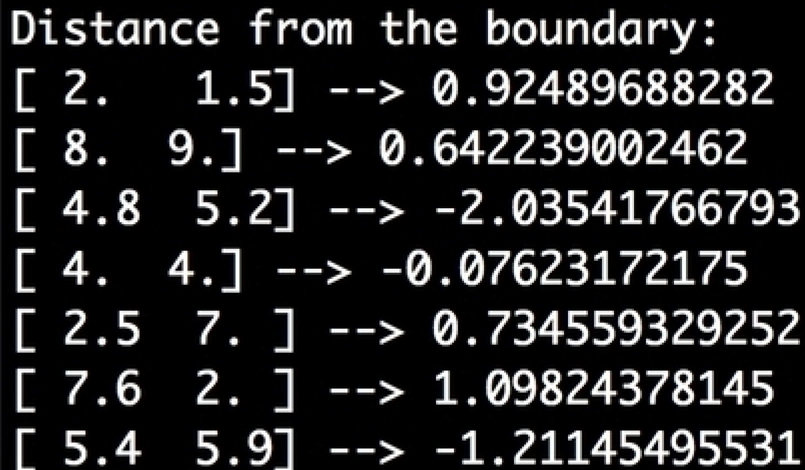
1. The full code is given in the `svm_confidence.py` file already provided to you. We will only discuss the core of the recipe here. Let's define some input data:

```
# Measure distance from the boundary
input_datapoints = np.array([[2, 1.5], [8, 9], [4.8, 5.2], [4,
4], [2.5, 7], [7.6, 2], [5.4, 5.9]])
```

2. Let's measure the distance from the boundary:

```
print "\nDistance from the boundary:"
for i in input_datapoints:
    print i, '-->', classifier.decision_function(i)[0]
```

3. You will see the following printed on your Terminal:



4. Distance from the boundary gives us some information about the datapoint, but it doesn't exactly tell us how confident the classifier is about the output tag. To do this, we need **Platt scaling**. This is a method that converts the distance measure into probability measure between classes. You can check out the following tutorial to learn more about Platt scaling: http://fastml.com/classifier-calibration-with-platts-scaling-and-isotonic-regression. Let's go ahead and train an SVM using Platt scaling:

```
# Confidence measure
params = {'kernel': 'rbf', 'probability': True}
classifier = SVC(**params)
```

The `probability` parameter tells the SVM that it should train to compute the probabilities as well.

5. Let's train the classifier:

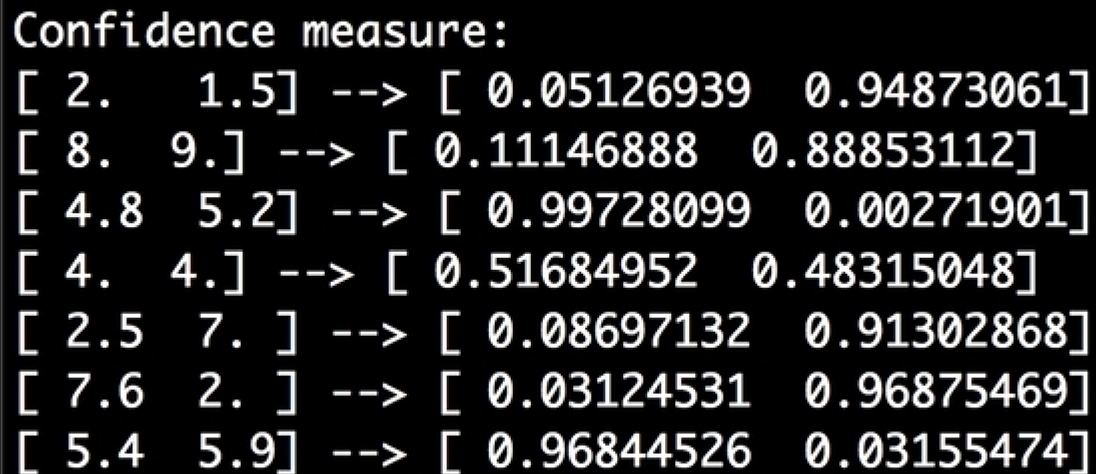
```
classifier.fit(X_train, y_train)
```

6. Let's compute the confidence measurements for these input datapoints:

```
print "\nConfidence measure:"
for i in input_datapoints:
    print i, '-->', classifier.predict_proba(i)[0]
```

The `predict_proba` function measures the confidence value.

7. You will see the following on your Terminal:



8. Let's see where the points are with respect to the boundary:

```
utilities.plot_classifier(classifier, input_datapoints,
[0]*len(input_datapoints), 'Input datapoints', 'True')
```

9. If you run this, you will get the following figure:

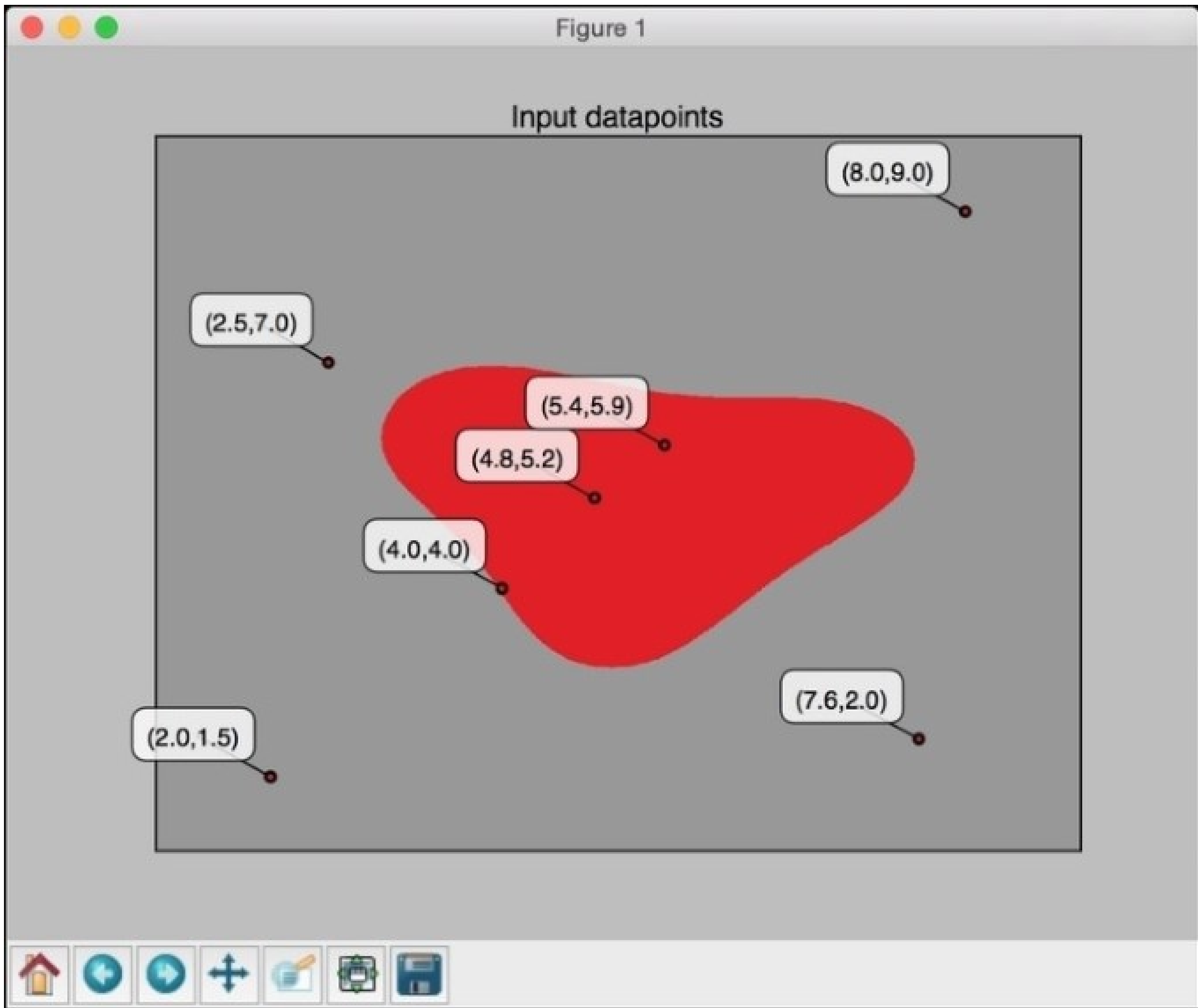


Figure 1
Input datapoints
(8.0,9.0)
(2.5,7.0)
(5.4,5.9)
(4.8,5.2)
(4.0,4.0)
(7.6,2.0)
(2.0,1.5)

Finding optimal hyperparameters

As discussed in the previous chapter, hyperparameters are important in determining the performance of a classifier. Let's see how to extract optimal hyperparameters for SVMs.

How to do it…

1. The full code is given in the `perform_grid_search.py` file that's already provided to you. We will only discuss the core parts of the recipe here. We will use cross-validation here, which we covered in the previous recipes. Once you load the data and split it into training and testing datasets, add the following to the file:

```
# Set the parameters by cross-validation
parameter_grid = [  {'kernel': ['linear'], 'C': [1, 10, 50, 600]},
                    {'kernel': ['poly'], 'degree': [2, 3]},
                    {'kernel': ['rbf'], 'gamma': [0.01, 0.001], 'C': [1, 10, 50, 600]},
                 ]
```

2. Let's define the metrics that we want to use:

```
metrics = ['precision', 'recall_weighted']
```

3. Let's start the search for optimal hyperparameters for each of the metrics:

```
for metric in metrics:
    print "\n#### Searching optimal hyperparameters for", metric

    classifier = grid_search.GridSearchCV(svm.SVC(C=1),
            parameter_grid, cv=5, scoring=metric)
    classifier.fit(X_train, y_train)
```

4. Let's look at the scores:

```
    print "\nScores across the parameter grid:"
    for params, avg_score, _ in classifier.grid_scores_:
        print params, '-->', round(avg_score, 3)
```

5. Let's print the best parameter set:

```
    print "\nHighest scoring parameter set:", classifier.best_params_
```

6. If you run this code, you will see the following on your Terminal:

```
#### Searching optimal hyperparameters for precision

Scores across the parameter grid:
{'kernel': 'linear', 'C': 1} --> 0.676
{'kernel': 'linear', 'C': 10} --> 0.676
{'kernel': 'linear', 'C': 50} --> 0.676
{'kernel': 'linear', 'C': 600} --> 0.676
{'kernel': 'poly', 'degree': 2} --> 0.872
{'kernel': 'poly', 'degree': 3} --> 0.872
{'kernel': 'rbf', 'C': 1, 'gamma': 0.01} --> 0.98
{'kernel': 'rbf', 'C': 1, 'gamma': 0.001} --> 0.533
{'kernel': 'rbf', 'C': 10, 'gamma': 0.01} --> 0.983
{'kernel': 'rbf', 'C': 10, 'gamma': 0.001} --> 0.543
{'kernel': 'rbf', 'C': 50, 'gamma': 0.01} --> 0.959
{'kernel': 'rbf', 'C': 50, 'gamma': 0.001} --> 0.806
{'kernel': 'rbf', 'C': 600, 'gamma': 0.01} --> 0.967
{'kernel': 'rbf', 'C': 600, 'gamma': 0.001} --> 0.983

Highest scoring parameter set: {'kernel': 'rbf', 'C': 10, 'gamma': 0.01}
```

7. As we can see in the preceding figure, it searches for all the optimal hyperparameters. In this case, the hyperparameters are the type of kernel, the C value, and gamma. It will try out various combinations of these parameters to find the best parameters. Let's test it out on the testing dataset:

```
y_true, y_pred = y_test, classifier.predict(X_test)
print "\nFull performance report:\n"
print classification_report(y_true, y_pred)
```

8. If you run this code, you will see the following on your Terminal:

```
Full performance report:

             precision    recall  f1-score   support

        0.0       0.92      0.98      0.95        45
        1.0       0.96      0.87      0.91        30

avg / total       0.94      0.93      0.93        75
```

Building an event predictor

Let's apply all of this knowledge to a real-world problem. We will build an SVM to predict the number of people going in and out of a building. The dataset is available at https://archive.ics.uci.edu/ml/datasets/CalIt2+Building+People+Counts. We will use a slightly modified version of this dataset so that it's easier to analyze. The modified data is available in the `building_event_binary.txt` and `building_event_multiclass.txt` files that are already provided to you.

Getting ready

Let's understand the data format before we start building the model. Each line in `building_event_binary.txt` consists of six comma-separated strings. The ordering of these six strings is as follows:

- Day
- Date
- Time
- The number of people going out of the building
- The number of people coming into the building
- The output indicating whether or not it's an event

The first five strings form the input data, and our task is to predict whether or not an event is going on in the building.

Each line in `building_event_multiclass.txt` consists of six comma-separated strings. This is more granular than the previous file in the sense that the output is the exact type of event going on in the building. The ordering of these six strings is as follows:

- Day
- Date
- Time
- The number of people going out of the building
- The number of people coming into the building
- The output indicating the type of event

The first five strings form the input data and our task is to predict what type of event is going on in the building.

How to do it…

1. We will use `event.py` that's already provided to you for reference. Create a new Python file, and add the following lines:

```
import numpy as np
from sklearn import preprocessing
from sklearn.svm import SVC
```

```
input_file = 'building_event_binary.txt'

# Reading the data
X = []
count = 0
with open(input_file, 'r') as f:
    for line in f.readlines():
        data = line[:-1].split(',')
        X.append([data[0]] + data[2:])

X = np.array(X)
```

We just loaded all the data into `X`.

2. Let's convert the data into numerical form:

```
# Convert string data to numerical data
label_encoder = []
X_encoded = np.empty(X.shape)
for i,item in enumerate(X[0]):
    if item.isdigit():
        X_encoded[:, i] = X[:, i]
    else:
        label_encoder.append(preprocessing.LabelEncoder())
        X_encoded[:, i] = label_encoder[-1].fit_transform(X[:,
i])

X = X_encoded[:, :-1].astype(int)
y = X_encoded[:, -1].astype(int)
```

3. Let's train the SVM using the radial basis function, Platt scaling, and class balancing:

```
# Build SVM
params = {'kernel': 'rbf', 'probability': True, 'class_weight':
'auto'}
classifier = SVC(**params)
classifier.fit(X, y)
```

4. We are now ready to perform cross-validation:

```
# Cross validation
from sklearn import cross_validation

accuracy = cross_validation.cross_val_score(classifier,
        X, y, scoring='accuracy', cv=3)
print "Accuracy of the classifier: " +
str(round(100*accuracy.mean(), 2)) + "%"
```

5. Let's test our SVM on a new datapoint:

```
# Testing on single data instance
input_data = ['Tuesday', '12:30:00','21','23']
input_data_encoded = [-1] * len(input_data)
count = 0
for i,item in enumerate(input_data):
    if item.isdigit():
        input_data_encoded[i] = int(input_data[i])
    else:
        input_data_encoded[i] =
int(label_encoder[count].transform(input_data[i]))
        count = count + 1

input_data_encoded = np.array(input_data_encoded)

# Predict and print output for a particular datapoint
output_class = classifier.predict(input_data_encoded)
print "Output class:",
label_encoder[-1].inverse_transform(output_class)[0]
```

6. If you run this code, you will see the following output on your Terminal:

```
Accuracy of the classifier: 89.88%
Output class: event
```

7. If you use the `building_event_multiclass.txt` file as the input data file instead of `building_event_binary.txt`, you will see the following output on your Terminal:

```
Accuracy of the classifier: 65.9%
Output class: eventA
```

Estimating traffic

An interesting application of SVMs is to predict the traffic, based on related data. In the previous recipe, we used an SVM as a classifier. In this recipe, we will use it as a regressor to estimate the traffic.

Getting ready

We will use the dataset available at https://archive.ics.uci.edu/ml/datasets/Dodgers+Loop+Sensor. This is a dataset that counts the number of cars passing by during baseball games at Los Angeles Dodgers home stadium. We will use a slightly modified form of that dataset so that it's easier to analyze. You can use the `traffic_data.txt` file already provided to you. Each line in this file contains comma-separated strings formatted in the following manner:

- Day
- Time
- The opponent team
- Whether or not a baseball game is going on
- The number of cars passing by

How to do it…

1. Let's see how to build an SVM regressor. We will use `traffic.py` that's already provided to you as a reference. Create a new Python file, and add the following lines:

```
# SVM regressor to estimate traffic

import numpy as np
from sklearn import preprocessing
from sklearn.svm import SVR

input_file = 'traffic_data.txt'

# Reading the data
X = []
count = 0
with open(input_file, 'r') as f:
    for line in f.readlines():
        data = line[:-1].split(',')
        X.append(data)

X = np.array(X)
```

   We loaded all the input data into `X`.

2. Let's encode this data:

```
# Convert string data to numerical data
label_encoder = []
```

```
X_encoded = np.empty(X.shape)
for i,item in enumerate(X[0]):
    if item.isdigit():
        X_encoded[:, i] = X[:, i]
    else:
        label_encoder.append(preprocessing.LabelEncoder())
        X_encoded[:, i] = label_encoder[-1].fit_transform(X[:,
i])

X = X_encoded[:, :-1].astype(int)
y = X_encoded[:, -1].astype(int)
```

3. Let's build and train the SVM regressor using the radial basis function:

```
# Build SVR
params = {'kernel': 'rbf', 'C': 10.0, 'epsilon': 0.2}
regressor = SVR(**params)
regressor.fit(X, y)
```

   In the preceding lines, the `C` parameter specifies the penalty for misclassification and `epsilon` specifies the limit within which no penalty is applied.

4. Let's perform cross-validation to check the performance of the regressor:

```
# Cross validation
import sklearn.metrics as sm

y_pred = regressor.predict(X)
print "Mean absolute error =", round(sm.mean_absolute_error(y,
y_pred), 2)
```

5. Let's test it on a datapoint:

```
# Testing encoding on single data instance
input_data = ['Tuesday', '13:35', 'San Francisco', 'yes']
input_data_encoded = [-1] * len(input_data)
count = 0
for i,item in enumerate(input_data):
    if item.isdigit():
        input_data_encoded[i] = int(input_data[i])
    else:
        input_data_encoded[i] =
int(label_encoder[count].transform(input_data[i]))
        count = count + 1

input_data_encoded = np.array(input_data_encoded)

# Predict and print output for a particular datapoint
```

```
print "Predicted traffic:",
int(regressor.predict(input_data_encoded)[0])
```

6. If you run this code, you will see the following printed on your Terminal:

```
Mean absolute error = 4.08
Predicted traffic: 29
```

Chapter 4. Clustering with Unsupervised Learning

In this chapter, we will cover the following recipes:

- Clustering data using the k-means algorithm
- Compressing an image using vector quantization
- Building a Mean Shift clustering model
- Grouping data using agglomerative clustering
- Evaluating the performance of clustering algorithms
- Automatically estimating the number of clusters using DBSCAN algorithm
- Finding patterns in stock market data
- Building a customer segmentation model

Introduction

Unsupervised learning is a paradigm in machine learning where we build models without relying on labeled training data. Until this point, we dealt with data that was labeled in some way. This means that learning algorithms can look at this data and learn to categorize them based on labels. In the world of unsupervised learning, we don't have this luxury! These algorithms are used when we want to find subgroups within datasets using some similarity metric.

One of the most common methods is **clustering**. You must have heard this term being used quite frequently. We mainly use it for data analysis where we want to find clusters in our data. These clusters are usually found using certain kind of similarity measure such as Euclidean distance. Unsupervised learning is used extensively in many fields, such as data mining, medical imaging, stock market analysis, computer vision, market segmentation, and so on.

Clustering data using the k-means algorithm

The **k-means algorithm** is one of the most popular clustering algorithms. This algorithm is used to divide the input data into *k* subgroups using various attributes of the data. Grouping is achieved using an optimization technique where we try to minimize the sum of squares of distances between the datapoints and the corresponding centroid of the cluster. If you need a quick refresher, you can learn more about k-means at http://www.onmyphd.com/?p=k-means.clustering&ckattempt=1.

How to do it…

1. The full code for this recipe is given in the `kmeans.py` file already provided to you. Let's look at how it's built. Create a new Python file, and import the following packages:

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn import metrics
from sklearn.cluster import KMeans

import utilities
```

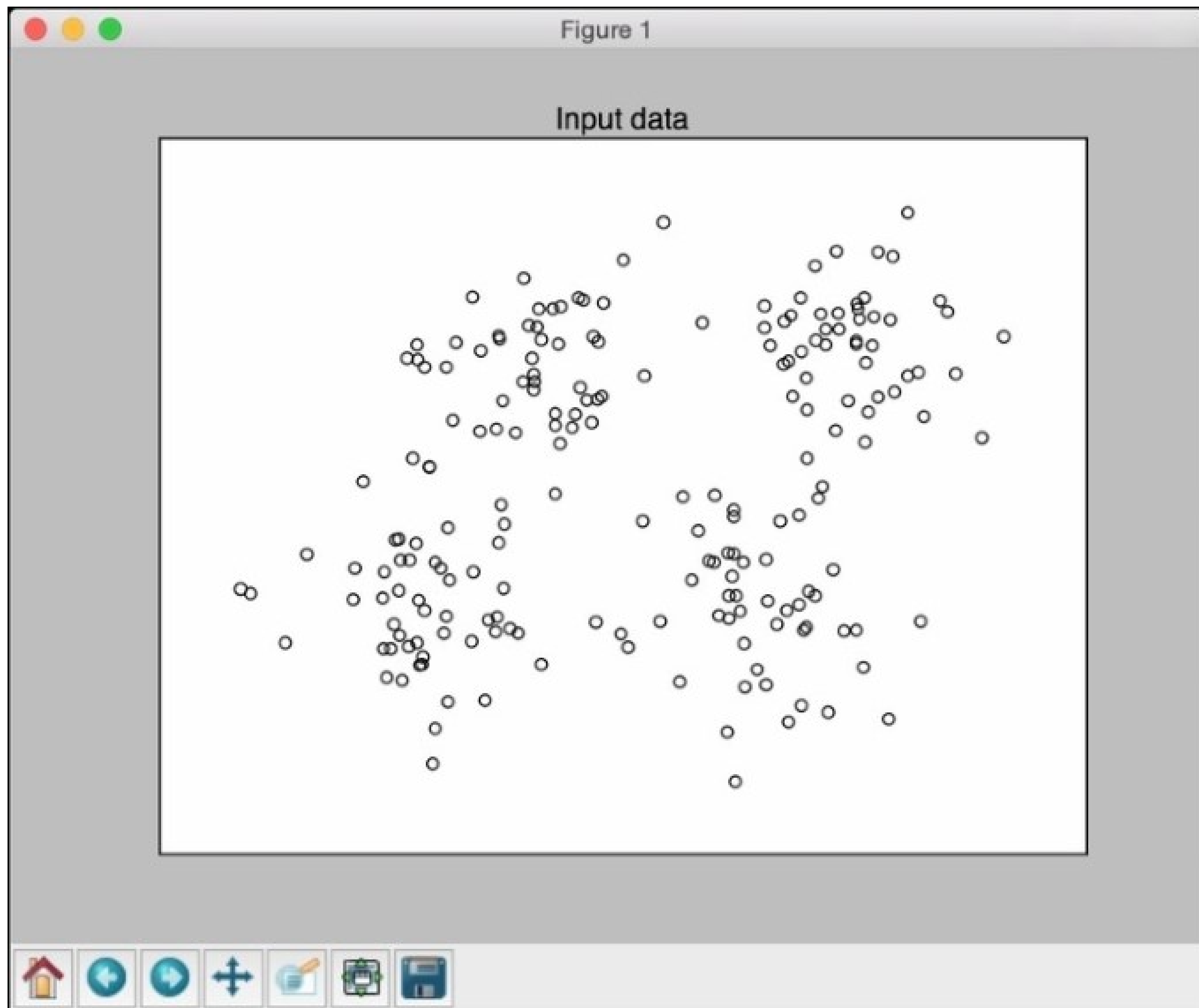
2. Let's load the input data and define the number of clusters. We will use the `data_multivar.txt` file that's already provided to you:

```
data = utilities.load_data('data_multivar.txt')
num_clusters = 4
```

3. We need to see what the input data looks like. Let's go ahead and add the following lines of the code to the Python file:

```
plt.figure()
plt.scatter(data[:,0], data[:,1], marker='o',
        facecolors='none', edgecolors='k', s=30)
x_min, x_max = min(data[:, 0]) - 1, max(data[:, 0]) + 1
y_min, y_max = min(data[:, 1]) - 1, max(data[:, 1]) + 1
plt.title('Input data')
plt.xlim(x_min, x_max)
plt.ylim(y_min, y_max)
plt.xticks(())
plt.yticks(())
```

If you run this code, you will get the following figure:



4. We are now ready to train the model. Let's initialize the k-means object and train it:

```
kmeans = KMeans(init='k-means++', n_clusters=num_clusters,
n_init=10)
kmeans.fit(data)
```

5. Now that the data is trained, we need to visualize the boundaries. Let's go ahead and add the following lines of code to the Python file:

```
# Step size of the mesh
step_size = 0.01

# Plot the boundaries
x_min, x_max = min(data[:, 0]) - 1, max(data[:, 0]) + 1
y_min, y_max = min(data[:, 1]) - 1, max(data[:, 1]) + 1
```

```
x_values, y_values = np.meshgrid(np.arange(x_min, x_max,
step_size), np.arange(y_min, y_max, step_size))

# Predict labels for all points in the mesh
predicted_labels = kmeans.predict(np.c_[x_values.ravel(),
y_values.ravel()])
```

6. We just evaluated the model across a grid of points. Let's plot these results to view the boundaries:

```
# Plot the results
predicted_labels = predicted_labels.reshape(x_values.shape)
plt.figure()
plt.clf()
plt.imshow(predicted_labels, interpolation='nearest',
           extent=(x_values.min(), x_values.max(),
y_values.min(), y_values.max()),
           cmap=plt.cm.Paired,
           aspect='auto', origin='lower')

plt.scatter(data[:,0], data[:,1], marker='o',
        facecolors='none', edgecolors='k', s=30)
```

7. Let's overlay the centroids on top of it:

```
centroids = kmeans.cluster_centers_
plt.scatter(centroids[:,0], centroids[:,1], marker='o', s=200,
linewidths=3,
        color='k', zorder=10, facecolors='black')
x_min, x_max = min(data[:, 0]) - 1, max(data[:, 0]) + 1
y_min, y_max = min(data[:, 1]) - 1, max(data[:, 1]) + 1
plt.title('Centoids and boundaries obtained using KMeans')
plt.xlim(x_min, x_max)
plt.ylim(y_min, y_max)
plt.xticks(())
plt.yticks(())
plt.show()
```

If you run this code, you should see the following figure:

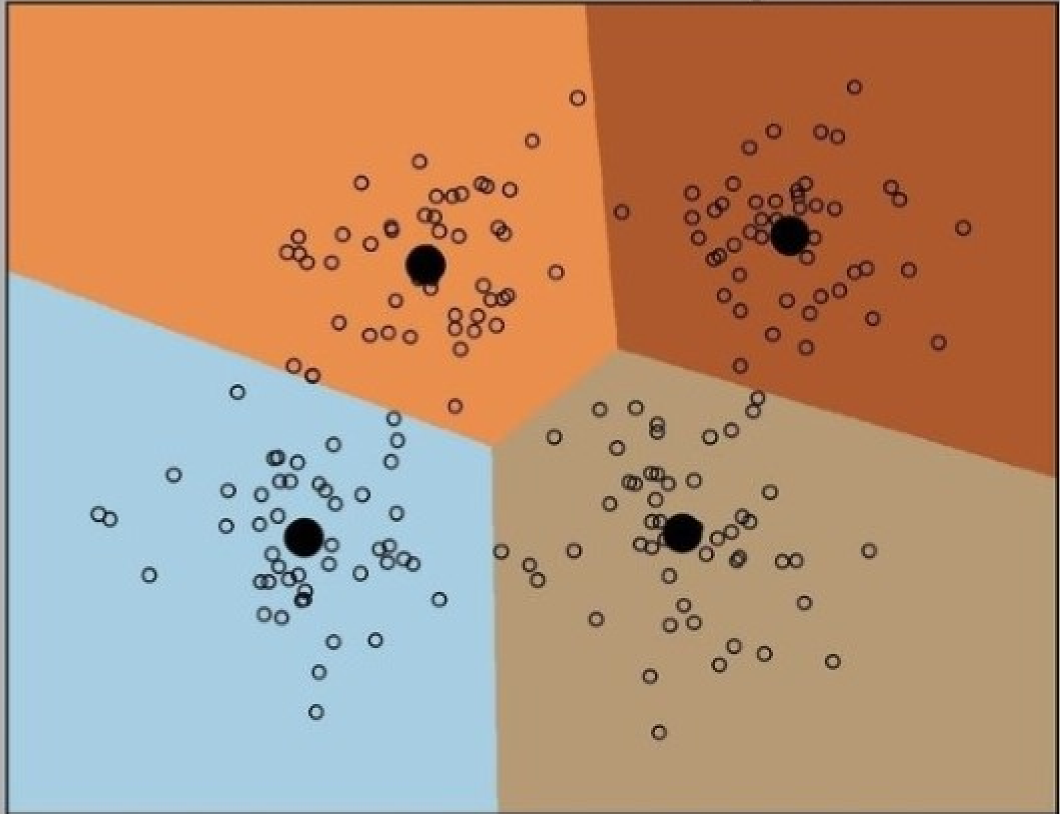
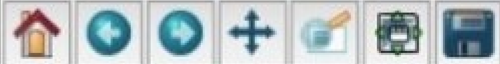


Figure 2
Centoids and boundaries obtained using KMeans



Compressing an image using vector quantization

One of the main applications of k-means clustering is **vector quantization**. Simply speaking, vector quantization is the *N*-dimensional version of "rounding off". When we deal with 1D data, such as numbers, we use the rounding-off technique to reduce the memory needed to store that value. For example, instead of storing 23.73473572, we just store 23.73 if we want to be accurate up to the second decimal place. Or, we can just store 24 if we don't care about decimal places. It depends on our needs and the trade-off that we are willing to make.

Similarly, when we extend this concept to *N*-dimensional data, it becomes vector quantization. Of course there are more nuances to it! You can learn more about it at http://www.data-compression.com/vq.shtml. Vector quantization is popularly used in image compression where we store each pixel using fewer bits than the original image to achieve compression.

How to do it…

1. The full code for this recipe is given in the `vector_quantization.py` file already provided to you. Let's look at how it's built. We'll start by importing the required packages. Create a new Python file, and add the following lines:

```
import argparse

import numpy as np
from scipy import misc
from sklearn import cluster
import matplotlib.pyplot as plt
```

2. Let's create a function to parse the input arguments. We will be able to pass the image and the number of bits per pixel as input arguments:

```
def build_arg_parser():
    parser = argparse.ArgumentParser(description='Compress the
input image \
            using clustering')
    parser.add_argument("--input-file", dest="input_file",
required=True,
            help="Input image")
    parser.add_argument("--num-bits", dest="num_bits",
required=False,
            type=int, help="Number of bits used to represent
each pixel")
    return parser
```

3. Let's create a function to compress the input image:

```
def compress_image(img, num_clusters):
    # Convert input image into (num_samples, num_features)
    # array to run kmeans clustering algorithm
```

```
    X = img.reshape((-1, 1))

    # Run kmeans on input data
    kmeans = cluster.KMeans(n_clusters=num_clusters, n_init=4,
random_state=5)
    kmeans.fit(X)
    centroids = kmeans.cluster_centers_.squeeze()
    labels = kmeans.labels_

    # Assign each value to the nearest centroid and
    # reshape it to the original image shape
    input_image_compressed = np.choose(labels,
centroids).reshape(img.shape)

    return input_image_compressed
```

4. Once we compress the image, we need to see how it affects the quality. Let's define a function to plot the output image:

```
def plot_image(img, title):
    vmin = img.min()
    vmax = img.max()
    plt.figure()
    plt.title(title)
    plt.imshow(img, cmap=plt.cm.gray, vmin=vmin, vmax=vmax)
```

5. We are now ready to use all these functions. Let's define the `main` function that takes the input arguments, processes them, and extracts the output image:

```
if __name__=='__main__':
    args = build_arg_parser().parse_args()
    input_file = args.input_file
    num_bits = args.num_bits

    if not 1 <= num_bits <= 8:
        raise TypeError('Number of bits should be between 1 and
8')

    num_clusters = np.power(2, num_bits)

    # Print compression rate
    compression_rate = round(100 * (8.0 - args.num_bits) / 8.0,
2)
    print "\nThe size of the image will be reduced by a factor
of", 8.0/args.num_bits
    print "\nCompression rate = " + str(compression_rate) + "%"
```

6. Let's load the input image:

```
    # Load input image
    input_image = misc.imread(input_file, True).astype(np.uint8)

    # original image
    plot_image(input_image, 'Original image')
```

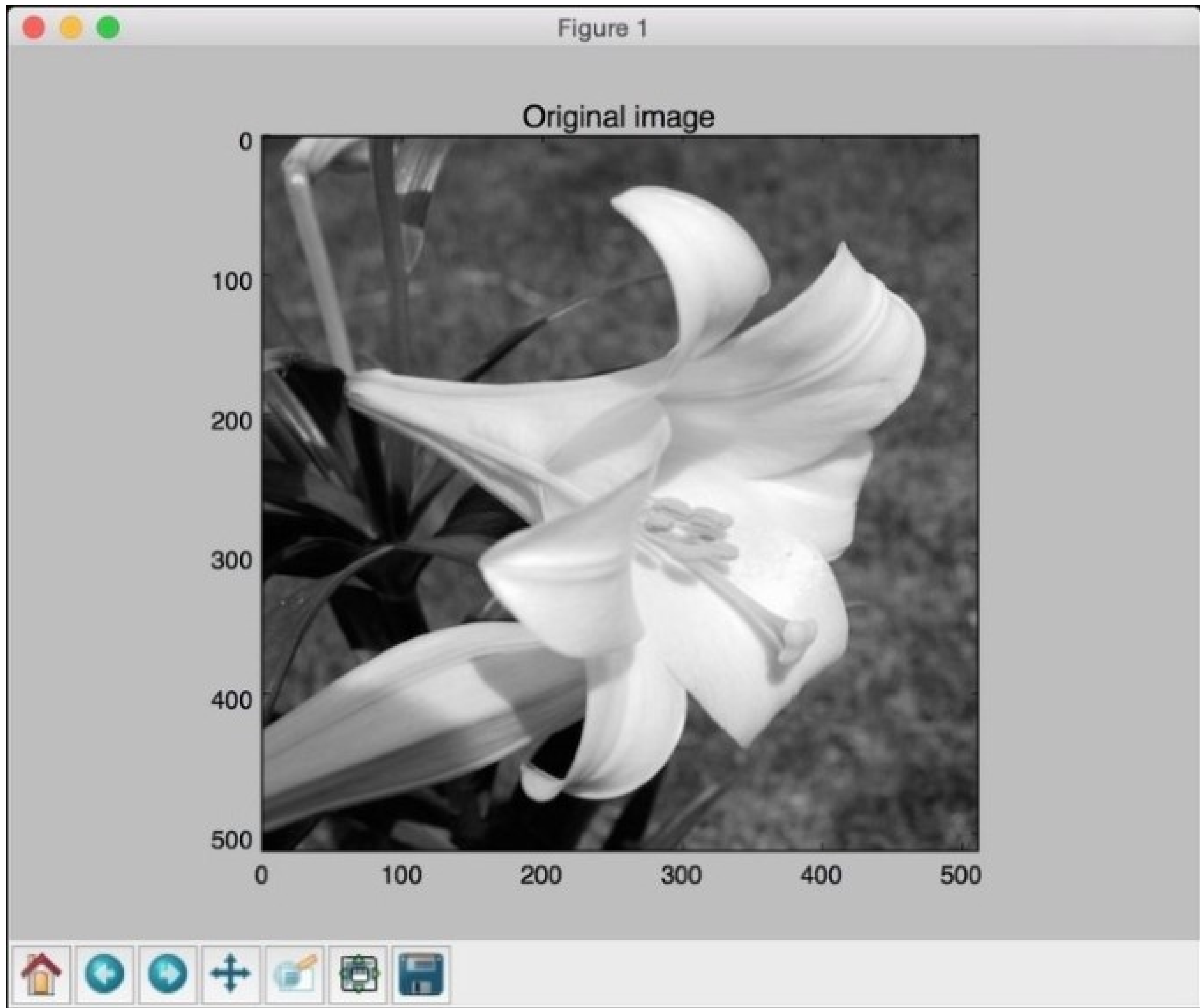
7. Let's compress this image using the input argument:

```
    # compressed image
    input_image_compressed = compress_image(input_image, 
num_clusters)
    plot_image(input_image_compressed, 'Compressed image; 
compression rate = '
            + str(compression_rate) + '%')

    plt.show()
```

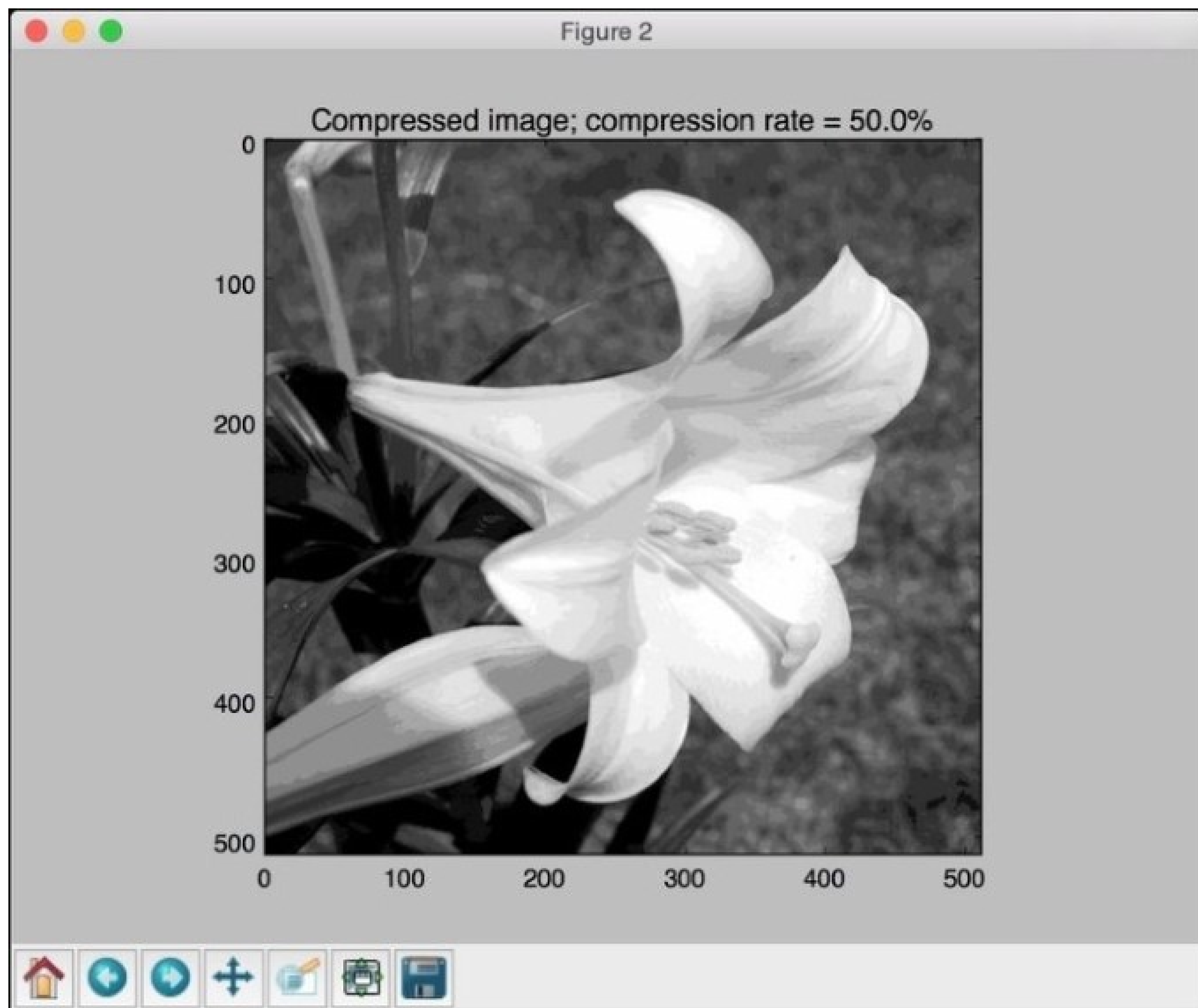
8. We are now ready to run the code. Run the following command on your Terminal:

```
$ python vector_quantization.py --input-file flower_image.jpg 
--num-bits 4
```

The input image looks like the following:



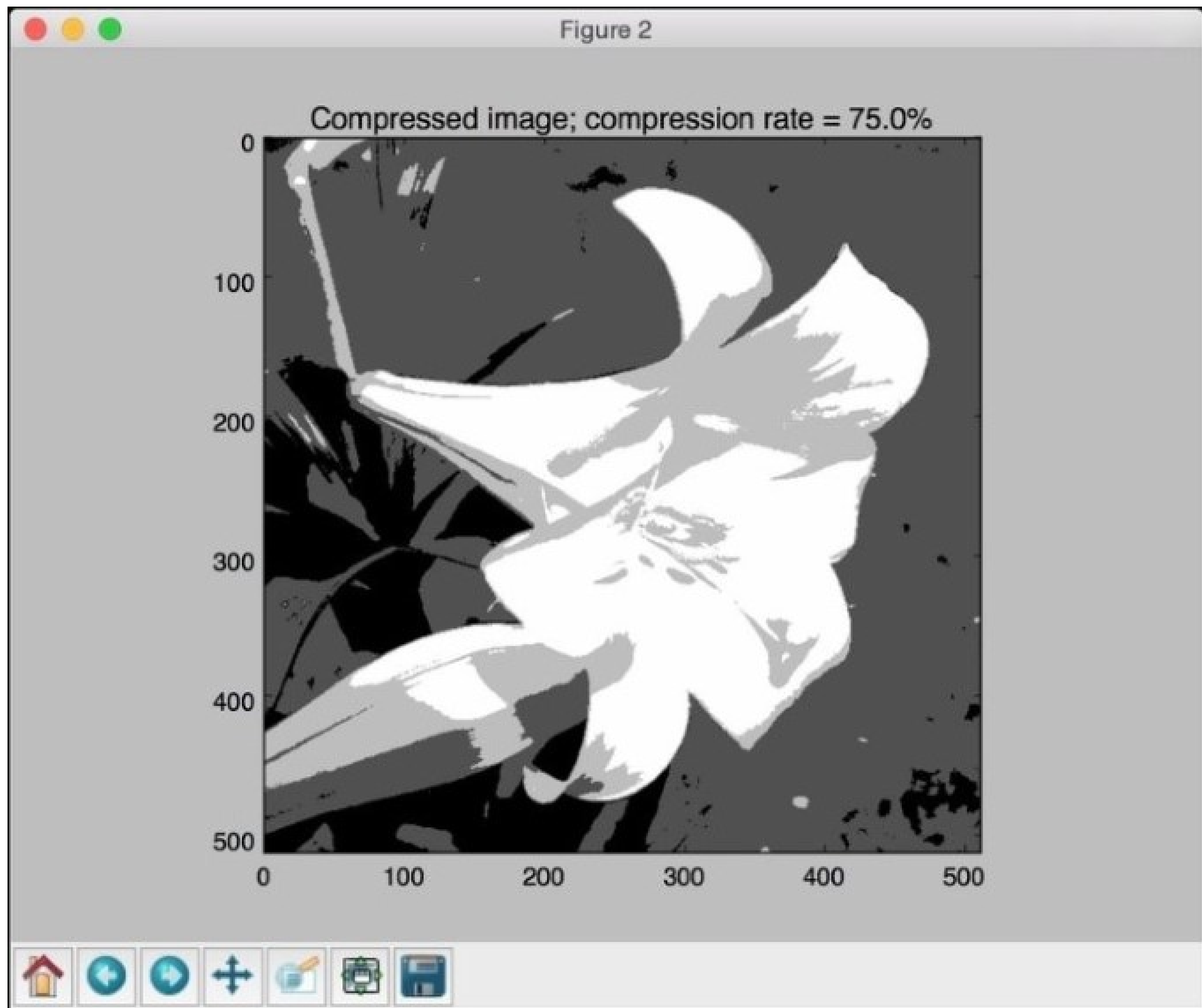
You should get the following compressed image as the output:



9. Let's compress the image further by reducing the number of bits to 2. Run the following command on your Terminal:

```
$ python vector_quantization.py --input-file flower_image.jpg
--num-bits 2
```

You should get the following compressed image as the output:



10. If you reduce the number of bits to 1, you can see that it will become a binary image with black and white as the only two colors. Run the following command:

```
$ python vector_quantization.py --input-file flower_image.jpg
--num-bits 1
```

You will get the following output:

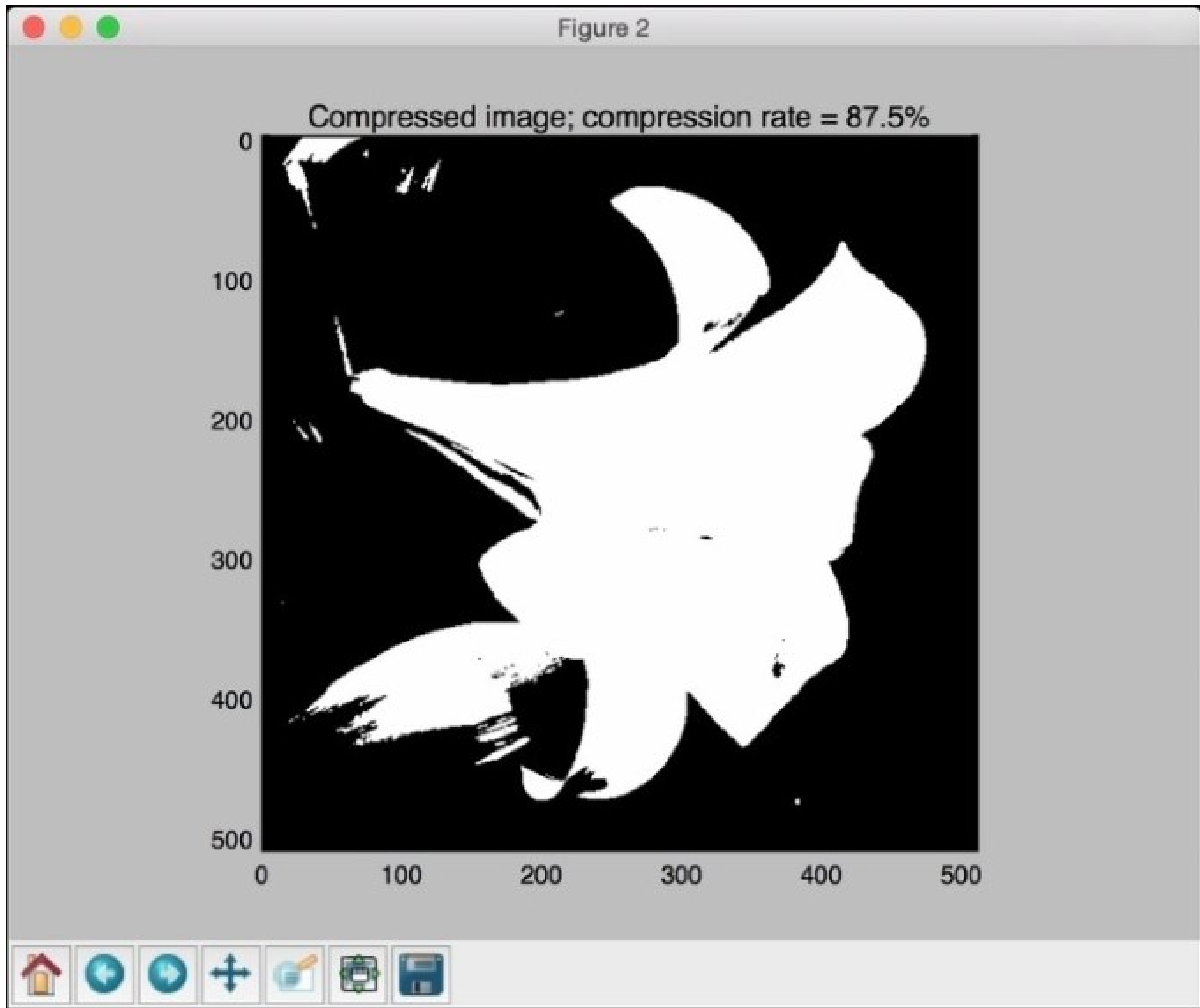


Figure 2
Compressed image; compression rate = 87.5%
0
100
200
300
400
500
0
100
200
300
400
500

Building a Mean Shift clustering model

The **Mean Shift** is a powerful unsupervised learning algorithm that's used to cluster datapoints. It considers the distribution of datapoints as a probability-density function and tries to find the *modes* in the feature space. These *modes* are basically points corresponding to local maxima. The main advantage of Mean Shift algorithm is that we are not required to know the number of clusters beforehand.

Let's say that we have a set of input points, and we are trying to find clusters in them without knowing how many clusters we are looking for. Mean Shift algorithm considers these points to be sampled from a probability density function. If there are clusters in the datapoints, then they correspond to the peaks of that probability-density function. The algorithm starts from random points and iteratively converges toward these peaks. You can learn more about it at http://homepages.inf.ed.ac.uk/rbf/CVonline/LOCAL_COPIES/TUZEL1/MeanShift.pdf.

How to do it…

1. The full code for this recipe is given in the `mean_shift.py` file that's already provided to you. Let's look at how it's built. Create a new Python file, and import a couple of required packages:

   ```
   import numpy as np
   from sklearn.cluster import MeanShift, estimate_bandwidth

   import utilities
   ```

2. Let's load the input data from the `data_multivar.txt` file that's already provided to you:

   ```
   # Load data from input file
   X = utilities.load_data('data_multivar.txt')
   ```

3. Build a Mean Shift clustering model by specifying the input parameters:

   ```
   # Estimating the bandwidth
   bandwidth = estimate_bandwidth(X, quantile=0.1,
   n_samples=len(X))

   # Compute clustering with MeanShift
   meanshift_estimator = MeanShift(bandwidth=bandwidth,
   bin_seeding=True)
   ```

4. Train the model:

   ```
   meanshift_estimator.fit(X)
   ```

5. Extract the labels:

   ```
   labels = meanshift_estimator.labels_
   ```

6. Extract the centroids of the clusters from the model and print out the number of clusters:

```
centroids = meanshift_estimator.cluster_centers_
num_clusters = len(np.unique(labels))

print "Number of clusters in input data =", num_clusters
```

7. Let's go ahead and visualize it:

```
# Plot the points and centroids
import matplotlib.pyplot as plt
from itertools import cycle

plt.figure()

# specify marker shapes for different clusters
markers = '.*xv'
```

8. Iterate through the datapoints and plot them:

```
for i, marker in zip(range(num_clusters), markers):
    # plot the points belong to the current cluster
    plt.scatter(X[labels==i, 0], X[labels==i, 1],
marker=marker, color='k')

    # plot the centroid of the current cluster
    centroid = centroids[i]
    plt.plot(centroid[0], centroid[1], marker='o',
markerfacecolor='k',
             markeredgecolor='k', markersize=15)

plt.title('Clusters and their centroids')
plt.show()
```

9. If you run this code, you will get the following output:

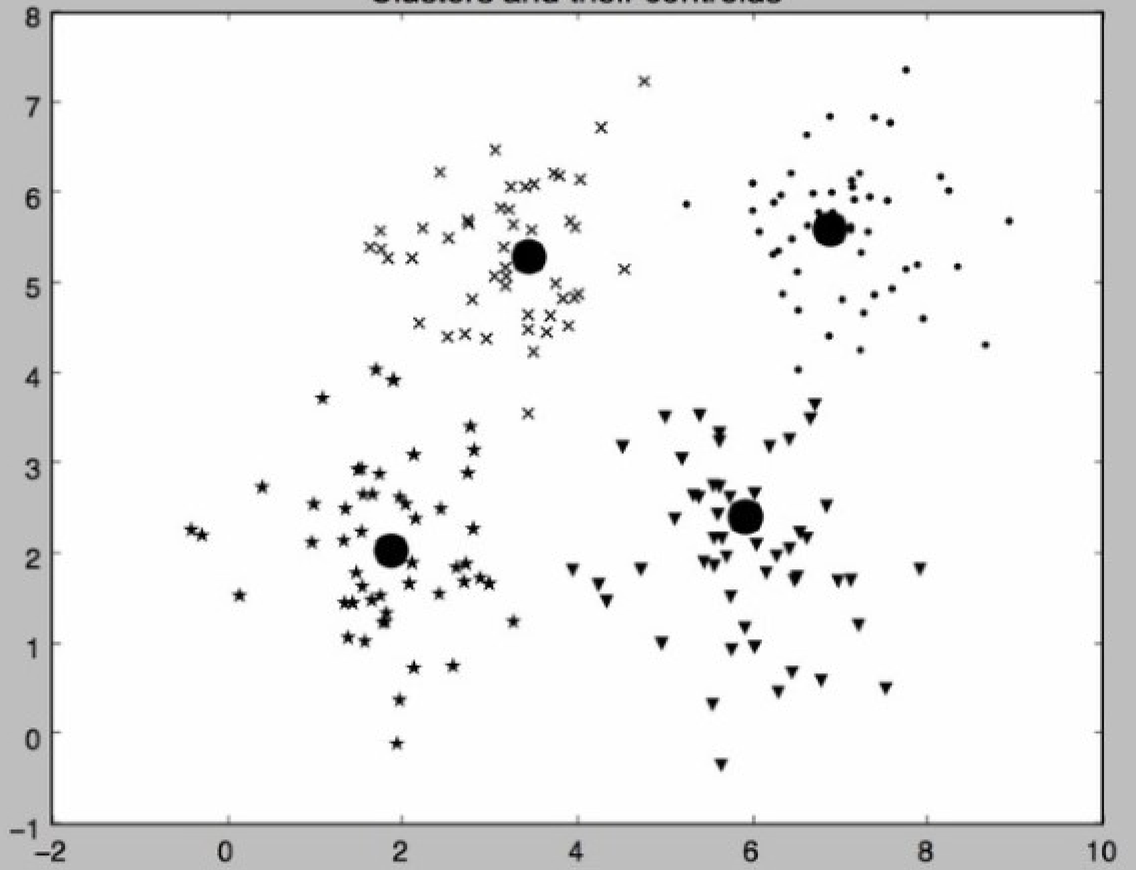
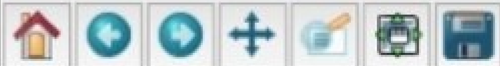


Figure 1
Clusters and their centroids
8
7
6
5
4
3
2
1
0
−1
−2
0
2
4
6
8
10



Grouping data using agglomerative clustering

Before we talk about agglomerative clustering, we need to understand hierarchical clustering. **Hierarchical clustering** refers to a set of clustering algorithms that build tree-like clusters by successively splitting or merging them. This hierarchical structure is represented using a tree.

Hierarchical clustering algorithms can be either bottom-up or top-down. Now what does this mean? In bottom-up algorithms, each datapoint is treated as a separate cluster with a single object. These clusters are then successively merged until all the clusters are merged into a single giant cluster. This is called **agglomerative clustering**. On the other hand, top-down algorithms start with a giant cluster and successively split these clusters until individual datapoints are reached. You can learn more about it at http://nlp.stanford.edu/IR-book/html/htmledition/hierarchical-agglomerative-clustering-1.html.

How to do it…

1. The full code for this recipe is given in the `agglomerative.py` file that's provided to you. Let's look at how it's built. Create a new Python file, and import the necessary packages:

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.cluster import AgglomerativeClustering
from sklearn.neighbors import kneighbors_graph
```

2. Let's define the function that we need to perform agglomerative clustering:

```
def perform_clustering(X, connectivity, title, num_clusters=3,
linkage='ward'):
    plt.figure()
    model = AgglomerativeClustering(linkage=linkage,
                    connectivity=connectivity,
n_clusters=num_clusters)
    model.fit(X)
```

3. Let's extract the labels and specify the shapes of the markers for the graph:

```
    # extract labels
    labels = model.labels_
`
    # specify marker shapes for different clusters
    markers = '.vx'
```

4. Iterate through the datapoints and plot them accordingly using different markers:

```
    for i, marker in zip(range(num_clusters), markers):
        # plot the points belong to the current cluster
        plt.scatter(X[labels==i, 0], X[labels==i, 1], s=50,
                    marker=marker, color='k', facecolors='none')
```

```
    plt.title(title)
```

5. In order to demonstrate the advantage of agglomerative clustering, we need to run it on datapoints that are linked spatially but also located close to each other in space. We want the linked datapoints to belong to the same cluster as opposed to datapoints that are just spatially close to each other. Let's define a function to get a set of datapoints on a spiral:

```
def get_spiral(t, noise_amplitude=0.5):
    r = t
    x = r * np.cos(t)
    y = r * np.sin(t)

    return add_noise(x, y, noise_amplitude)
```

6. In the previous function, we added some noise the curve because it adds some uncertainty. Let's define this function:

```
def add_noise(x, y, amplitude):
    X = np.concatenate((x, y))
    X += amplitude * np.random.randn(2, X.shape[1])
    return X.T
```

7. Let's define another function to get datapoints located on a rose curve:

```
def get_rose(t, noise_amplitude=0.02):
    # Equation for "rose" (or rhodonea curve); if k is odd, then
    # the curve will have k petals, else it will have 2k petals
    k = 5
    r = np.cos(k*t) + 0.25
    x = r * np.cos(t)
    y = r * np.sin(t)

    return add_noise(x, y, noise_amplitude)
```

8. Just to add more variety, let's also define a **hypotrochoid**:

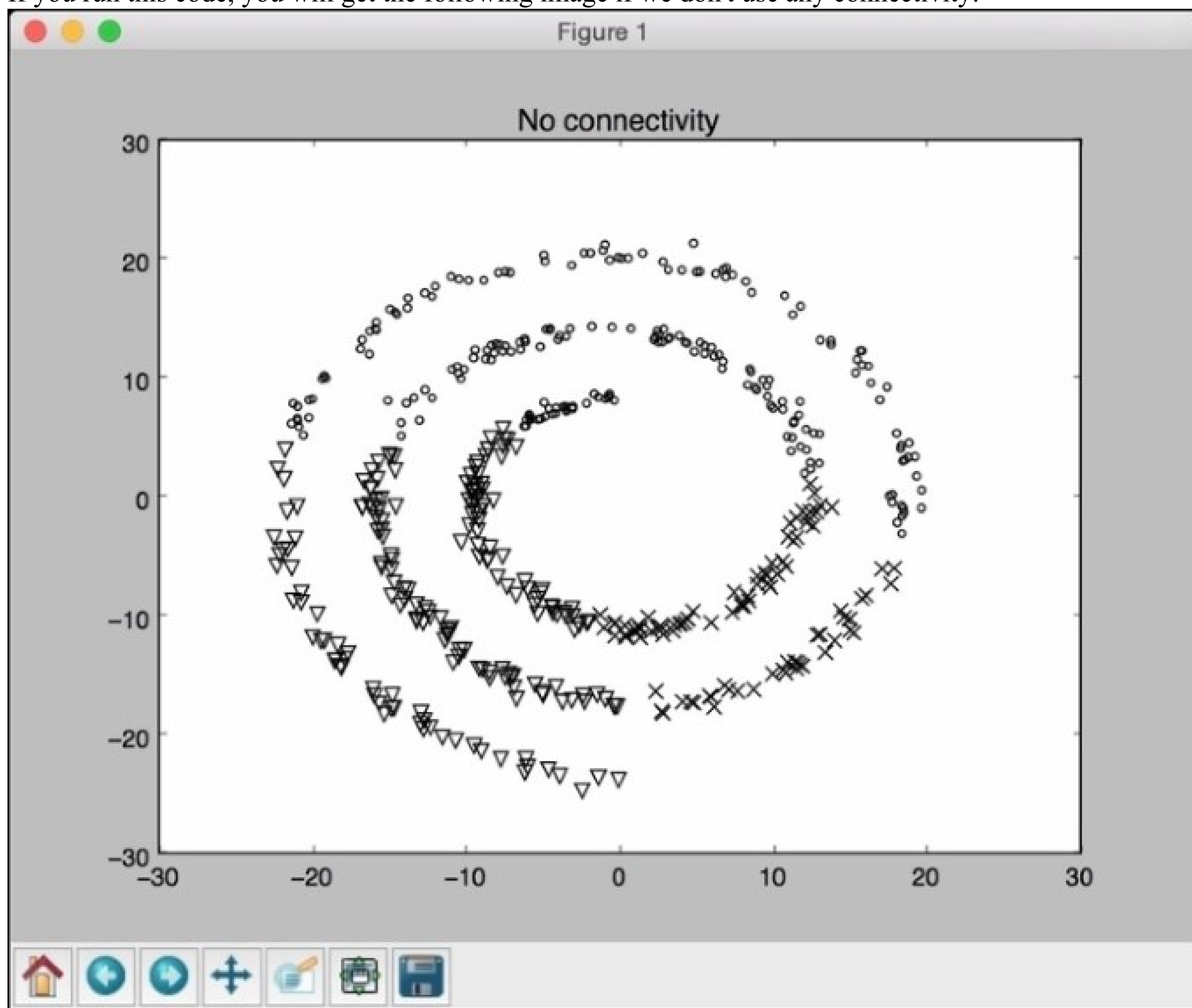
```
def get_hypotrochoid(t, noise_amplitude=0):
    a, b, h = 10.0, 2.0, 4.0
    x = (a - b) * np.cos(t) + h * np.cos((a - b) / b * t)
    y = (a - b) * np.sin(t) - h * np.sin((a - b) / b * t)

    return add_noise(x, y, 0)
```

9. We are now ready to define the `main` function:

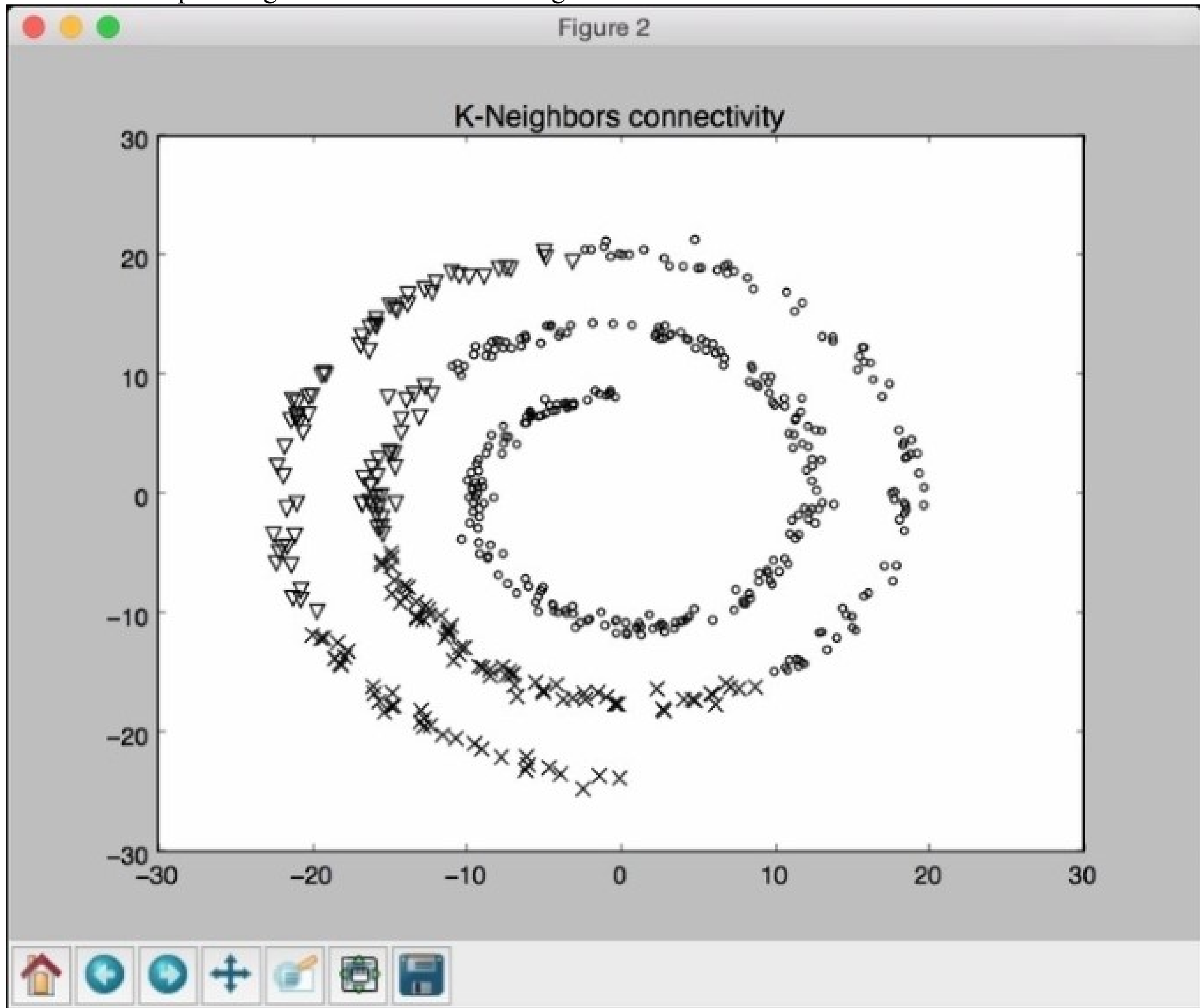
```
if __name__=='__main__':
    # Generate sample data
    n_samples = 500
    np.random.seed(2)
```

```
    t = 2.5 * np.pi * (1 + 2 * np.random.rand(1, n_samples))
    X = get_spiral(t)

    # No connectivity
    connectivity = None
    perform_clustering(X, connectivity, 'No connectivity')

    # Create K-Neighbors graph
    connectivity = kneighbors_graph(X, 10, include_self=False)
    perform_clustering(X, connectivity, 'K-Neighbors
connectivity')

    plt.show()
```

10. If you run this code, you will get the following image if we don't use any connectivity:



11. The second output image looks like the following:



As we can see, using the connectivity feature enables us to the group datapoints that are linked to each other as opposed to clustering them, based on their spatial locations.

Evaluating the performance of clustering algorithms

So far, we built different clustering algorithms but didn't measure their performances. In supervised learning, we just compare the predicted values with the original labels to compute their accuracy. In unsupervised learning, we don't have any labels. Therefore, we need a way to measure the performance of our algorithms.

A good way to measure a clustering algorithm is by seeing how well the clusters are separated. Are the clusters well separated? Are the datapoints in a cluster tight enough? We need a metric that can quantify this behavior. We will use a metric, called **Silhouette Coefficient** score. This score is defined for each datapoint. This coefficient is defined as follows:

$score = (x - y) / max(x, y)$

Here, `x` is the average distance between the current datapoint and all the other datapoints in the same cluster; `y` is the average distance between the current datapoint and all the datapoints in the next nearest cluster.

How to do it…

1. The full code for this recipe is given in the `performance.py` file that's already provided to you. Let's look at how it's built. Create a new Python file, and import the following packages:

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn import metrics
from sklearn.cluster import KMeans

import utilities
```

2. Let's load the input data from the `data_perf.txt` file already provided to you:

```
# Load data
data = utilities.load_data('data_perf.txt')
```

3. In order to determine the optimal number of clusters, let's iterate through a range of values and see where it peaks:

```
scores = []
range_values = np.arange(2, 10)

for i in range_values:
    # Train the model
    kmeans = KMeans(init='k-means++', n_clusters=i, n_init=10)
    kmeans.fit(data)
    score = metrics.silhouette_score(data, kmeans.labels_,
```

```
                metric='euclidean', sample_size=len(data))

    print "\nNumber of clusters =", i
    print "Silhouette score =", score

    scores.append(score)
```

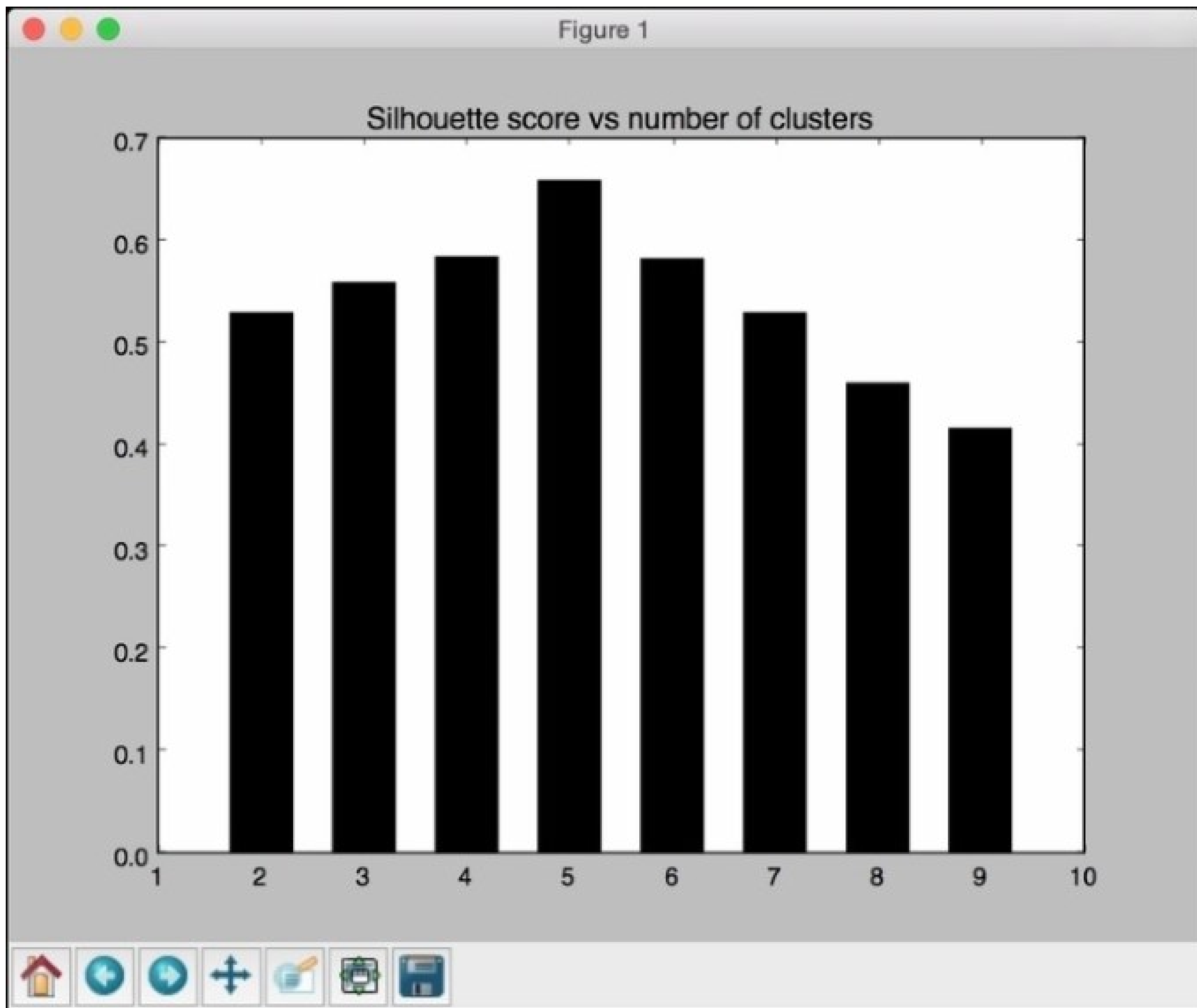
4. Let's plot the graph to see where it peaked:

```
# Plot scores
plt.figure()
plt.bar(range_values, scores, width=0.6, color='k',
align='center')
plt.title('Silhouette score vs number of clusters')

# Plot data
plt.figure()
plt.scatter(data[:,0], data[:,1], color='k', s=30, marker='o',
facecolors='none')
x_min, x_max = min(data[:, 0]) - 1, max(data[:, 0]) + 1
y_min, y_max = min(data[:, 1]) - 1, max(data[:, 1]) + 1
plt.title('Input data')
plt.xlim(x_min, x_max)
plt.ylim(y_min, y_max)
plt.xticks(())
plt.yticks(())

plt.show()
```

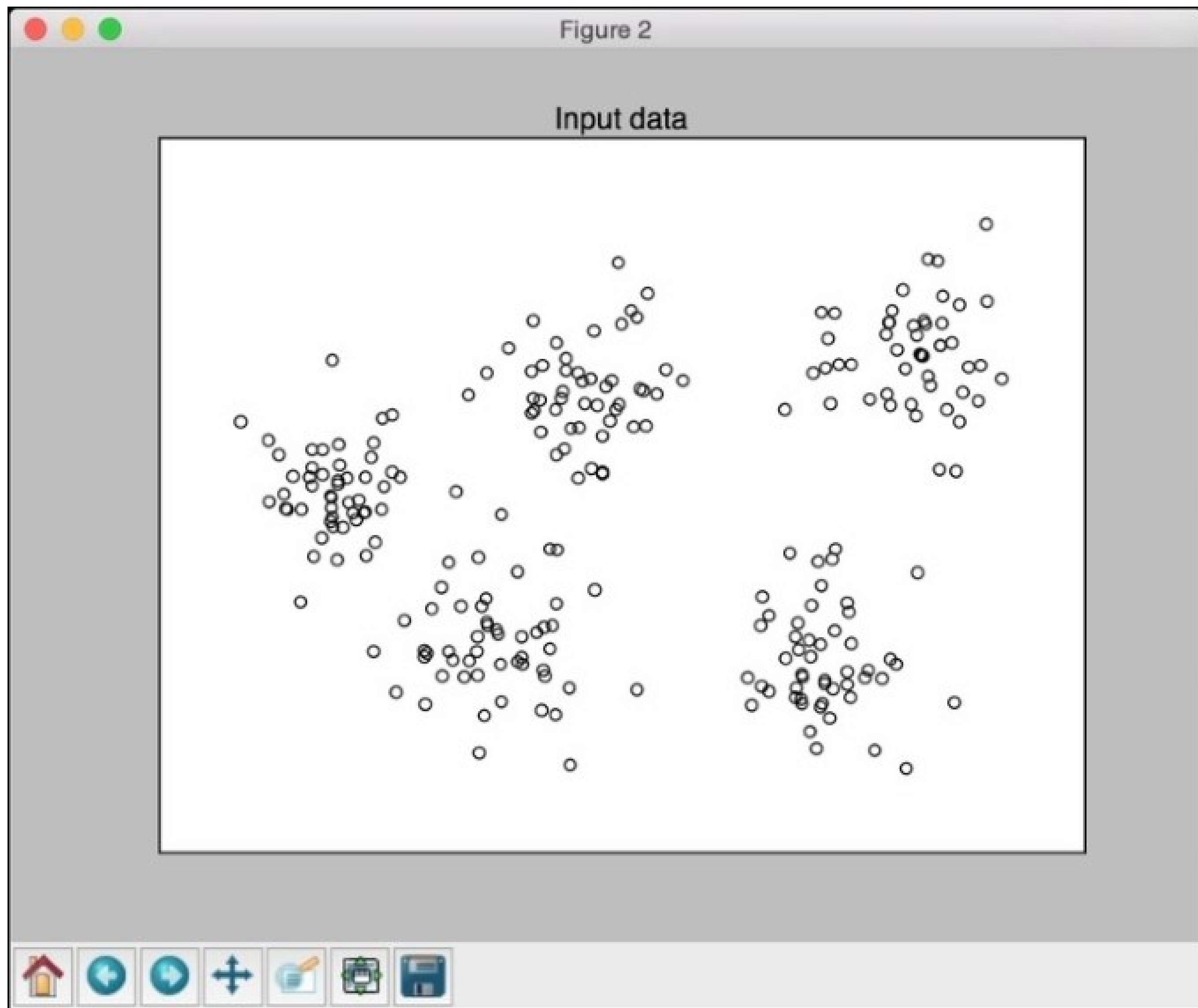
5. If you run this code, you will get the following output on the Terminal:

```
Number of clusters = 2
Silhouette score = 0.529039717547

Number of clusters = 3
Silhouette score = 0.557246639118

Number of clusters = 4
Silhouette score = 0.583275751783

Number of clusters = 5
Silhouette score = 0.658279690976

Number of clusters = 6
Silhouette score = 0.582358411948

Number of clusters = 7
Silhouette score = 0.528610740989

Number of clusters = 8
Silhouette score = 0.459759448983

Number of clusters = 9
Silhouette score = 0.415953573837
```

6. The bar graph looks like the following:



7. As per these scores, the best configuration is five clusters. Let's see what the data actually looks like:



We can visually confirm that the data in fact has five clusters. We just took the example of a small dataset that contains five distinct clusters. This method becomes very useful when you are dealing with a huge dataset that contains high-dimensional data that cannot be visualized easily.

Automatically estimating the number of clusters using DBSCAN algorithm

When we discussed the k-means algorithm, we saw that we had to give the number of clusters as one of the input parameters. In the real world, we wouldn't have this information available. We can definitely sweep the parameter space to find out the optimal number of clusters using the silhouette coefficient score, but this would be an expensive process! Wouldn't it be nice if there were a method that can just tell us the number of clusters in our data? This is where **Density-Based Spatial Clustering of Applications with Noise** (**DBSCAN**) comes into the picture.

This works by treating datapoints as groups of dense clusters. If a point belongs to a cluster, then there should be a lot of others points that belong to the same cluster. One of the parameters that we can control is the maximum distance of this point from other points. This is called **epsilon**. No two points in a given cluster should be further away than epsilon. You can learn more about it at http://staffwww.itn.liu.se/~aidvi/courses/06/dm/Seminars2011/DBSCAN(4).pdf. One of the main advantages of this method is that it can deal with outliers. If there are some points located alone in a low-density area, DBSCAN will detect these points as outliers as opposed to forcing them into a cluster.

How to do it…

1. The full code for this recipe is given in the `estimate_clusters.py` file that's already provided to you. Let's look at how it's built. Create a new Python file, and import the necessary packages:

```
from itertools import cycle

import numpy as np
from sklearn.cluster import DBSCAN
from sklearn import metrics
import matplotlib.pyplot as plt

from utilities import load_data
```

2. Load the input data from the `data_perf.txt` file. This is the same file that we used in the previous recipe, which will help us compare the methods on the same dataset:

```
# Load input data
input_file = 'data_perf.txt'
X = load_data(input_file)
```

3. We need to find the best parameter. Let's initialize a few variables:

```
# Find the best epsilon
eps_grid = np.linspace(0.3, 1.2, num=10)
silhouette_scores = []
eps_best = eps_grid[0]
silhouette_score_max = -1
```

```
model_best = None
labels_best = None
```

4. Let's sweep the parameter space:

```
for eps in eps_grid:
    # Train DBSCAN clustering model
    model = DBSCAN(eps=eps, min_samples=5).fit(X)

    # Extract labels
    labels = model.labels_
```

5. For each iteration, we need to extract the performance metric:

```
    # Extract performance metric
    silhouette_score = round(metrics.silhouette_score(X,
labels), 4)
    silhouette_scores.append(silhouette_score)

    print "Epsilon:", eps, " --> silhouette score:",
silhouette_score
```

6. We need to store the best score and its associated epsilon value:

```
    if silhouette_score > silhouette_score_max:
        silhouette_score_max = silhouette_score
        eps_best = eps
        model_best = model
        labels_best = labels
```

7. Let's plot the bar graph:

```
# Plot silhouette scores vs epsilon
plt.figure()
plt.bar(eps_grid, silhouette_scores, width=0.05, color='k',
align='center')
plt.title('Silhouette score vs epsilon')

# Best params
print "\nBest epsilon =", eps_best
```

8. Let's store the best models and labels:

```
# Associated model and labels for best epsilon
model = model_best
labels = labels_best
```

9. Some datapoints may remain unassigned. We need to identify them, as follows:

```
# Check for unassigned datapoints in the labels
offset = 0
if -1 in labels:
    offset = 1
```

10. Extract the number of clusters:

```
# Number of clusters in the data
num_clusters = len(set(labels)) - offset

print "\nEstimated number of clusters =", num_clusters
```

11. We need to extract all core samples:

```
# Extracts the core samples from the trained model
mask_core = np.zeros(labels.shape, dtype=np.bool)
mask_core[model.core_sample_indices_] = True
```

12. Let's visualize the resultant clusters. We will start by extracting the set of unique labels and specifying different markers:

```
# Plot resultant clusters
plt.figure()
labels_uniq = set(labels)
markers = cycle('vo^s<>')
```

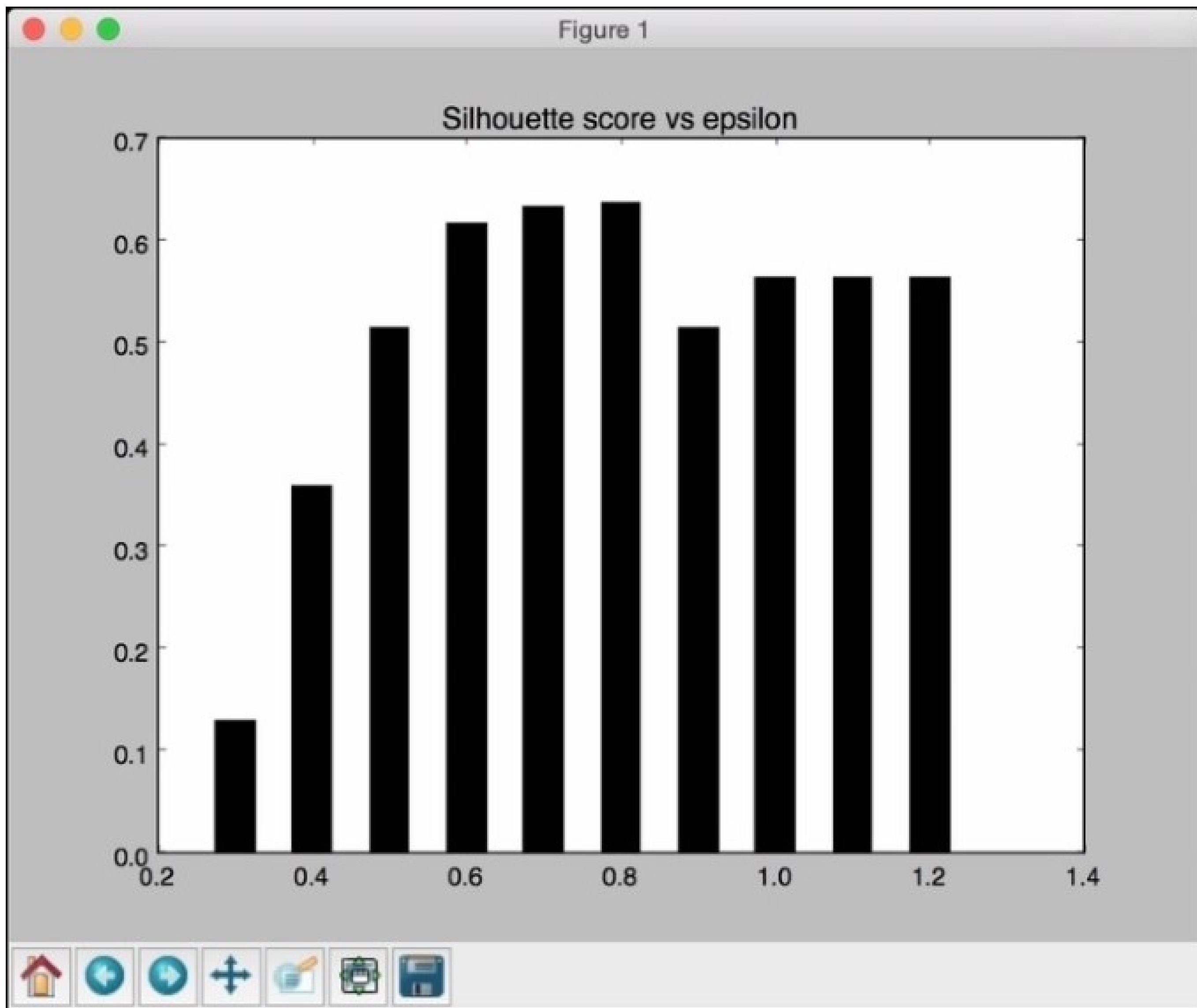
13. Let's iterate through the clusters and plot the datapoints using different markers:

```
for cur_label, marker in zip(labels_uniq, markers):
    # Use black dots for unassigned datapoints
    if cur_label == -1:
        marker = '.'

    # Create mask for the current label
    cur_mask = (labels == cur_label)

    cur_data = X[cur_mask & mask_core]
    plt.scatter(cur_data[:, 0], cur_data[:, 1], marker=marker,
             edgecolors='black', s=96, facecolors='none')

    cur_data = X[cur_mask & ~mask_core]
    plt.scatter(cur_data[:, 0], cur_data[:, 1], marker=marker,
             edgecolors='black', s=32)

plt.title('Data separated into clusters')
plt.show()
```

14. If you run this code, you will get the following on your Terminal:

```
Epsilon: 0.3  --> silhouette score: 0.1287
Epsilon: 0.4  --> silhouette score: 0.3594
Epsilon: 0.5  --> silhouette score: 0.5134
Epsilon: 0.6  --> silhouette score: 0.6165
Epsilon: 0.7  --> silhouette score: 0.6322
Epsilon: 0.8  --> silhouette score: 0.6366
Epsilon: 0.9  --> silhouette score: 0.5142
Epsilon: 1.0  --> silhouette score: 0.5629
Epsilon: 1.1  --> silhouette score: 0.5629
Epsilon: 1.2  --> silhouette score: 0.5629

Best epsilon = 0.8

Estimated number of clusters = 5
```

15. You will get the following bar graph:



16. Let's look the labeled datapoints along with unassigned datapoints marked by solid points in the following figure:

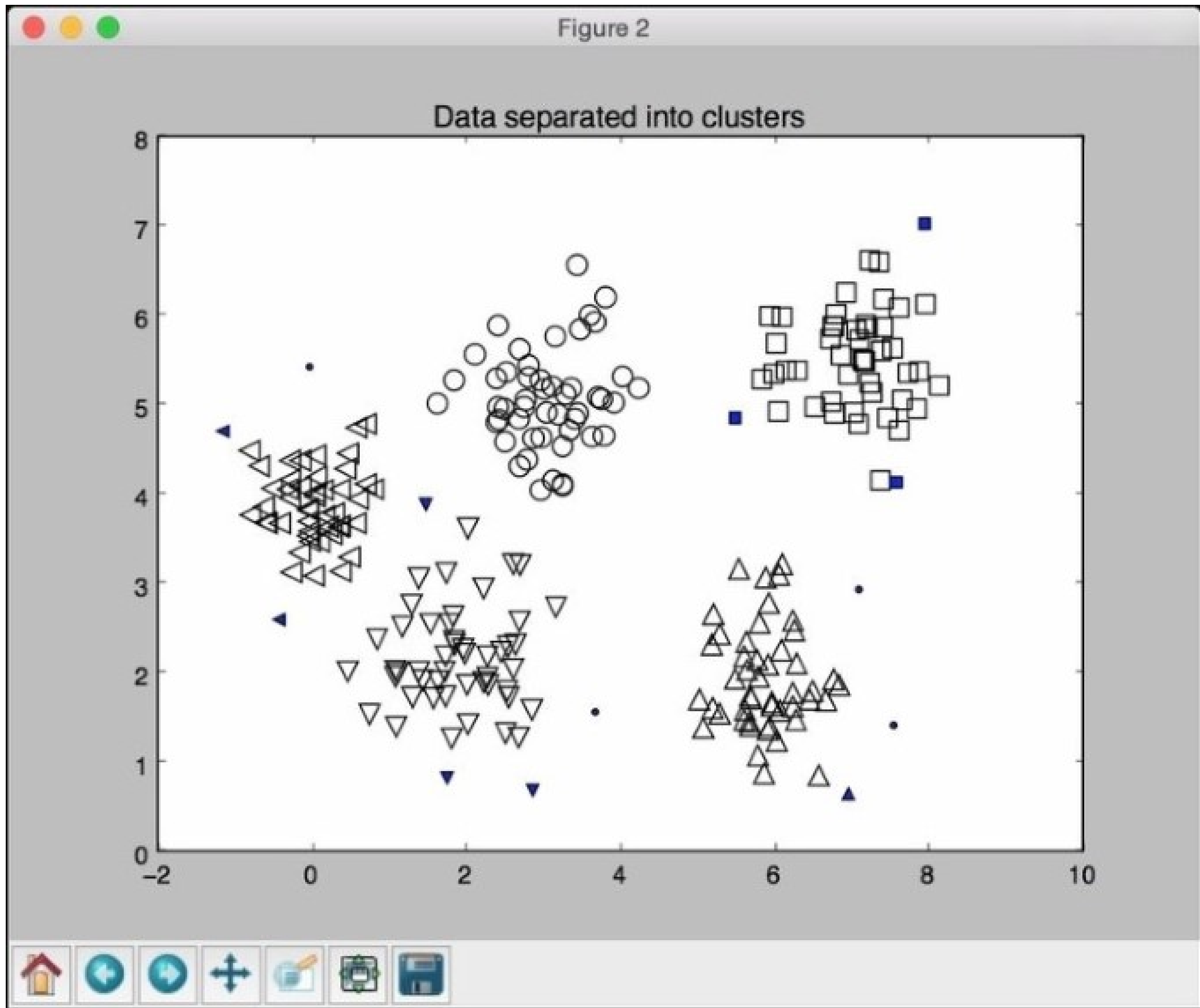


Figure 2
Data separated into clusters
8
7
6
5
4
3
2
1
0
−2
0
2
4
6
8
10

Finding patterns in stock market data

Let's see how we can use unsupervised learning for stock market analysis. We will operate with the assumption that we don't know how many clusters there are. As we don't know the number of clusters, we will use an algorithm called **Affinity Propagation** to cluster. It tries to find a representative datapoint for each cluster in our data. It tries to find measures of similarity between pairs of datapoints and considers all our datapoints as potential representatives, also called **exemplars**, of their respective clusters. You can learn more about it at http://www.cs.columbia.edu/~delbert/docs/DDueck-thesis_small.pdf

In this recipe, we will analyze the stock market variations of companies in a specified duration of time. Our goal is to then find out what companies behave similarly in terms of their quotes over time.

How to do it…

1. The full code for this recipe is given in the `stock_market.py` file that's already provided to you. Let's look at how it's built. Create a new Python file, and import the following packages:

```
import json
import datetime

import numpy as np
import matplotlib.pyplot as plt
from sklearn import covariance, cluster
from matplotlib.finance import quotes_historical_yahoo_ochl as
quotes_yahoo
```

2. We need a file that contains all the symbols and the associated names. This information is located in the `symbol_map.json` file provided to you. Let's load this, as follows:

```
# Input symbol file
symbol_file = 'symbol_map.json'
```

3. Let's read the data from the symbol map file:

```
# Load the symbol map
with open(symbol_file, 'r') as f:
    symbol_dict = json.loads(f.read())

symbols, names = np.array(list(symbol_dict.items())).T
```

4. Let's specify a time period for the purpose of this analysis. We will use these start and end dates to load the input data:

```
# Choose a time period
start_date = datetime.datetime(2004, 4, 5)
end_date = datetime.datetime(2007, 6, 2)
```

5. Let's read the input data:

```
quotes = [quotes_yahoo(symbol, start_date, end_date,
asobject=True)
                for symbol in symbols]
```

6. As we need some feature points for analysis, we will use the difference between the opening and closing quotes every day to analyze the data:

```
# Extract opening and closing quotes
opening_quotes = np.array([quote.open for quote in
quotes]).astype(np.float)
closing_quotes = np.array([quote.close for quote in
quotes]).astype(np.float)

# The daily fluctuations of the quotes
delta_quotes = closing_quotes - opening_quotes
```

7. Let's build a graph model:

```
# Build a graph model from the correlations
edge_model = covariance.GraphLassoCV()
```

8. We need to standardize the data before we use it:

```
# Standardize the data
X = delta_quotes.copy().T
X /= X.std(axis=0)
```

9. Let's train the model using this data:

```
# Train the model
with np.errstate(invalid='ignore'):
    edge_model.fit(X)
```

10. We are now ready to build the clustering model:

```
# Build clustering model using affinity propagation
_, labels = cluster.affinity_propagation(edge_model.covariance_)
num_labels = labels.max()

# Print the results of clustering
for i in range(num_labels + 1):
    print "Cluster", i+1, "-->", ', '.join(names[labels == i])
```

11. If you run this code, you will get the following output on the Terminal:

```
Cluster 1 --> ConocoPhillips, Chevron, Total, Valero Energy, Exxon
Cluster 2 --> CVS, Walgreen
Cluster 3 --> IBM, Cisco, Microsoft, Texas instruments, Ford, HP, Dell
Cluster 4 --> Cablevision
Cluster 5 --> Pfizer, Apple, Caterpillar, Canon, Boeing, Toyota, SAP, Honda, Mitsubishi, Sony, Mc Donalds
, Unilever, Wal-Mart
Cluster 6 --> Kimberly-Clark, Colgate-Palmolive, Procter Gamble
Cluster 7 --> Yahoo, Amazon
Cluster 8 --> American express, Wells Fargo, Navistar, Bank of America, Time Warner, Ryder, Kellogg, Home
 Depot, AIG, Goldman Sachs, General Electrics, Marriott, Xerox, JPMorgan Chase, DuPont de Nemours, 3M, Co
mcast
Cluster 9 --> GlaxoSmithKline, Novartis, Sanofi-Aventis
Cluster 10 --> Pepsi, Coca Cola
Cluster 11 --> Raytheon, Lookheed Martin, General Dynamics, Northrop Grumman
Cluster 12 --> Kraft Foods
```

Building a customer segmentation model

One of the main applications of unsupervised learning is market segmentation. This is when we don't have labeled data available all the time, but it's important to segment the market so that people can target individual groups. This is very useful in advertising, inventory management, implementing strategies for distribution, mass media, and so on. Let's go ahead and apply unsupervised learning to one such case to see how it can be useful.

We will be dealing with a wholesale vendor and his customers. We will be using the data available at https://archive.ics.uci.edu/ml/datasets/Wholesale+customers. The spreadsheet contains data regarding the consumption of different types of items by their customers and our goal is to find clusters so that they can optimize their sales and distribution strategy.

How to do it…

1. The full code for this recipe is given in the `customer_segmentation.py` file that's already provided to you. Let's look at how it's built. Create a new Python file, and import the following packages:

```
import csv

import numpy as np
from sklearn import cluster, covariance, manifold
from sklearn.cluster import MeanShift, estimate_bandwidth
import matplotlib.pyplot as plt
```

2. Let's load the input data from the `wholesale.csv` file that's already provided to you:

```
# Load data from input file
input_file = 'wholesale.csv'
file_reader = csv.reader(open(input_file, 'rb'), delimiter=',')
X = []
for count, row in enumerate(file_reader):
    if not count:
        names = row[2:]
        continue

    X.append([float(x) for x in row[2:]])

# Input data as numpy array
X = np.array(X)
```

3. Let's build a Mean Shift model like we did in one of the earlier recipes:

```
# Estimating the bandwidth
bandwidth = estimate_bandwidth(X, quantile=0.8,
n_samples=len(X))
```

```
# Compute clustering with MeanShift
meanshift_estimator = MeanShift(bandwidth=bandwidth,
bin_seeding=True)
meanshift_estimator.fit(X)
labels = meanshift_estimator.labels_
centroids = meanshift_estimator.cluster_centers_
num_clusters = len(np.unique(labels))

print "\nNumber of clusters in input data =", num_clusters
```

4. Let's print the centroids of clusters that we obtained, as follows:

```
print "\nCentroids of clusters:"
print '\t'.join([name[:3] for name in names])
for centroid in centroids:
    print '\t'.join([str(int(x)) for x in centroid])
```

5. Let's visualize a couple of features to get a sense of the output:

```
# Visualizing data

centroids_milk_groceries = centroids[:, 1:3]

# Plot the nodes using the coordinates of our
centroids_milk_groceries
plt.figure()
plt.scatter(centroids_milk_groceries[:,0],
centroids_milk_groceries[:,1],
        s=100, edgecolors='k', facecolors='none')

offset = 0.2
plt.xlim(centroids_milk_groceries[:,0].min() - offset *
centroids_milk_groceries[:,0].ptp(),
        centroids_milk_groceries[:,0].max() + offset *
centroids_milk_groceries[:,0].ptp(),)
plt.ylim(centroids_milk_groceries[:,1].min() - offset *
centroids_milk_groceries[:,1].ptp(),
        centroids_milk_groceries[:,1].max() + offset *
centroids_milk_groceries[:,1].ptp())

plt.title('Centroids of clusters for milk and groceries')
plt.show()
```

6. If you run this code, you will get the following output on the Terminal:

```
Number of clusters in input data = 8

Centroids of clusters:
Fre     Mil     Gro     Fro     Det     Del
9632    4671    6593    2570    2296    1248
40204   46314   57584   5518    25436   4241
8565    4980    67298   131     38102   1215
32717   16784   13626   60869   1272    5609
22925   73498   32114   987     20070   903
112151  29627   18148   16745   4948    8550
16117   46197   92780   1026    40827   2944
36847   43950   20170   36534   239     47943
```

7. You will get the following image that depicts the centroids for the features *milk* and *groceries*, where milk is on the X-axis and groceries is on the Y-axis:

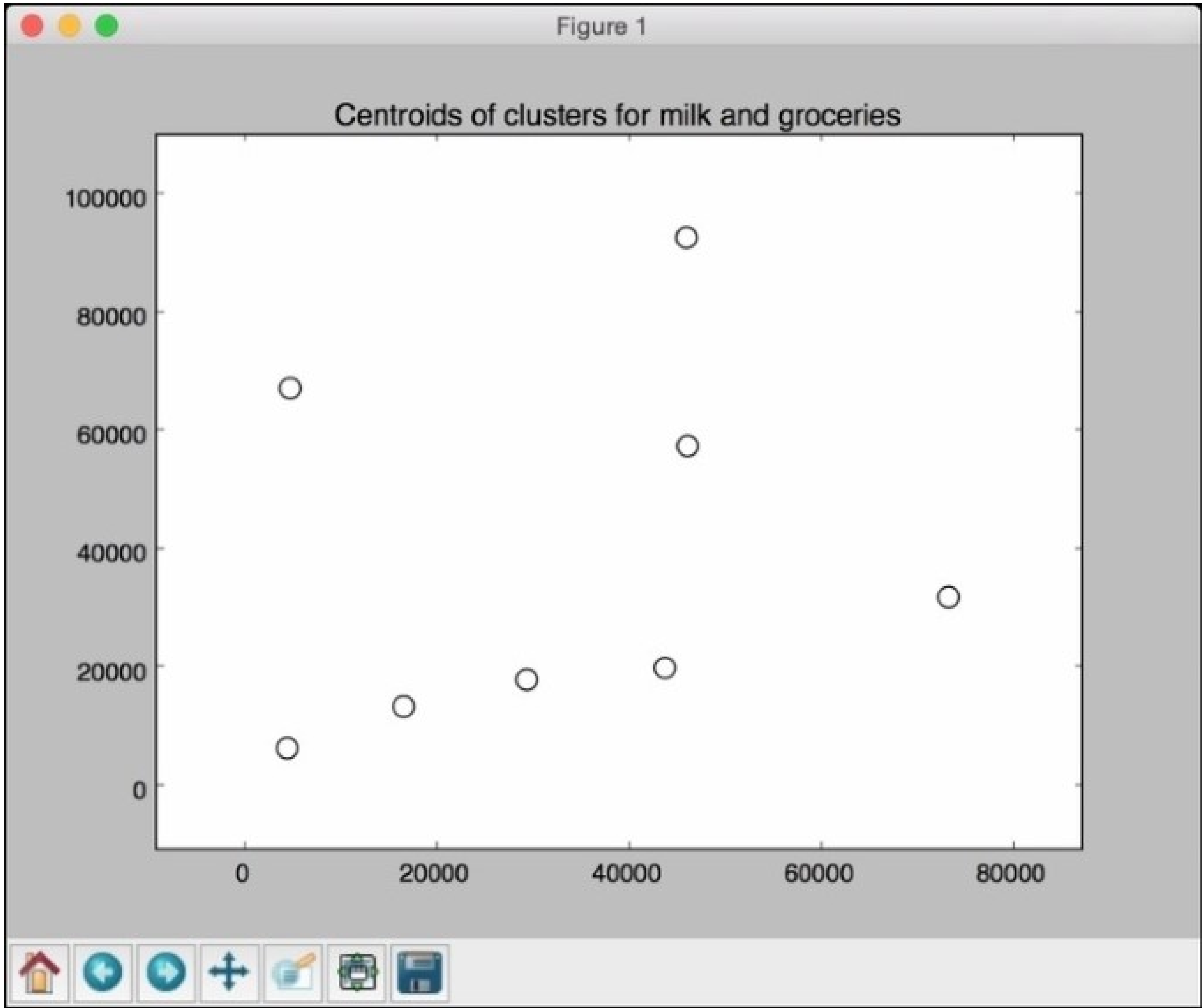


Figure 1
Centroids of clusters for milk and groceries
100000
80000
60000
40000
20000
0
0
20000
40000
60000
80000

Chapter 5. Building Recommendation Engines

In this chapter, we will cover the following recipes:

- Building function compositions for data processing
- Building machine learning pipelines
- Finding the nearest neighbors
- Constructing a k-nearest neighbors classifier
- Constructing a k-nearest neighbors regressor
- Computing the Euclidean distance score
- Computing the Pearson correlation score
- Finding similar users in the dataset
- Generating movie recommendations

Introduction

A recommendation engine is a model that can predict what a user may be interested in. When we apply this to the context of movies, this becomes a movie-recommendation engine. We filter items in our database by predicting how the current user might rate them. This helps us in connecting the users with the right content in our dataset. Why is this relevant? If you have a massive catalog, then the users may or may not find all the relevant content. By recommending the right content, you increase consumption. Companies such as Netflix heavily rely on recommendations to keep the user engaged.

Recommendation engines usually produce a set of recommendations using either collaborative filtering or content-based filtering. The difference between the two approaches is in the way the recommendations are mined. Collaborative filtering builds a model from the past behavior of the current user as well as ratings given by other users. We then use this model to predict what this user might be interested in. Content-based filtering, on the other hand, uses the characteristics of the item itself in order to recommend more items to the user. The similarity between items is the main driving force here. In this chapter, we will focus on collaborative filtering.

Building function compositions for data processing

One of the major parts of any machine learning system is the data processing pipeline. Before data is fed into the machine learning algorithm for training, we need to process it in different ways to make it suitable for that algorithm. Having a robust data processing pipeline goes a long way in building an accurate and scalable machine learning system. There are a lot of basic functionalities available, and data processing pipelines usually consist of a combination of these. Instead of calling these functions in a nested or loopy way, it's better to use the functional programming paradigm to build the combination. Let's take a look at how to combine these functions to form a reusable function composition. In this recipe, we will create three basic functions and look at how to compose a pipeline.

How to do it…

1. Create a new Python file, and add the following line:

```
import numpy as np
```

2. Let's define a function to add `3` to each element of the array:

```
def add3(input_array):
    return map(lambda x: x+3, input_array)
```

3. Let's define a second function to multiply `2` with each element of the array:

```
def mul2(input_array):
    return map(lambda x: x*2, input_array)
```

4. Let's define a third function to subtract `5` from each element of the array:

```
def sub5(input_array):
    return map(lambda x: x-5, input_array)
```

5. Let's define a function composer that takes functions as input arguments and returns a composed function. This composed function is basically a function that applies all the input functions in sequence:

```
def function_composer(*args):
    return reduce(lambda f, g: lambda x: f(g(x)), args)
```

We use the `reduce` function to combine all the input functions by successively applying the functions in sequence.

6. We are now ready to play with this function composer. Let's define some data and a sequence of operations:

```
if __name__=='__main__':
    arr = np.array([2,5,4,7])
```

```
print "\nOperation: add3(mul2(sub5(arr)))"
```

7. If we were to use the regular method, we apply this successively, as follows:

```
arr1 = add3(arr)
arr2 = mul2(arr1)
arr3 = sub5(arr2)
print "Output using the lengthy way:", arr3
```

8. Let's use the function composer to achieve the same thing in a single line:

```
    func_composed = function_composer(sub5, mul2, add3)
    print "Output using function composition:",
func_composed(arr)
```

9. We can do the same thing in a single line with the previous method as well, but the notation becomes really nested and unreadable. Also, this is not reusable! You have to write the whole thing again if you want to reuse this sequence of operations:

```
    print "\nOperation:
sub5(add3(mul2(sub5(mul2(arr)))))\nOutput:", \
            function_composer(mul2, sub5, mul2, add3, sub5)(arr)
```

10. If you run this code, you will get the following output on the Terminal:

```
Operation: add3(mul2(sub5(arr)))
Output using the lengthy way: [5, 11, 9, 15]
Output using function composition: [5, 11, 9, 15]

Operation: sub5(add3(mul2(sub5(mul2(arr)))))
Output: [-10, 2, -2, 10]
```

Building machine learning pipelines

The scikit-learn library has provisions to build machine learning pipelines. We just need to specify the functions, and it will build a composed object that makes the data go through the whole pipeline. This pipeline can include functions, such as preprocessing, feature selection, supervised learning, unsupervised learning, and so on. In this recipe, we will be building a pipeline to take the input feature vector, select the top *k* features, and then classify them using a random forest classifier.

How to do it…

1. Create a new Python file, and import the following packages:

   ```
   from sklearn.datasets import samples_generator
   from sklearn.ensemble import RandomForestClassifier
   from sklearn.feature_selection import SelectKBest, f_regression
   from sklearn.pipeline import Pipeline
   ```

2. Let's generate some sample data to play with:

   ```
   # generate sample data
   X, y = samples_generator.make_classification(
           n_informative=4, n_features=20, n_redundant=0,
   random_state=5)
   ```

   This line generated `20` dimensional feature vectors because this is the default value. You can change it using the `n_features` parameter in the previous line.
3. Our first step of the pipeline is to select the *k* best features and before the datapoint is used further. In this case, let's set `k` to `10`:

   ```
   # Feature selector
   selector_k_best = SelectKBest(f_regression, k=10)
   ```

4. The next step is to use a random forest classifier to classify the data:

   ```
   # Random forest classifier
   classifier = RandomForestClassifier(n_estimators=50,
   max_depth=4)
   ```

5. We are now ready to build the pipeline. The pipeline method allows us to use predefined objects to build the pipeline:

   ```
   # Build the machine learning pipeline
   pipeline_classifier = Pipeline([('selector', selector_k_best),
   ('rf', classifier)])
   ```

   We can also assign names to the blocks in our pipeline. In the preceding line, we assign the `selector` name to our feature selector and `rf` to our random forest classifier. You are free to use any other random names here!

6. We can also update these parameters as we go along. We can set the parameters using the names that we assigned in the previous step. For example, if we want to set `k` to `6` in the feature selector and set `n_estimators` to `25` in the random forest classifier, we can do it like in the following code. Note that these are the variable names given in the previous step:

```
pipeline_classifier.set_params(selector__k=6,
        rf__n_estimators=25)
```

7. Let's go ahead and train the classifier:

```
# Training the classifier
pipeline_classifier.fit(X, y)
```

8. Let's predict the outputs for the training data:

```
# Predict the output
prediction = pipeline_classifier.predict(X)
print "\nPredictions:\n", prediction
```

9. Let's estimate the performance of this classifier:

```
# Print score
print "\nScore:", pipeline_classifier.score(X,
y)
```

10. We can also see which features get selected. Let's go ahead and print them:

```
# Print the selected features chosen by the selector
features_status =
pipeline_classifier.named_steps['selector'].get_support()
selected_features = []
for count, item in enumerate(features_status):
    if item:
        selected_features.append(count)

print "\nSelected features (0-indexed):", ', '.join([str(x) for
x in selected_features])
```

11. If you run this code, you will get the following output on your Terminal:

```
Predictions:
[0 0 0 1 1 0 1 1 0 1 0 0 0 0 1 0 1 1 1 0 0 1 0 1 1 1 0 0 1 0 1 0 1 0 0 0 0
 0 0 0 0 0 0 1 0 1 0 1 1 1 1 1 1 0 0 1 0 0 1 0 0 1 1 1 1 0 1 1 0 1 1 0 0 1
 0 0 1 1 1 1 0 0 0 1 1 1 1 0 1 0 0 0 0 1 0 0 1 1 0 1]

Score: 0.96

Selected features (0-indexed): 4, 8, 11, 15, 17, 19
```

How it works…

The advantage of selecting the *k* best features is that we will be able to work with low-dimensional data. This is helpful in reducing the computational complexity. The way in which we select the *k* best features is based on univariate feature selection. This performs univariate statistical tests and then extracts the top performing features from the feature vector. Univariate statistical tests refer to analysis techniques where a single variable is involved.

Once these tests are performed, each feature in the feature vector is assigned a score. Based on these scores, we select the top *k* features. We do this as a preprocessing step in our classifier pipeline. Once we extract the top *k* features, a *k*-dimensional feature vector is formed, and we use it as the input training data for the random forest classifier.

Finding the nearest neighbors

Nearest neighbors model refers to a general class of algorithms that aim to make a decision based on the number of nearest neighbors in the training dataset. Let's see how to find the nearest neighbors.

How to do it…

1. Create a new Python file, and import the following packages:

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.neighbors import NearestNeighbors
```

2. Let's create some sample two-dimensional data:

```
# Input data
X = np.array([[1, 1], [1, 3], [2, 2], [2.5, 5], [3, 1],
        [4, 2], [2, 3.5], [3, 3], [3.5, 4]])
```

3. Our goal is to find the three closest neighbors to any given point. Let's define this parameter:

```
# Number of neighbors we want to find
num_neighbors = 3
```

4. Let's define a random datapoint that's not present in the input data:

```
# Input point
input_point = [2.6, 1.7]
```

5. We need to see what this data looks like. Let's plot it, as follows:

```
# Plot datapoints
plt.figure()
plt.scatter(X[:,0], X[:,1], marker='o', s=25, color='k')
```

6. In order to find the nearest neighbors, we need to define the `NearestNeighbors` object with the right parameters and train it on the input data:

```
# Build nearest neighbors model
knn = NearestNeighbors(n_neighbors=num_neighbors,
algorithm='ball_tree').fit(X)
```

7. We can now find the distances of the input point to all the points in the input data:

```
distances, indices = knn.kneighbors(input_point)
```

8. We can print the k-nearest neighbors, as follows:

```
# Print the 'k' nearest neighbors
print "\nk nearest neighbors"
```

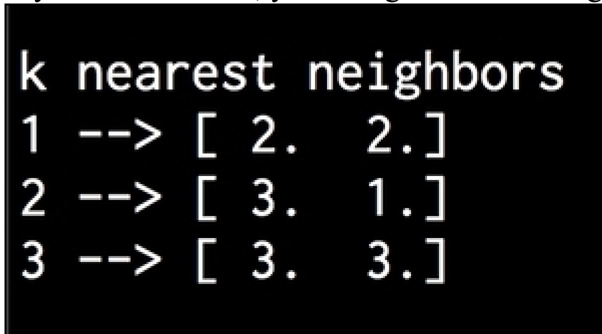
```
for rank, index in enumerate(indices[0][:num_neighbors]):
    print str(rank+1) + " -->", X[index]
```

The `indices` array is already sorted, so we just need to parse it and print the datapoints.

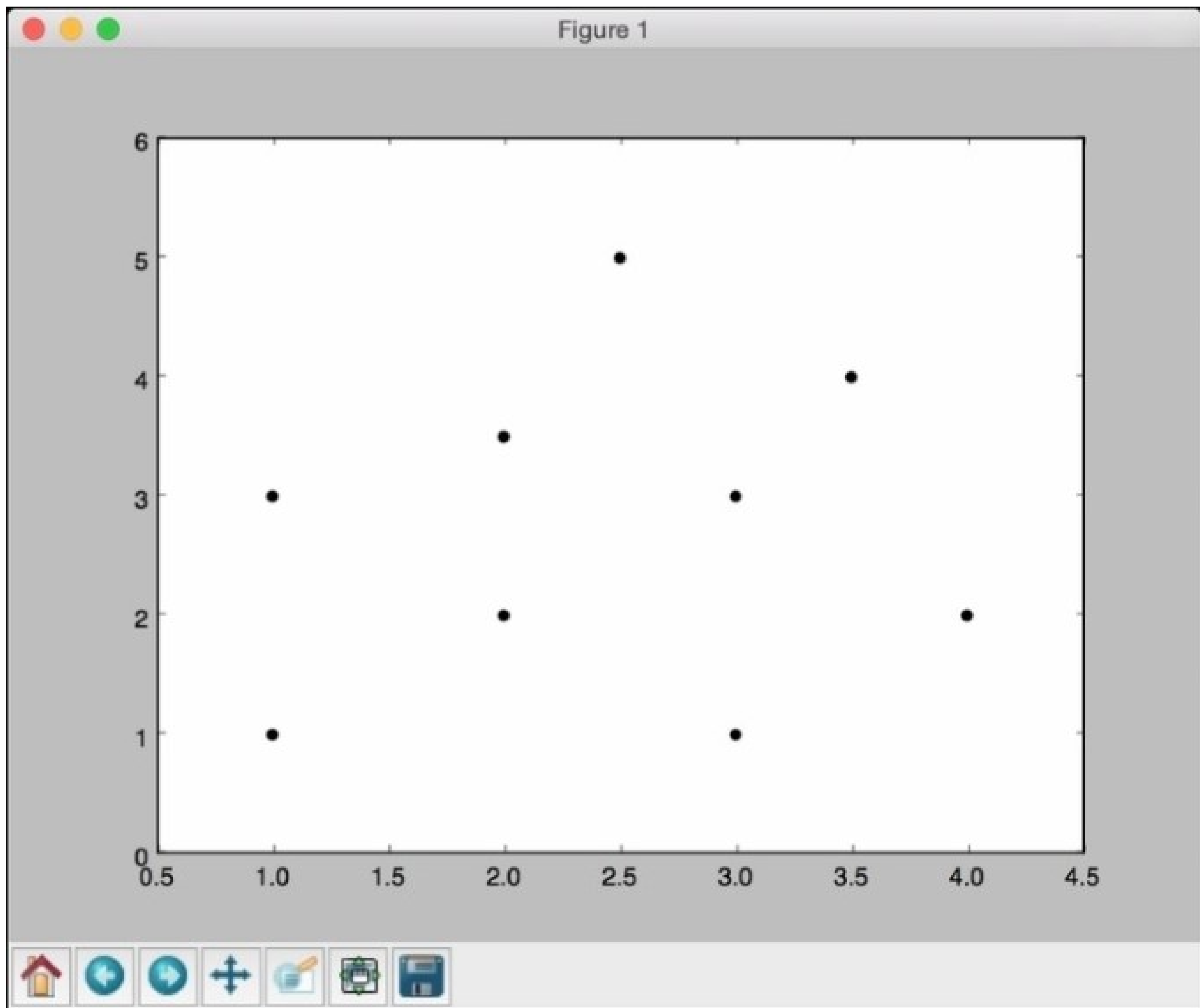
9. Let's plot the input datapoint and highlight the `k`-nearest neighbors:

```
# Plot the nearest neighbors
plt.figure()
plt.scatter(X[:,0], X[:,1], marker='o', s=25, color='k')
plt.scatter(X[indices][0][:][:,0], X[indices][0][:][:,1],
        marker='o', s=150, color='k', facecolors='none')
plt.scatter(input_point[0], input_point[1],
        marker='x', s=150, color='k', facecolors='none')

plt.show()
```

10. If you run this code, you will get the following output on your Terminal:



11. Here is the plot of the input datapoints:



12. The second output figure depicts the location of the test datapoint and the three nearest neighbors, as shown in the following screenshot:

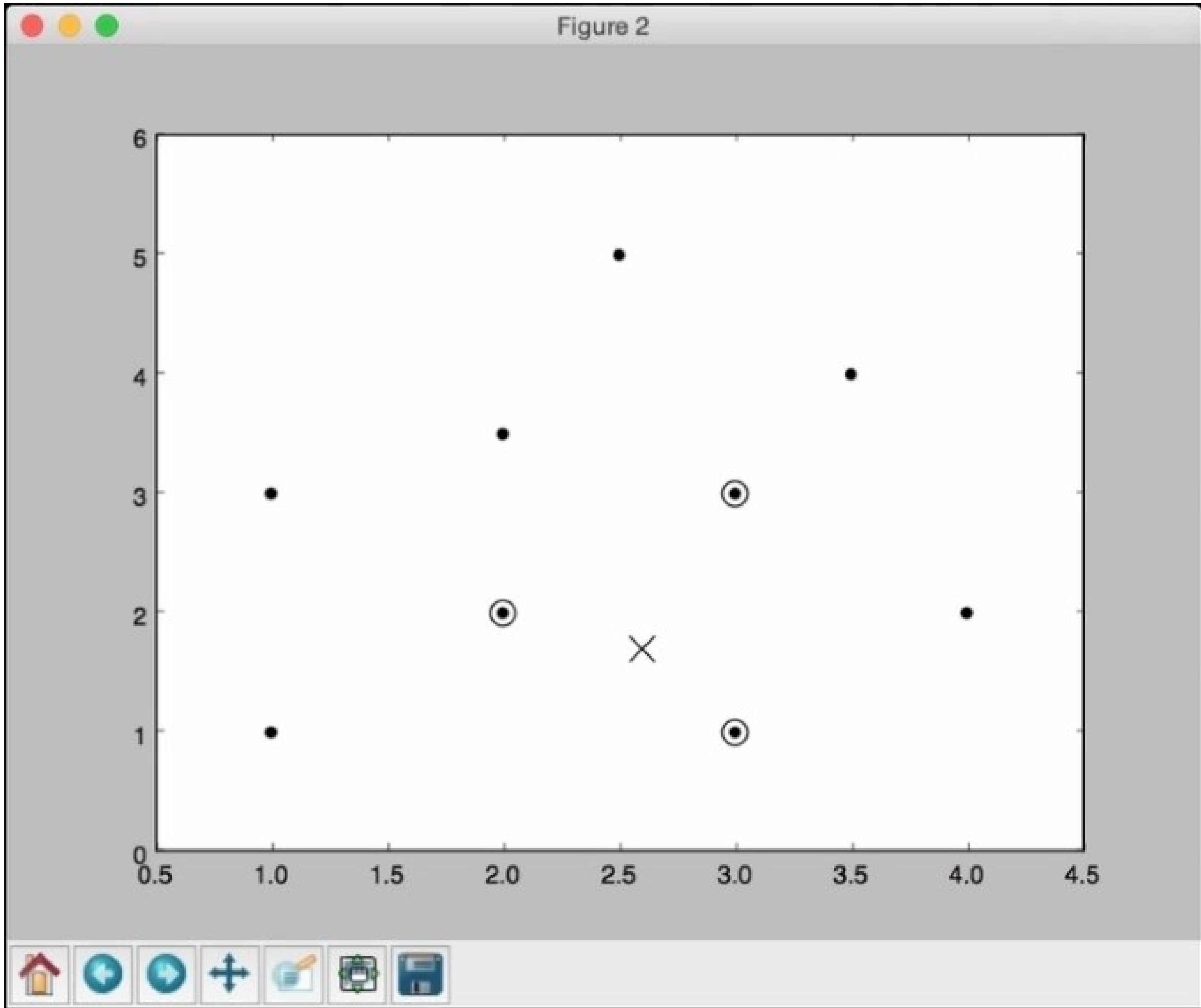


Figure 2
6
5
4
3
2
1
0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0
4.5

Constructing a k-nearest neighbors classifier

The **k-nearest neighbors** is an algorithm that uses *k*-nearest neighbors in the training dataset to find the category of an unknown object. When we want to find the class to which an unknown point belongs to, we find the k-nearest neighbors and take a majority vote. Let's take a look at how to construct this.

How to do it…

1. Create a new Python file, and import the following packages:

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.cm as cm
from sklearn import neighbors, datasets

from utilities import load_data
```

2. We will use the `data_nn_classifier.txt` file for input data. Let's load this input data:

```
# Load input data
input_file = 'data_nn_classifier.txt'
data = load_data(input_file)
X, y = data[:,:-1], data[:,-1].astype(np.int)
```

The first two columns contain input data and the last column contains the labels. Hence, we separated them into `X` and `y`, as shown in the preceding code.

3. Let's visualize the input data:

```
# Plot input data
plt.figure()
plt.title('Input datapoints')
markers = '^sov<>hp'
mapper = np.array([markers[i] for i in y])
for i in range(X.shape[0]):
    plt.scatter(X[i, 0], X[i, 1], marker=mapper[i],
            s=50, edgecolors='black', facecolors='none')
```

We iterate through all the datapoints and use the appropriate markers to separate the classes.

4. In order to build the classifier, we need to specify the number of nearest neighbors that we want to consider. Let's define this parameter:

```
# Number of nearest neighbors to consider
num_neighbors = 10
```

5. In order to visualize the boundaries, we need to define a grid and evaluate the classifier on that grid. Let's define the step size:

```
# step size of the grid
h = 0.01
```

6. We are now ready to build the k-nearest neighbors classifier. Let's define this and train it:

```
# Create a K-Neighbours Classifier model and train it
classifier = neighbors.KNeighborsClassifier(num_neighbors,
weights='distance')
classifier.fit(X, y)
```

7. We need to create a mesh to plot the boundaries. Let's define this, as follows:

```
# Create the mesh to plot the boundaries
x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
x_grid, y_grid = np.meshgrid(np.arange(x_min, x_max, h),
np.arange(y_min, y_max, h))
```

8. Let's evaluate the classifier output for all the points:

```
# Compute the outputs for all the points on the mesh
predicted_values = classifier.predict(np.c_[x_grid.ravel(),
y_grid.ravel()])
```

9. Let's plot it, as follows:

```
# Put the computed results on the map
predicted_values = predicted_values.reshape(x_grid.shape)
plt.figure()
plt.pcolormesh(x_grid, y_grid, predicted_values,
cmap=cm.Pastel1)
```

10. Now that we plotted the color mesh, let's overlay the training datapoints to see where they lie in relation to the boundaries:

```
# Overlay the training points on the map
for i in range(X.shape[0]):
    plt.scatter(X[i, 0], X[i, 1], marker=mapper[i],
            s=50, edgecolors='black', facecolors='none')

plt.xlim(x_grid.min(), x_grid.max())
plt.ylim(y_grid.min(), y_grid.max())
plt.title('k nearest neighbors classifier boundaries')
```

11. Now, we can consider a test datapoint and see whether the classifier performs correctly. Let's define it and plot it:

```
# Test input datapoint
test_datapoint = [4.5, 3.6]
plt.figure()
```

```
plt.title('Test datapoint')
for i in range(X.shape[0]):
    plt.scatter(X[i, 0], X[i, 1], marker=mapper[i],
            s=50, edgecolors='black', facecolors='none')

plt.scatter(test_datapoint[0], test_datapoint[1], marker='x',
        linewidth=3, s=200, facecolors='black')
```

12. We need to extract the k-nearest neighbors using the following model:

```
# Extract k nearest neighbors
dist, indices = classifier.kneighbors(test_datapoint)
```

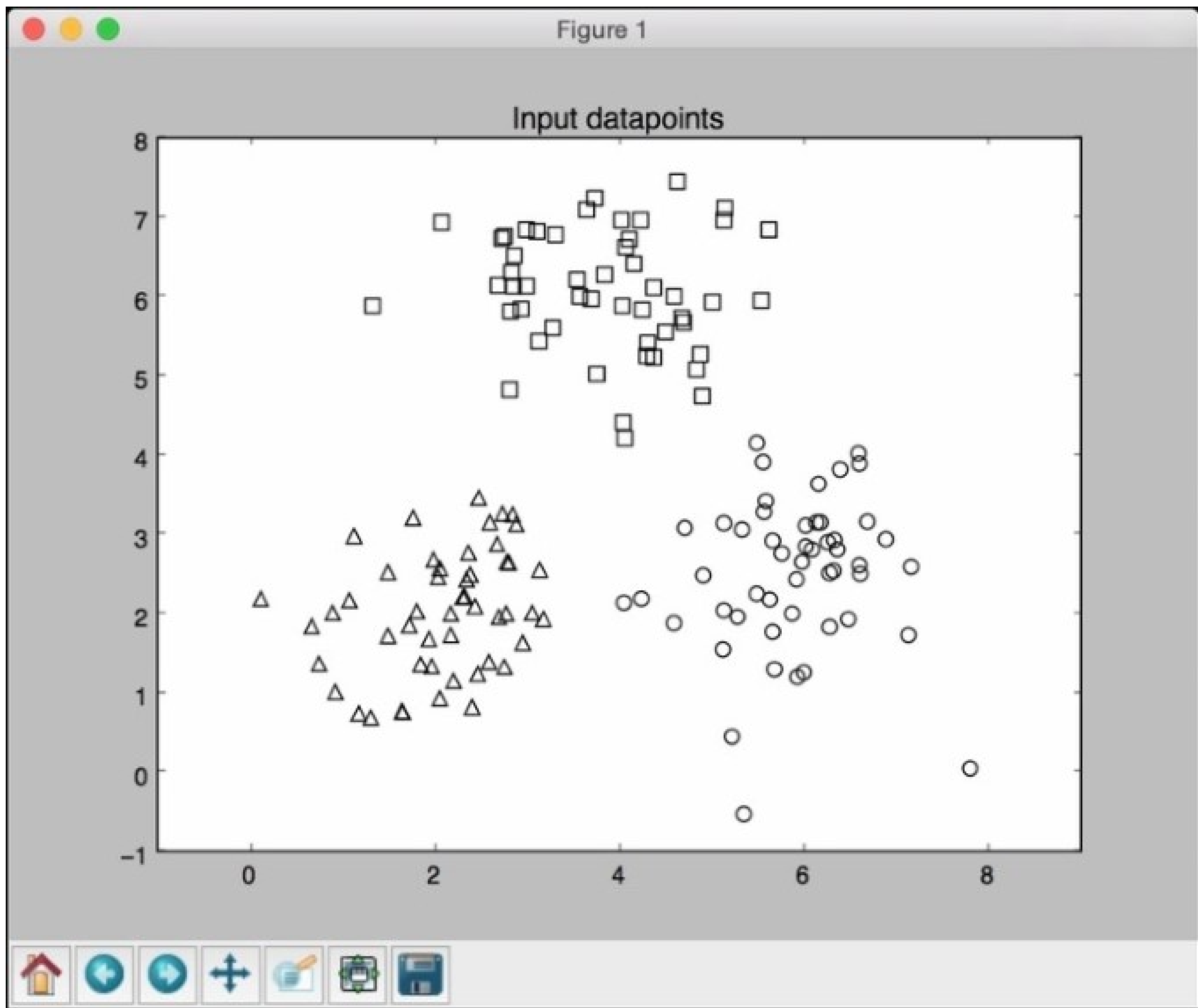
13. Let's plot the k-nearest neighbors and highlight them:

```
# Plot k nearest neighbors
plt.figure()
plt.title('k nearest neighbors')

for i in indices:
    plt.scatter(X[i, 0], X[i, 1], marker='o',
            linewidth=3, s=100, facecolors='black')

plt.scatter(test_datapoint[0], test_datapoint[1], marker='x',
        linewidth=3, s=200, facecolors='black')

for i in range(X.shape[0]):
    plt.scatter(X[i, 0], X[i, 1], marker=mapper[i],
            s=50, edgecolors='black', facecolors='none')

plt.show()
```

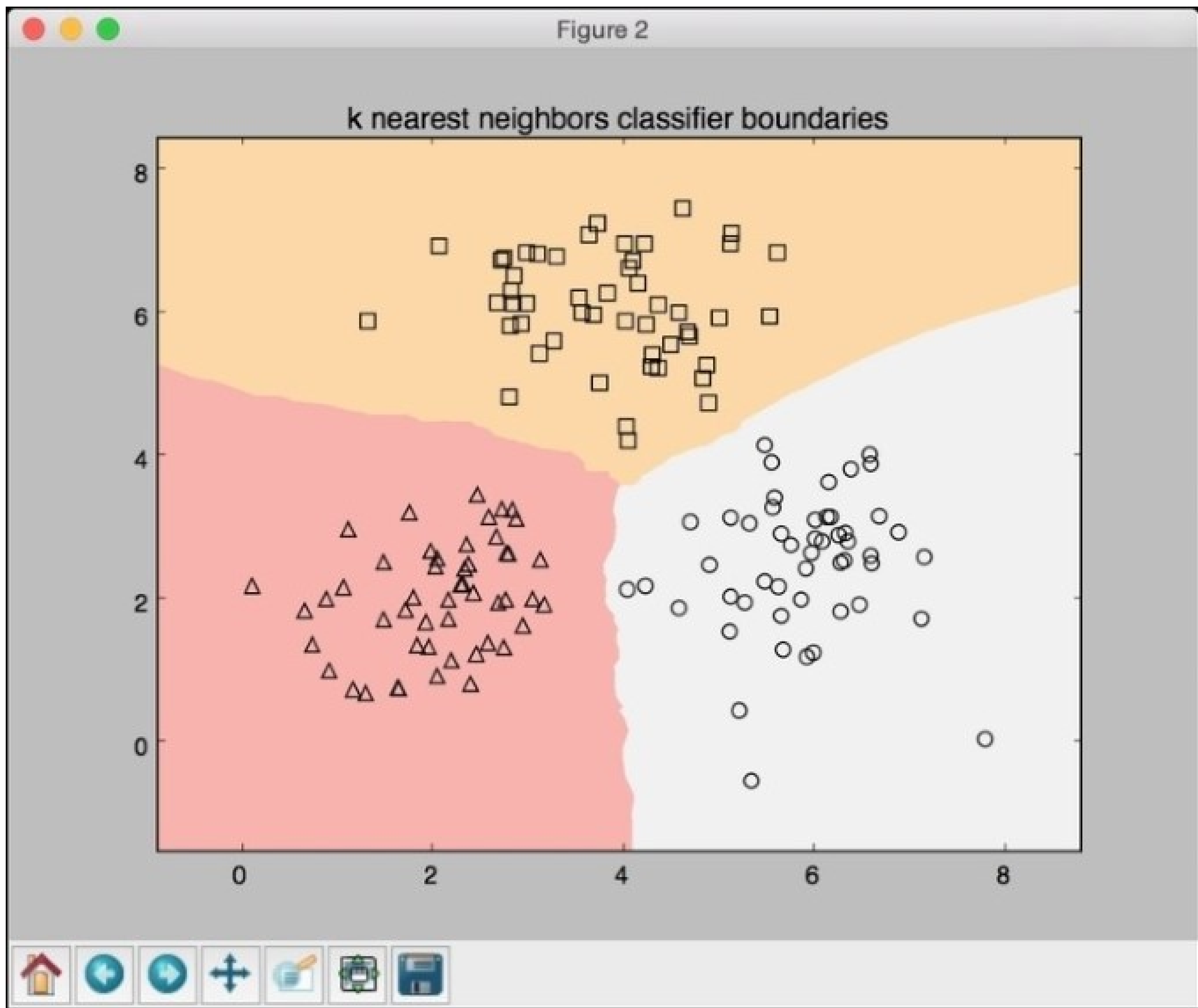
14. Let's print the classifier output on the Terminal:

```
print "Predicted output:", classifier.predict(test_datapoint)[0]
```

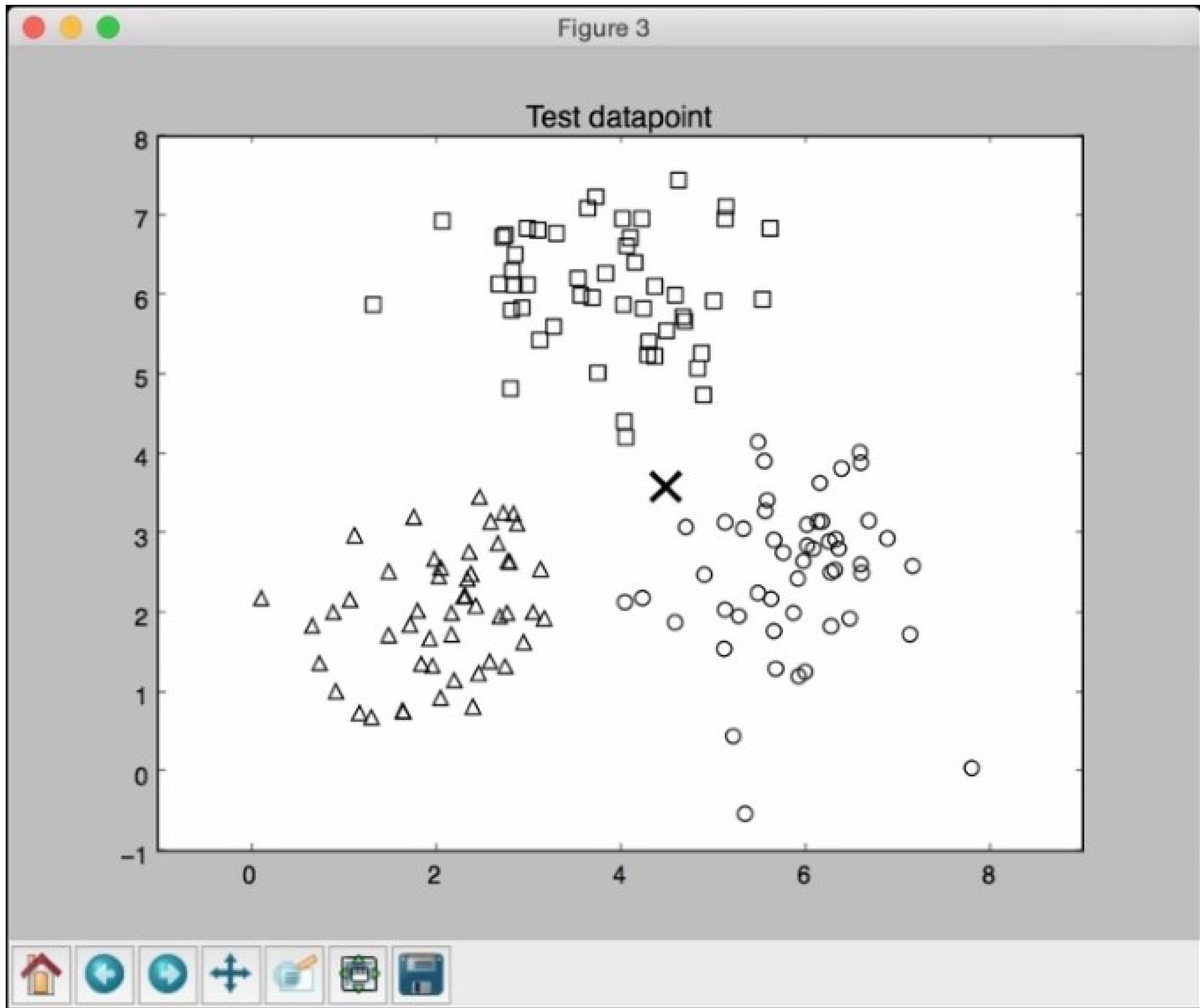
15. If you run this code, the first output figure depicts the distribution of the input datapoints:



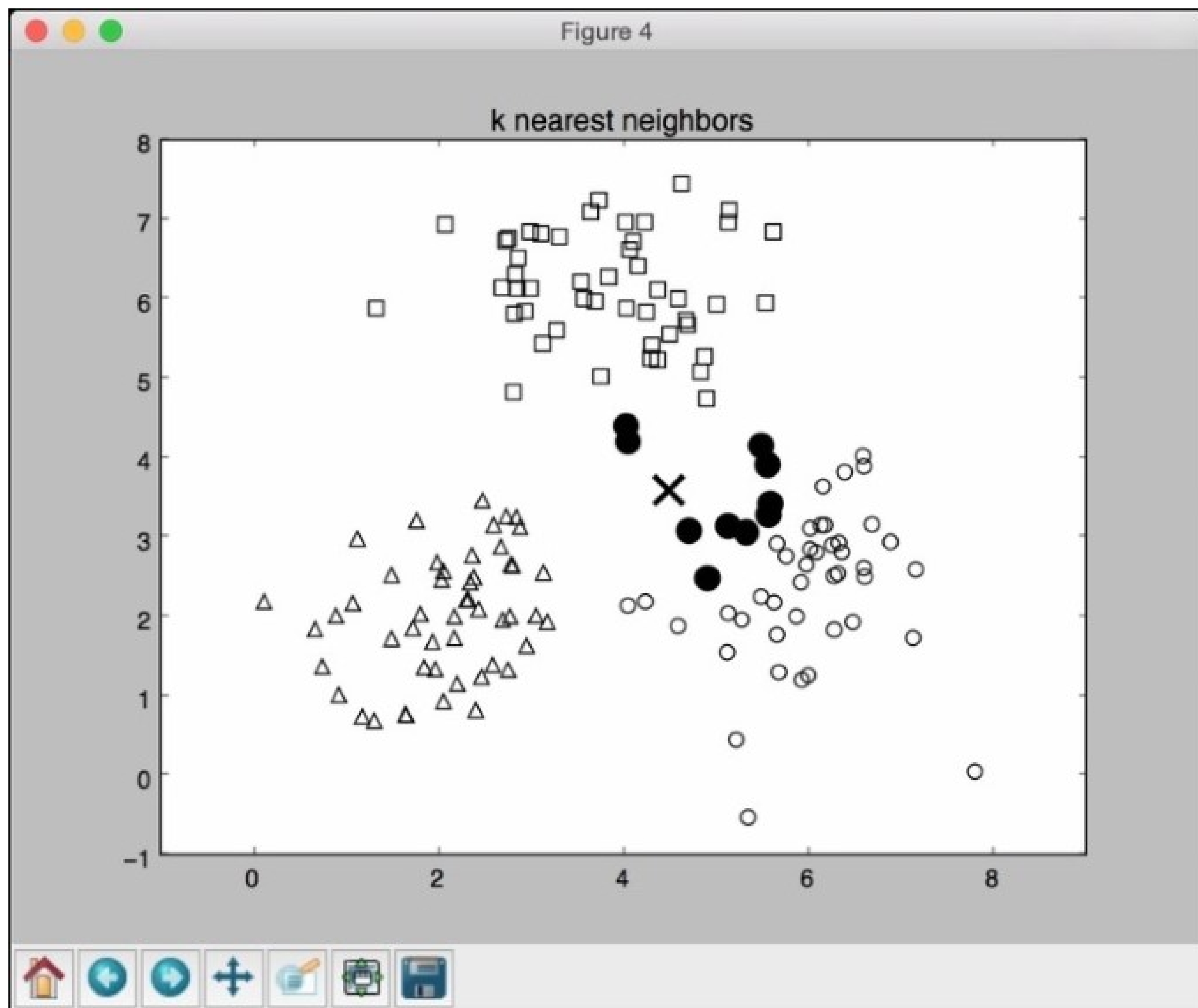
16. The second output figure depicts the boundaries obtained using the k-nearest neighbors classifier:



17. The third output figure depicts the location of the test datapoint:



18. The fourth output figure depicts the location of the 10 nearest neighbors:



How it works…

The k-nearest neighbors classifier stores all the available datapoints and classifies new datapoints based on a similarity metric. This similarity metric usually appears in the form of a distance function. This algorithm is a nonparametric technique, which means it doesn't need to find out any underlying parameters before formulation. All we need to do is select a value of `k` that works for us.

Once we find out the k-nearest neighbors, we take a majority vote. A new datapoint is classified by this majority vote of the k-nearest neighbors. This datapoint is assigned to the class that is most common among its k-nearest neighbors. If we set the value of `k` to `1`, then this simply becomes a case of a nearest neighbor classifier where we just assign the datapoint to the class of its nearest neighbor in the training dataset. You can learn more about it at http://www.fon.hum.uva.nl/praat/manual/kNN_classifiers_1__What_is_a_kNN_classifier_.html.

Constructing a k-nearest neighbors regressor

We learned how to use k-nearest neighbors algorithm to build a classifier. The good thing is that we can also use this algorithm as a regressor. Let's see how to use it as a regressor.

How to do it…

1. Create a new Python file, and import the following packages:

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn import neighbors
```

2. Let's generate some sample Gaussian-distributed data:

```
# Generate sample data
amplitude = 10
num_points = 100
X = amplitude * np.random.rand(num_points, 1) - 0.5 * amplitude
```

3. We need to add some noise into the data to introduce some randomness into it. The goal of adding noise is to see whether our algorithm can get past it and still function in a robust way:

```
# Compute target and add noise
y = np.sinc(X).ravel()
y += 0.2 * (0.5 - np.random.rand(y.size))
```

4. Let's visualize it as follows:

```
# Plot input data
plt.figure()
plt.scatter(X, y, s=40, c='k', facecolors='none')
plt.title('Input data')
```

5. We just generated some data and evaluated a continuous-valued function on all these points. Let's define a denser grid of points:

```
# Create the 1D grid with 10 times the density of the input data
x_values = np.linspace(-0.5*amplitude, 0.5*amplitude,
10*num_points)[:, np.newaxis]
```

   We defined this denser grid because we want to evaluate our regressor on all these points and look at how well it approximates our function.

6. Let's define the number of nearest neighbors that we want to consider:

```
# Number of neighbors to consider
n_neighbors = 8
```

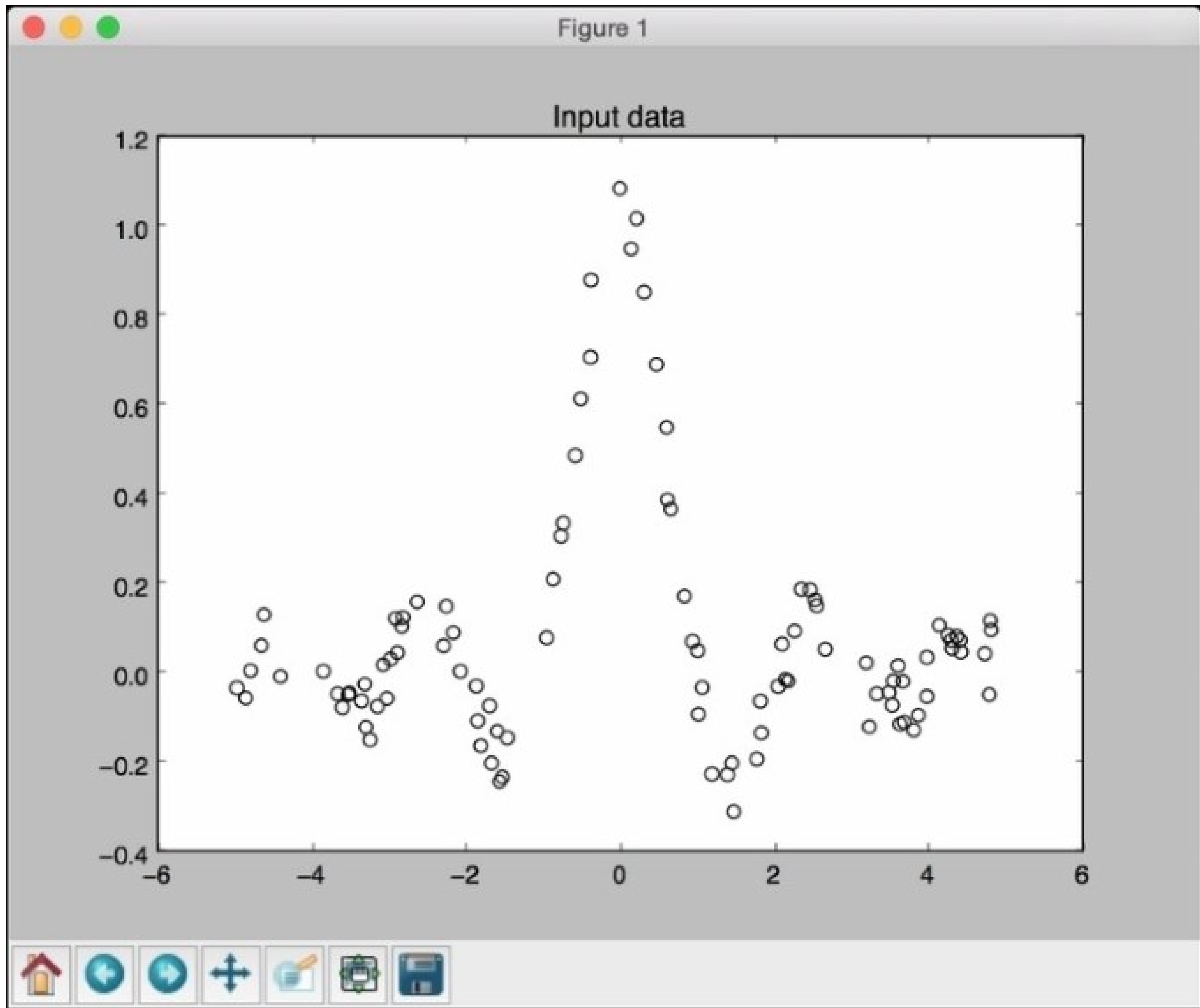
7. Let's initialize and train the k-nearest neighbors regressor using the parameters that we defined earlier:

```
# Define and train the regressor
knn_regressor = neighbors.KNeighborsRegressor(n_neighbors,
weights='distance')
y_values = knn_regressor.fit(X, y).predict(x_values)
```

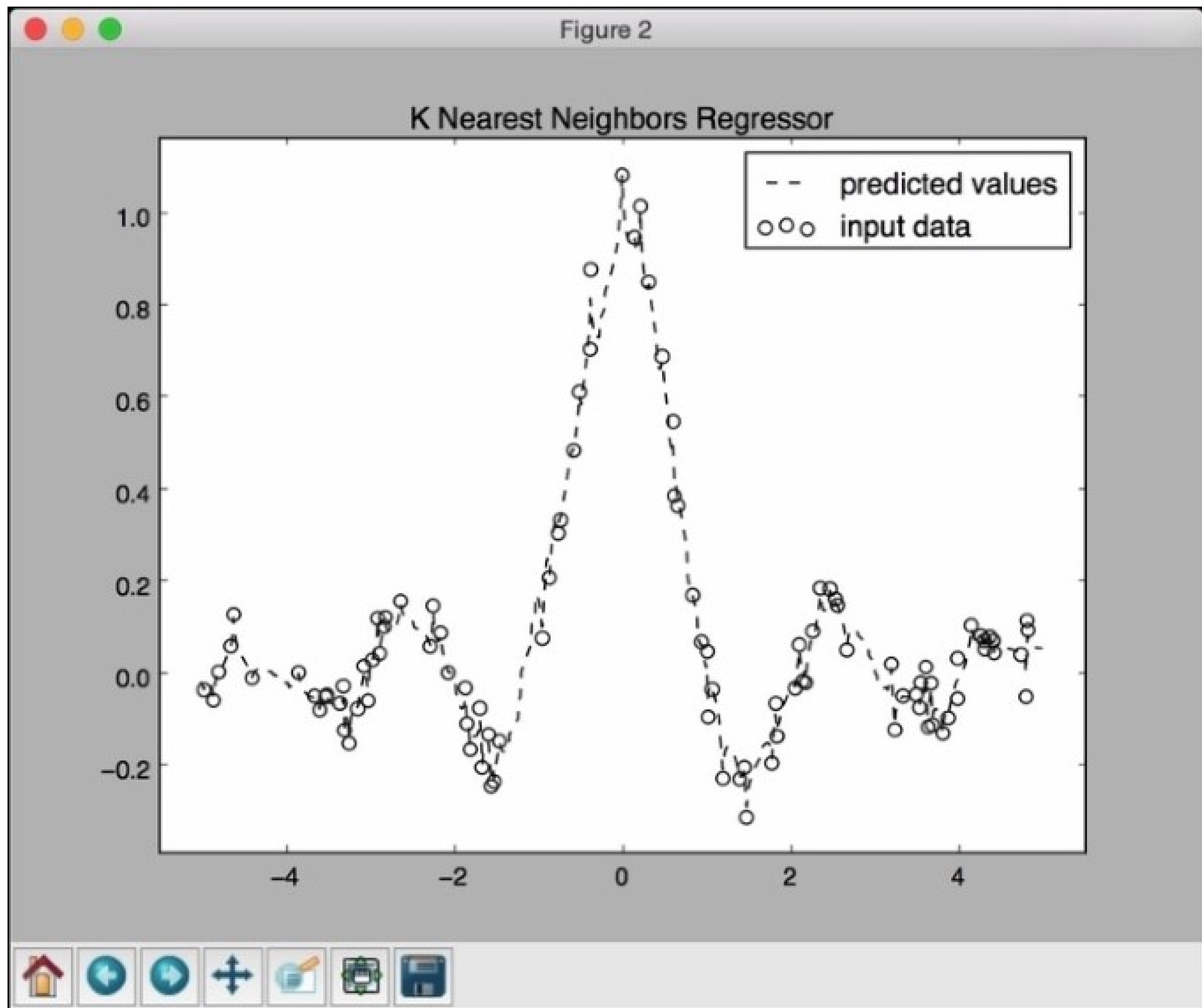
8. Let's see how the regressor performs by overlapping the input and output data on top of each other:

```
plt.figure()
plt.scatter(X, y, s=40, c='k', facecolors='none', label='input
data')
plt.plot(x_values, y_values, c='k', linestyle='--',
label='predicted values')
plt.xlim(X.min() - 1, X.max() + 1)
plt.ylim(y.min() - 0.2, y.max() + 0.2)
plt.axis('tight')
plt.legend()
plt.title('K Nearest Neighbors Regressor')

plt.show()
```

9. If you run this code, the first figure depicts the input datapoints:



10. The second figure depicts the predicted values by the regressor:



How it works…

The goal of a regressor is to predict continuous valued outputs. We don't have a fixed number of output categories in this case. We just have a set of real-valued output values, and we want our regressor to predict the output values for unknown datapoints. In this case, we used a `sinc` function to demonstrate the k-nearest neighbors regressor. This is also referred to as the **cardinal sine function**. A `sinc` function is defined by the following:

sinc(x) = sin(x)/x when x is not 0

= 1 when x is 0

When `x` is `0`, *sin(x)/x* takes the indeterminate form of *0/0*. Hence, we have to compute the limit of this function as `x` tends to `0`. We used a set of values for training, and we defined a denser grid for testing. As we can see in the preceding figure, the output curve is close to the training outputs.

Computing the Euclidean distance score

Now that we have sufficient background in machine learning pipelines and nearest neighbors classifier, let's start the discussion on recommendation engines. In order to build a recommendation engine, we need to define a similarity metric so that we can find users in the database who are similar to a given user. Euclidean distance score is one such metric that we can use to compute the distance between datapoints. We will focus the discussion towards movie recommendation engines. Let's see how to compute the Euclidean score between two users.

How to do it…

1. Create a new Python file, and import the following packages:

```
import json
import numpy as np
```

2. We will now define a function to compute the Euclidean score between two users. The first step is to check whether the users are present in the database:

```
# Returns the Euclidean distance score between user1 and user2
def euclidean_score(dataset, user1, user2):
    if user1 not in dataset:
        raise TypeError('User ' + user1 + ' not present in the
dataset')

    if user2 not in dataset:
        raise TypeError('User ' + user2 + ' not present in the
dataset')
```

3. In order to compute the score, we need to extract the movies that both the users rated:

```
    # Movies rated by both user1 and user2
    rated_by_both = {}

    for item in dataset[user1]:
        if item in dataset[user2]:
            rated_by_both[item] = 1
```

4. If there are no common movies, then there is no similarity between the users (or at least we cannot compute it given the ratings in the database):

```
    # If there are no common movies, the score is 0
    if len(rated_by_both) == 0:
        return 0
```

5. For each of the common ratings, we just compute the square root of the sum of squared differences and normalize it so that the score is between 0 and 1:

```
    squared_differences = []

    for item in dataset[user1]:
        if item in dataset[user2]:

squared_differences.append(np.square(dataset[user1][item] -
dataset[user2][item]))

    return 1 / (1 + np.sqrt(np.sum(squared_differences)))
```

If the ratings are similar, then the sum of squared differences will be very low. Hence, the score will become high, which is what we want from this metric.

6. We will use the `movie_ratings.json` file as our data file. Let's load it:

```
if __name__=='__main__':
    data_file = 'movie_ratings.json'

    with open(data_file, 'r') as f:
        data = json.loads(f.read())
```

7. Let's consider two random users and compute the Euclidean distance score:

```
    user1 = 'John Carson'
    user2 = 'Michelle Peterson'

    print "\nEuclidean score:"
    print euclidean_score(data, user1, user2)
```

8. When you run this code, you will see the Euclidean distance score printed on the Terminal.

Computing the Pearson correlation score

The Euclidean distance score is a good metric, but it has some shortcomings. Hence, Pearson correlation score is frequently used in recommendation engines. Let's see how to compute it.

How to do it…

1. Create a new Python file, and import the following packages:

```
import json
import numpy as np
```

2. We will define a function to compute the Pearson correlation score between two users in the database. Our first step is to confirm that these users exist in the database:

```
# Returns the Pearson correlation score between user1 and user2
def pearson_score(dataset, user1, user2):
    if user1 not in dataset:
        raise TypeError('User ' + user1 + ' not present in the
dataset')

    if user2 not in dataset:
        raise TypeError('User ' + user2 + ' not present in the
dataset')
```

3. The next step is to get the movies that both these users rated:

```
    # Movies rated by both user1 and user2
    rated_by_both = {}

    for item in dataset[user1]:
        if item in dataset[user2]:
            rated_by_both[item] = 1

    num_ratings = len(rated_by_both)
```

4. If there are no common movies, then there is no discernible similarity between these users; hence, we return `0`:

```
    # If there are no common movies, the score is 0
    if num_ratings == 0:
        return 0
```

5. We need to compute the sum of squared values of common movie ratings:

```
    # Compute the sum of ratings of all the common preferences
    user1_sum = np.sum([dataset[user1][item] for item in
rated_by_both])
```

```
    user2_sum = np.sum([dataset[user2][item] for item in
rated_by_both])
```

6. Let's compute the sum of squared ratings of all the common movie ratings:

```
    # Compute the sum of squared ratings of all the common
preferences
    user1_squared_sum = np.sum([np.square(dataset[user1][item])
for item in rated_by_both])
    user2_squared_sum = np.sum([np.square(dataset[user2][item])
for item in rated_by_both])
```

7. Let's compute the sum of the products:

```
    # Compute the sum of products of the common ratings
    product_sum = np.sum([dataset[user1][item] *
dataset[user2][item] for item in rated_by_both])
```

8. We are now ready to compute the various elements that we require to calculate the Pearson correlation score:

```
    # Compute the Pearson correlation
    Sxy = product_sum - (user1_sum * user2_sum / num_ratings)
    Sxx = user1_squared_sum - np.square(user1_sum) / num_ratings
    Syy = user2_squared_sum - np.square(user2_sum) / num_ratings
```

9. We need to take care of the case where the denominator becomes `0`:

```
    if Sxx * Syy == 0:
        return 0
```

10. If everything is good, we return the Pearson correlation score, as follows:

```
    return Sxy / np.sqrt(Sxx * Syy)
```

11. Let's define the `main` function and compute the Pearson correlation score between two users:

```
if __name__=='__main__':
    data_file = 'movie_ratings.json'

    with open(data_file, 'r') as f:
        data = json.loads(f.read())

    user1 = 'John Carson'
    user2 = 'Michelle Peterson'

    print "\nPearson score:"
    print pearson_score(data, user1, user2)
```

12. If you run this code, you will see the Pearson correlation score printed on the Terminal.

Finding similar users in the dataset

One of the most important tasks in building a recommendation engine is finding users that are similar. This guides in creating the recommendations that will be provided to these users. Let's see how to build this.

How to do it…

1. Create a new Python file, and import the following packages:

```
import json
import numpy as np

from pearson_score import pearson_score
```

2. Let's define a function to find similar users to the input user. It takes three input arguments: the database, input user, and the number of similar users that we are looking for. Our first step is to check whether the user is present in the database. If the user exists, we need to compute the Pearson correlation score between this user and all the other users in the database:

```
# Finds a specified number of users who are similar to the
input user
def find_similar_users(dataset, user, num_users):
    if user not in dataset:
        raise TypeError('User ' + user + ' not present in the
dataset')

    # Compute Pearson scores for all the users
    scores = np.array([[x, pearson_score(dataset, user, x)] for
x in dataset if user != x])
```

3. The next step is to sort these scores in descending order:

```
    # Sort the scores based on second column
    scores_sorted = np.argsort(scores[:, 1])

    # Sort the scores in decreasing order (highest score first)
    scored_sorted_dec = scores_sorted[::-1]
```

4. Let's extract the *k* top scores and return them:

```
    # Extract top 'k' indices
    top_k = scored_sorted_dec[0:num_users]

    return scores[top_k]
```

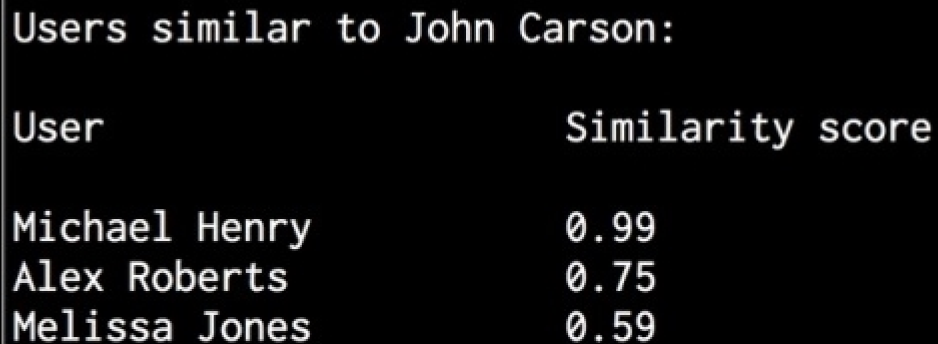
5. Let's define the `main` function and load the input database:

```
if __name__=='__main__':
    data_file = 'movie_ratings.json'

    with open(data_file, 'r') as f:
        data = json.loads(f.read())
```

6. We want to find three similar users to, for example, `John Carson`. We do this using the following steps:

```
    user = 'John Carson'
    print "\nUsers similar to " + user + ":\n"
    similar_users = find_similar_users(data, user, 3)
    print "User\t\t\tSimilarity score\n"
    for item in similar_users:
        print item[0], '\t\t', round(float(item[1]), 2)
```

7. If you run this code, you will see the following printed on your Terminal:



Generating movie recommendations

Now that we built all of the different parts of a recommendation engine, we are ready to generate movie recommendations. We will use all the functionality that we built in the previous recipes to build a movie recommendation engine. Let's see how to build it.

How to do it…

1. Create a new Python file, and import the following packages:

```
import json
import numpy as np

from euclidean_score import euclidean_score
from pearson_score import pearson_score
from find_similar_users import find_similar_users
```

2. We will define a function to generate movie recommendations for a given user. The first step is to check whether the user exists in the dataset:

```
# Generate recommendations for a given user
def generate_recommendations(dataset, user):
    if user not in dataset:
        raise TypeError('User ' + user + ' not present in the
dataset')
```

3. Let's compute the Pearson score of this user with all the other users in the dataset:

```
    total_scores = {}
    similarity_sums = {}

    for u in [x for x in dataset if x != user]:
        similarity_score = pearson_score(dataset, user, u)

        if similarity_score <= 0:
            continue
```

4. We need to find the movies that haven't been rated by this user:

```
        for item in [x for x in dataset[u] if x not in
dataset[user] or dataset[user][x] == 0]:
            total_scores.update({item: dataset[u][item] *
similarity_score})
            similarity_sums.update({item: similarity_score})
```

5. If the user has watched every single movie in the database, then we cannot recommend anything to this user. Let's take care of this condition:

```
    if len(total_scores) == 0:
        return ['No recommendations possible']
```

6. We now have a list of these scores. Let's create a normalized list of movie ranks:

```
    # Create the normalized list
    movie_ranks = np.array([[total/similarity_sums[item], item]
            for item, total in total_scores.items()])
```

7. We need to sort it in descending order based on the score:

```
    # Sort in decreasing order based on the first column
    movie_ranks = movie_ranks[np.argsort(movie_ranks[:,
0])[::-1]]
```

8. We are finally ready to extract the movie recommendations:

```
    # Extract the recommended movies
    recommendations = [movie for _, movie in movie_ranks]

    return recommendations
```

9. Let's define the `main` function and load the dataset:

```
if __name__=='__main__':
    data_file = 'movie_ratings.json'

    with open(data_file, 'r') as f:
        data = json.loads(f.read())
```

10. Let's generate recommendations for `Michael Henry`:

```
    user = 'Michael Henry'
    print "\nRecommendations for " + user + ":"
    movies = generate_recommendations(data, user)
    for i, movie in enumerate(movies):
        print str(i+1) + '. ' + movie
```

11. The user `John Carson` has watched all the movies. Therefore, if we try to generate recommendations for him, it should display 0 recommendations. Let's see whether this happens:

```
    user = 'John Carson'
    print "\nRecommendations for " + user + ":"
    movies = generate_recommendations(data, user)
    for i, movie in enumerate(movies):
        print str(i+1) + '. ' + movie
```

12. If you run this code, you will see the following on your Terminal:

```
Recommendations for Michael Henry:
1. Jerry Maguire
2. Anger Management
3. Inception

Recommendations for John Carson:
1. No recommendations possible
```

Chapter 6. Analyzing Text Data

In this chapter, we will cover the following recipes:

- Preprocessing data using tokenization
- Stemming text data
- Converting text to its base form using lemmatization
- Dividing text using chunking
- Building a bag-of-words model
- Building a text classifier
- Identifying the gender
- Analyzing the sentiment of a sentence
- Identifying patterns in text using topic modeling

Introduction

Text analysis and **natural language processing** (**NLP**) is an integral part of modern artificial intelligence systems. Computers are good at understanding rigidly-structured data with limited variety. However, when we deal with unstructured free-form text, things begin to get difficult. Developing NLP applications is challenging because computers have a hard time understanding underlying concepts. There are also many subtle variations to the way in which we communicate things. These can be in the form of dialects, context, slang, and so on.

In order to solve this problem, NLP applications are developed based on machine learning. These algorithms detect patterns in text data so that we can extract insights from it. Artificial intelligence companies make heavy use of NLP and text analysis to deliver relevant results. Some of the most common applications of NLP include search engines, sentiment analysis, topic modeling, part-of-speech tagging, entity recognition, and so on. The goal of NLP is to develop a set of algorithms so that we can interact with computers in plain English. If we can achieve this, then we wouldn't need programming languages to instruct computers about what they should do. In this chapter, we will look at a few recipes that focus on text analysis and how we can extract meaningful information from text data. We will use a Python package called **Natural Language Toolkit** (**NLTK**) heavily in this chapter. Make sure that you install this before you proceed. You can find the installation steps at http://www.nltk.org/install.html. You also need to install NLTK Data, which contains many corpora and trained models. This is an integral part of text analysis! You can find the installation steps at http://www.nltk.org/data.html.

Preprocessing data using tokenization

Tokenization is the process of dividing text into a set of meaningful pieces. These pieces are called **tokens**. For example, we can divide a chunk of text into words, or we can divide it into sentences. Depending on the task at hand, we can define our own conditions to divide the input text into meaningful tokens. Let's take a look at how to do this.

How to do it…

1. Create a new Python file and add the following lines. Let's define some sample text for analysis:

   ```
   text = "Are you curious about tokenization? Let's see how it
   works! We need to analyze a couple of sentences with
   punctuations to see it in action."
   ```

2. Let's start with sentence tokenization. NLTK provides a sentence tokenizer, so let's import this:

   ```
   # Sentence tokenization
   from nltk.tokenize import sent_tokenize
   ```

3. Run the sentence tokenizer on the input text and extract the tokens:

   ```
   sent_tokenize_list = sent_tokenize(text)
   ```

4. Print the list of sentences to see whether it works correctly:

   ```
   print "\nSentence tokenizer:"
   print sent_tokenize_list
   ```

5. Word tokenization is very commonly used in NLP. NLTK comes with a couple of different word tokenizers. Let's start with the basic word tokenizer:

   ```
   # Create a new word tokenizer
   from nltk.tokenize import word_tokenize

   print "\nWord tokenizer:"
   print word_tokenize(text)
   ```

6. There is another word tokenizer that is available called `PunktWord` Tokenizer. This splits the text on punctuation, but this keeps it within the words:

   ```
   # Create a new punkt word tokenizer
   from nltk.tokenize import PunktWordTokenizer

   punkt_word_tokenizer = PunktWordTokenizer()
   print "\nPunkt word tokenizer:"
   print punkt_word_tokenizer.tokenize(text)
   ```

7. If you want to split these punctuations into separate tokens, then we need to use `WordPunct` Tokenizer:

```
# Create a new WordPunct tokenizer
from nltk.tokenize import WordPunctTokenizer

word_punct_tokenizer = WordPunctTokenizer()
print "\nWord punct tokenizer:"
print word_punct_tokenizer.tokenize(text)
```

8. The full code is in the `tokenizer.py` file. If you run this code, you will see the following output on your Terminal:

```
Sentence tokenizer:
['Are you curious about tokenization?', "Let's see how it works!", 'We need to analyze a couple of senten
ces with punctuations to see it in action.']

Word tokenizer:
['Are', 'you', 'curious', 'about', 'tokenization', '?', 'Let', "'s", 'see', 'how', 'it', 'works', '!', 'W
e', 'need', 'to', 'analyze', 'a', 'couple', 'of', 'sentences', 'with', 'punctuations', 'to', 'see', 'it',
 'in', 'action', '.']

Punkt word tokenizer:
['Are', 'you', 'curious', 'about', 'tokenization', '?', 'Let', "'s", 'see', 'how', 'it', 'works', '!', 'W
e', 'need', 'to', 'analyze', 'a', 'couple', 'of', 'sentences', 'with', 'punctuations', 'to', 'see', 'it',
 'in', 'action.']

Word punct tokenizer:
['Are', 'you', 'curious', 'about', 'tokenization', '?', 'Let', "'", 's', 'see', 'how', 'it', 'works', '!'
, 'We', 'need', 'to', 'analyze', 'a', 'couple', 'of', 'sentences', 'with', 'punctuations', 'to', 'see', '
it', 'in', 'action', '.']
```

Stemming text data

When we deal with a text document, we encounter different forms of a word. Consider the word "play". This word can appear in various forms, such as "play", "plays", "player", "playing", and so on. These are basically families of words with similar meanings. During text analysis, it's useful to extract the base form of these words. This will help us in extracting some statistics to analyze the overall text. The goal of stemming is to reduce these different forms into a common base form. This uses a heuristic process to cut off the ends of words to extract the base form. Let's see how to do this in Python.

How to do it…

1. Create a new Python file, and import the following packages:

```
from nltk.stem.porter import PorterStemmer
from nltk.stem.lancaster import LancasterStemmer
from nltk.stem.snowball import SnowballStemmer
```

2. Let's define a few words to play with, as follows:

```
words = ['table', 'probably', 'wolves', 'playing', 'is',
        'dog', 'the', 'beaches', 'grounded', 'dreamt',
'envision']
```

3. We'll define a list of stemmers that we want to use:

```
# Compare different stemmers
stemmers = ['PORTER', 'LANCASTER', 'SNOWBALL']
```

4. Initialize the required objects for all three stemmers:

```
stemmer_porter = PorterStemmer()
stemmer_lancaster = LancasterStemmer()
stemmer_snowball = SnowballStemmer('english')
```

5. In order to print the output data in a neat tabular form, we need to format it in the right way:

```
formatted_row = '{:>16}' * (len(stemmers) + 1)
print '\n', formatted_row.format('WORD', *stemmers), '\n'
```

6. Let's iterate through the list of words and stem them using the three stemmers:

```
for word in words:
    stemmed_words = [stemmer_porter.stem(word),
            stemmer_lancaster.stem(word),
            stemmer_snowball.stem(word)]
    print formatted_row.format(word, *stemmed_words)
```

7. The full code is in the `stemmer.py` file. If you run this code, you will see the following output on your Terminal. Observe how the Lancaster stemmer behaves differently for a couple of words:

WORD	PORTER	LANCASTER	SNOWBALL
table	tabl	tabl	tabl
probably	probabl	prob	probabl
wolves	wolv	wolv	wolv
playing	play	play	play
is	is	is	is
dog	dog	dog	dog
the	the	the	the
beaches	beach	beach	beach
grounded	ground	ground	ground
dreamt	dreamt	dreamt	dreamt
envision	envis	envid	envis

How it works…

All three stemming algorithms basically aim to achieve the same thing. The difference between the three stemming algorithms is basically the level of strictness with which they operate. If you observe the outputs, you will see that the Lancaster stemmer is stricter than the other two stemmers. The Porter stemmer is the least in terms of strictness and Lancaster is the strictest. The stemmed words that we get from Lancaster stemmer tend to get confusing and obfuscated. The algorithm is really fast but it will reduce the words a lot. So, a good rule of thumb is to use the Snowball stemmer.

Converting text to its base form using lemmatization

The goal of lemmatization is also to reduce words to their base forms, but this is a more structured approach. In the previous recipe, we saw that the base words that we obtained using stemmers don't really make sense. For example, the word "wolves" was reduced to "wolv", which is not a real word. Lemmatization solves this problem by doing things using a vocabulary and morphological analysis of words. It removes inflectional word endings, such as "ing" or "ed", and returns the base form of a word. This base form is known as the lemma. If you lemmatize the word "wolves", you will get "wolf" as the output. The output depends on whether the token is a verb or a noun. Let's take a look at how to do this in this recipe.

How to do it…

1. Create a new Python file, and import the following package:

```
from nltk.stem import WordNetLemmatizer
```

2. Let's define the same set of words that we used during stemming:

```
words = ['table', 'probably', 'wolves', 'playing', 'is',
        'dog', 'the', 'beaches', 'grounded', 'dreamt',
'envision']
```

3. We will compare two lemmatizers, the `NOUN` and `VERB` lemmatizers. Let's list them as follows:

```
# Compare different lemmatizers
lemmatizers = ['NOUN LEMMATIZER', 'VERB LEMMATIZER']
```

4. Create the object based on `WordNet` lemmatizer:

```
lemmatizer_wordnet = WordNetLemmatizer()
```

5. In order to print the output in a tabular form, we need to format it in the right way:

```
formatted_row = '{:>24}' * (len(lemmatizers) + 1)
print '\n', formatted_row.format('WORD', *lemmatizers), '\n'
```

6. Iterate through the words and lemmatize them:

```
for word in words:
    lemmatized_words = [lemmatizer_wordnet.lemmatize(word,
pos='n'),
           lemmatizer_wordnet.lemmatize(word, pos='v')]
    print formatted_row.format(word, *lemmatized_words)
```

7. The full code is in the `lemmatizer.py` file. If you run this code, you will see the following output. Observe how `NOUN` and `VERB` lemmatizers differ when they lemmatize the word "is" in the following image:

WORD	NOUN LEMMATIZER	VERB LEMMATIZER
table	table	table
probably	probably	probably
wolves	wolf	wolves
playing	playing	play
is	is	be
dog	dog	dog
the	the	the
beaches	beach	beach
grounded	grounded	ground
dreamt	dreamt	dream
envision	envision	envision

Dividing text using chunking

Chunking refers to dividing the input text into pieces, which are based on any random condition. This is different from tokenization in the sense that there are no constraints and the chunks do not need to be meaningful at all. This is used very frequently during text analysis. When you deal with really large text documents, you need to divide it into chunks for further analysis. In this recipe, we will divide the input text into a number of pieces, where each piece has a fixed number of words.

How to do it…

1. Create a new Python file, and import the following packages:

```
import numpy as np
from nltk.corpus import brown
```

2. Let's define a function to split text into chunks. The first step is to divide the text based on spaces:

```
# Split a text into chunks
def splitter(data, num_words):
    words = data.split(' ')
    output = []
```

3. Initialize a couple of required variables:

```
    cur_count = 0
    cur_words = []
```

4. Let's iterate through the words:

```
    for word in words:
        cur_words.append(word)
        cur_count += 1
```

5. Once you hit the required number of words, reset the variables:

```
        if cur_count == num_words:
            output.append(' '.join(cur_words))
            cur_words = []
            cur_count = 0
```

6. Append the chunks to the output variable, and return it:

```
    output.append(' '.join(cur_words) )

    return output
```

7. We can now define the `main` function. Load the data from Brown corpus. We will use the first 10,000 words:

```
if __name__=='__main__':
    # Read the data from the Brown corpus
    data = ' '.join(brown.words()[:10000])
```

8. Define the number of words in each chunk:

```
    # Number of words in each chunk
    num_words = 1700
```

9. Initialize a couple of relevant variables:

```
    chunks = []
    counter = 0
```

10. Call the `splitter` function on this text data and print the output:

```
    text_chunks = splitter(data, num_words)

    print "Number of text chunks =", len(text_chunks)
```

11. The full code is in the `chunking.py` file. If you run this code, you will see the number of chunks generated printed on the Terminal. It should be 6!

Building a bag-of-words model

When we deal with text documents that contain millions of words, we need to convert them into some kind of numeric representation. The reason for this is to make them usable for machine learning algorithms. These algorithms need numerical data so that they can analyze them and output meaningful information. This is where the **bag-of-words** approach comes into picture. This is basically a model that learns a vocabulary from all the words in all the documents. After this, it models each document by building a histogram of all the words in the document.

How to do it…

1. Create a new Python file, and import the following packages:

```
import numpy as np
from nltk.corpus import brown
from chunking import splitter
```

2. Let's define the `main` function. Load the input data from the Brown corpus:

```
if __name__=='__main__':
    # Read the data from the Brown corpus
    data = ' '.join(brown.words()[:10000])
```

3. Divide the text data into five chunks:

```
    # Number of words in each chunk
    num_words = 2000

    chunks = []
    counter = 0

    text_chunks = splitter(data, num_words)
```

4. Create a dictionary that is based on these text chunks:

```
    for text in text_chunks:
        chunk = {'index': counter, 'text': text}
        chunks.append(chunk)
        counter += 1
```

5. The next step is to extract a document term matrix. This is basically a matrix that counts the number of occurrences of each word in the document. We will use scikit-learn to do this because it has better provisions as compared to NLTK for this particular task. Import the following package:

```
    # Extract document term matrix
    from sklearn.feature_extraction.text import CountVectorizer
```

6. Define the object, and extract the document term matrix:

```
    vectorizer = CountVectorizer(min_df=5, max_df=.95)
    doc_term_matrix = vectorizer.fit_transform([chunk['text']
for chunk in chunks])
```

7. Extract the vocabulary from the `vectorizer` object and print it:

```
    vocab = np.array(vectorizer.get_feature_names())
    print "\nVocabulary:"
    print vocab
```

8. Print the document term matrix:

```
    print "\nDocument term matrix:"
    chunk_names = ['Chunk-0', 'Chunk-1', 'Chunk-2', 'Chunk-3',
'Chunk-4']
```

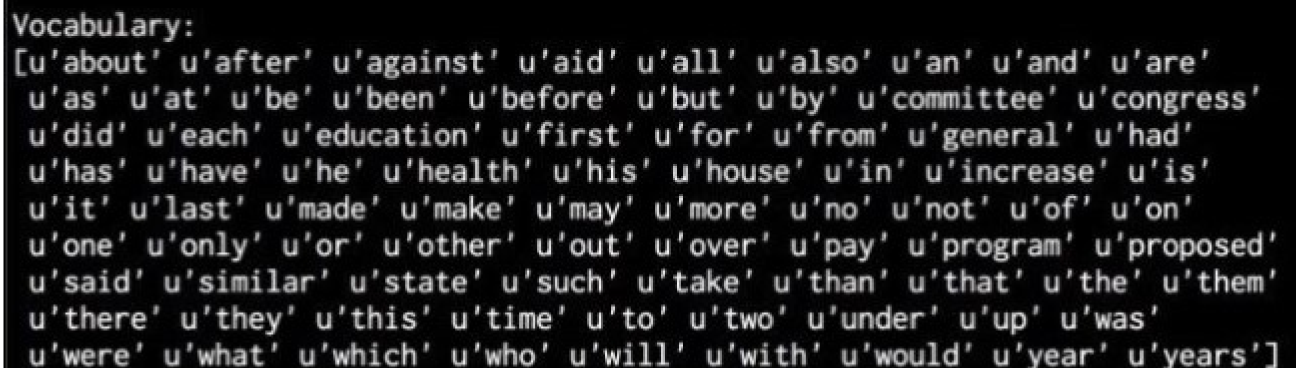
9. To print in tabular form, we need to format this, as follows:

```
    formatted_row = '{:>12}' * (len(chunk_names) + 1)
    print '\n', formatted_row.format('Word', *chunk_names), '\n'
```

10. Iterate through the words, and print the number of times each word has occurred in different chunks:

```
    for word, item in zip(vocab, doc_term_matrix.T):
        # 'item' is a 'csr_matrix' data structure
        output = [str(x) for x in item.data]
        print formatted_row.format(word, *output)
```

11. The full code is in the `bag_of_words.py` file. If you run this code, you will see two main things printed on the Terminal. The first output is the vocabulary as shown in the following image:



12. The second thing is the document term matrix, which is a pretty long. The first few lines will look like the following:

Document term matrix:

Word	Chunk-0	Chunk-1	Chunk-2	Chunk-3	Chunk-4
about	1	1	1	1	3
after	2	3	2	1	3
against	1	2	2	1	1
aid	1	1	1	3	5
all	2	2	5	2	1
also	3	3	3	4	3
an	5	7	5	7	10
and	34	27	36	36	41
are	5	3	6	3	2
as	13	4	14	18	4
at	5	7	9	3	6
be	20	14	7	10	18
been	7	1	6	15	5
before	2	2	1	1	2
but	3	3	2	9	5
by	8	22	15	14	12
committee	2	10	3	1	7

How it works…

Consider the following sentences:

- **Sentence 1**: The brown dog is running.
- **Sentence 2**: The black dog is in the black room.
- **Sentence 3**: Running in the room is forbidden.

If you consider all the three sentences, we have the following nine unique words:

- the
- brown
- dog
- is
- running
- black
- in
- room
- forbidden

Now, let's convert each sentence into a histogram using the count of words in each sentence. Each feature vector will be 9-dimensional because we have nine unique words:

- **Sentence 1**: [1, 1, 1, 1, 1, 0, 0, 0, 0]
- **Sentence 2**: [2, 0, 1, 1, 0, 2, 1, 1, 0]
- **Sentence 3**: [0, 0, 0, 1, 1, 0, 1, 1, 1]

Once we extract these feature vectors, we can use machine learning algorithms to analyze them.

Building a text classifier

The goal of text classification is to categorize text documents into different classes. This is an extremely important analysis technique in NLP. We will use a technique, which is based on a statistic called **tf-idf**, which stands for **term frequency—inverse document frequency**. This is an analysis tool that helps us understand how important a word is to a document in a set of documents. This serves as a feature vector that's used to categorize documents. You can learn more about it at http://www.tfidf.com.

How to do it…

1. Create a new Python file, and import the following package:

```
from sklearn.datasets import fetch_20newsgroups
```

2. Let's select a list of categories and name them using a dictionary mapping. These categories are available as part of the news groups dataset that we just imported:

```
category_map = {'misc.forsale': 'Sales', 'rec.motorcycles':
'Motorcycles',
        'rec.sport.baseball': 'Baseball', 'sci.crypt':
'Cryptography',
        'sci.space': 'Space'}
```

3. Load the training data based on the categories that we just defined:

```
training_data = fetch_20newsgroups(subset='train',
        categories=category_map.keys(), shuffle=True,
random_state=7)
```

4. Import the feature extractor:

```
# Feature extraction
from sklearn.feature_extraction.text import CountVectorizer
```

5. Extract the features using the training data:

```
vectorizer = CountVectorizer()
X_train_termcounts =
vectorizer.fit_transform(training_data.data)
print "\nDimensions of training data:", X_train_termcounts.shape
```

6. We are now ready to train the classifier. We will use the Multinomial Naive Bayes classifier:

```
# Training a classifier
from sklearn.naive_bayes import MultinomialNB
from sklearn.feature_extraction.text import TfidfTransformer
```

7. Define a couple of random input sentences:

```
input_data = [
    "The curveballs of right handed pitchers tend to curve to 
the left",
    "Caesar cipher is an ancient form of encryption",
    "This two-wheeler is really good on slippery roads"
]
```

8. Define the tf-idf transformer object and train it:

```
# tf-idf transformer
tfidf_transformer = TfidfTransformer()
X_train_tfidf = 
tfidf_transformer.fit_transform(X_train_termcounts)
```

9. Once we have the feature vectors, train the Multinomial Naive Bayes classifier using this data:

```
# Multinomial Naive Bayes classifier
classifier = MultinomialNB().fit(X_train_tfidf, 
training_data.target)
```

10. Transform the input data using the word counts:

```
X_input_termcounts = vectorizer.transform(input_data)
```

11. Transform the input data using the tf-idf transformer:

```
X_input_tfidf = tfidf_transformer.transform(X_input_termcounts)
```

12. Predict the output categories of these input sentences using the trained classifier:

```
# Predict the output categories
predicted_categories = classifier.predict(X_input_tfidf)
```

13. Print the outputs, as follows:

```
# Print the outputs
for sentence, category in zip(input_data, predicted_categories):
    print '\nInput:', sentence, '\nPredicted category:', \
            category_map[training_data.target_names[category]]
```

14. The full code is in the `tfidf.py` file. If you run this code, you will see the following output printed on your Terminal:

```
Dimensions of training data: (2968, 40605)

Input: The curveballs of right handed pitchers tend to curve to the left
Predicted category: Baseball

Input: Caesar cipher is an ancient form of encryption
Predicted category: Cryptography

Input: This two-wheeler is really good on slippery roads
Predicted category: Motorcycles
```

How it works…

The tf-idf technique is used frequently in information retrieval. The goal is to understand the importance of each word within a document. We want to identify words that are occur many times in a document. At the same time, common words like “is” and “be” don't really reflect the nature of the content. So we need to extract the words that are true indicators. The importance of each word increases as the count increases. At the same time, as it appears a lot, the frequency of this word increases too. These two things tend to balance each other out. We extract the term counts from each sentence. Once we convert this to a feature vector, we train the classifier to categorize these sentences.

The **term frequency** (**TF**) measures how frequently a word occurs in a given document. As multiple documents differ in length, the numbers in the histogram tend to vary a lot. So, we need to normalize this so that it becomes a level playing field. To achieve normalization, we divide term-frequency by the total number of words in a given document. The **inverse document frequency** (**IDF**) measures the importance of a given word. When we compute TF, all words are considered to be equally important. To counter-balance the frequencies of commonly-occurring words, we need to weigh them down and scale up the rare ones. We need to calculate the ratio of the number of documents with the given word and divide it by the total number of documents. IDF is calculated by taking the negative algorithm of this ratio.

For example, simple words, such as "is" or "the" tend to appear a lot in various documents. However, this doesn't mean that we can characterize the document based on these words. At the same time, if a word appears a single time, this is not useful either. So, we look for words that appear a number of times, but not so much that they become noisy. This is formulated in the tf-idf technique and used to classify documents. Search engines frequently use this tool to order the search results by relevance.

Identifying the gender

Identifying the gender of a name is an interesting task in NLP. We will use the heuristic that the last few characters in a name is its defining characteristic. For example, if the name ends with "la", it's most likely a female name, such as "Angela" or "Layla". On the other hand, if the name ends with "im", it's most likely a male name, such as "Tim" or "Jim". As we are sure of the exact number of characters to use, we will experiment with this. Let's see how to do it.

How to do it…

1. Create a new Python file, and import the following packages:

```
import random
from nltk.corpus import names
from nltk import NaiveBayesClassifier
from nltk.classify import accuracy as nltk_accuracy
```

2. We need to define a function to extract features from input words:

```
# Extract features from the input word
def gender_features(word, num_letters=2):
    return {'feature': word[-num_letters:].lower()}
```

3. Let's define the `main` function. We need some labeled training data:

```
if __name__=='__main__':
    # Extract labeled names
    labeled_names = ([(name, 'male') for name in 
names.words('male.txt')] +
            [(name, 'female') for name in 
names.words('female.txt')])
```

4. Seed the random number generator, and shuffle the training data:

```
    random.seed(7)
    random.shuffle(labeled_names)
```

5. Define some input names to play with:

```
    input_names = ['Leonardo', 'Amy', 'Sam']
```

6. As we don't know how many ending characters we need to consider, we will sweep the parameter space from `1` to `5`. Each time, we will extract the features, as follows:

```
    # Sweeping the parameter space
    for i in range(1, 5):
        print '\nNumber of letters:', i
        featuresets = [(gender_features(n, i), gender) for (n, 
gender) in labeled_names]
```

7. Divide this into train and test datasets:

```
        train_set, test_set = featuresets[500:],
featuresets[:500]
```

8. We will use the Naive Bayes classifier to do this:

```
        classifier = NaiveBayesClassifier.train(train_set)
```

9. Evaluate the classifier for each value in the parameter space:

```
        # Print classifier accuracy
        print 'Accuracy ==>', str(100 *
nltk_accuracy(classifier, test_set)) + str('%')

# Predict outputs for new inputs
        for name in input_names:
            print name, '==>',
classifier.classify(gender_features(name, i))
```

10. The full code is in the `gender_identification.py` file. If you run this code, you will see the following output printed on your Terminal:

```
Number of letters: 1
Accuracy ==> 76.6%
Leonardo ==> male
Amy ==> female
Sam ==> male

Number of letters: 2
Accuracy ==> 80.2%
Leonardo ==> male
Amy ==> female
Sam ==> male

Number of letters: 3
Accuracy ==> 78.4%
Leonardo ==> male
Amy ==> female
Sam ==> female

Number of letters: 4
Accuracy ==> 71.6%
Leonardo ==> male
Amy ==> female
Sam ==> female
```

Analyzing the sentiment of a sentence

Sentiment analysis is one of the most popular applications of NLP. Sentiment analysis refers to the process of determining whether a given piece of text is positive or negative. In some variations, we consider "neutral" as a third option. This technique is commonly used to discover how people feel about a particular topic. This is used to analyze sentiments of users in various forms, such as marketing campaigns, social media, e-commerce customers, and so on.

How to do it…

1. Create a new Python file, and import the following packages:

```
import nltk.classify.util
from nltk.classify import NaiveBayesClassifier
from nltk.corpus import movie_reviews
```

2. Define a function to extract features:

```
def extract_features(word_list):
    return dict([(word, True) for word in word_list])
```

3. We need training data for this, so we will use movie reviews in NLTK:

```
if __name__=='__main__':
    # Load positive and negative reviews
    positive_fileids = movie_reviews.fileids('pos')
    negative_fileids = movie_reviews.fileids('neg')
```

4. Let's separate these into positive and negative reviews:

```
    features_positive = 
[(extract_features(movie_reviews.words(fileids=[f])), 
            'Positive') for f in positive_fileids]
    features_negative = 
[(extract_features(movie_reviews.words(fileids=[f])), 
            'Negative') for f in negative_fileids]
```

5. Divide the data into training and testing datasets:

```
    # Split the data into train and test (80/20)
    threshold_factor = 0.8
    threshold_positive = int(threshold_factor * 
len(features_positive))
    threshold_negative = int(threshold_factor * 
len(features_negative))
```

6. Extract the features:

```
    features_train = features_positive[:threshold_positive] +
features_negative[:threshold_negative]
    features_test = features_positive[threshold_positive:] +
features_negative[threshold_negative:]
    print "\nNumber of training datapoints:",
len(features_train)
    print "Number of test datapoints:", len(features_test)
```

7. We will use a Naive Bayes classifier. Define the object and train it:

```
    # Train a Naive Bayes classifier
    classifier = NaiveBayesClassifier.train(features_train)
    print "\nAccuracy of the classifier:",
nltk.classify.util.accuracy(classifier, features_test)
```

8. The classifier object contains the most informative words that it obtained during analysis. These words basically have a strong say in what's classified as a positive or a negative review. Let's print them out:

```
    print "\nTop 10 most informative words:"
    for item in classifier.most_informative_features()[:10]:
        print item[0]
```

9. Create a couple of random input sentences:

```
    # Sample input reviews
    input_reviews = [
        "It is an amazing movie",
        "This is a dull movie. I would never recommend it to
anyone.",
        "The cinematography is pretty great in this movie",
        "The direction was terrible and the story was all over
the place"
    ]
```

10. Run the classifier on those input sentences and obtain the predictions:

```
    print "\nPredictions:"
    for review in input_reviews:
        print "\nReview:", review
        probdist =
classifier.prob_classify(extract_features(review.split()))
        pred_sentiment = probdist.max()
```

11. Print the output:

```
        print "Predicted sentiment:", pred_sentiment
        print "Probability:",
round(probdist.prob(pred_sentiment), 2)
```

12. The full code is in the `sentiment_analysis.py` file. If you run this code, you will see three main things printed on the Terminal. The first is the accuracy, as shown in the following image:

```
Number of training datapoints: 1600
Number of test datapoints: 400

Accuracy of the classifier: 0.735
```

13. The next is a list of most informative words:

```
Top 10 most informative words:
outstanding
insulting
vulnerable
ludicrous
uninvolving
astounding
avoids
fascination
animators
affecting
```

14. The last is the list of predictions, which are based on the input sentences:

```
Predictions:

Review: It is an amazing movie
Predicted sentiment: Positive
Probability: 0.61

Review: This is a dull movie. I would never recommend it to anyone.
Predicted sentiment: Negative
Probability: 0.77

Review: The cinematography is pretty great in this movie
Predicted sentiment: Positive
Probability: 0.67

Review: The direction was terrible and the story was all over the place
Predicted sentiment: Negative
Probability: 0.63
```

How it works…

We use NLTK's Naive Bayes classifier for our task here. In the feature extractor function, we basically extract all the unique words. However, the NLTK classifier needs the data to be arranged in the form of a dictionary. Hence, we arranged it in such a way that the NLTK classifier object can ingest it.

Once we divide the data into training and testing datasets, we train the classifier to categorize the sentences into positive and negative. If you look at the top informative words, you can see that we have words such as "outstanding" to indicate positive reviews and words such as "insulting" to indicate negative reviews. This is interesting information because it tells us what words are being used to indicate strong reactions.

Identifying patterns in text using topic modeling

The **topic modeling** refers to the process of identifying hidden patterns in text data. The goal is to uncover some hidden thematic structure in a collection of documents. This will help us in organizing our documents in a better way so that we can use them for analysis. This is an active area of research in NLP. You can learn more about it at http://www.cs.columbia.edu/~blei/topicmodeling.html. We will use a library called `gensim` during this recipe. Make sure that you install this before you proceed. The installation steps are given at https://radimrehurek.com/gensim/install.html.

How to do it…

1. Create a new Python file and import the following packages:

```
from nltk.tokenize import RegexpTokenizer
from nltk.stem.snowball import SnowballStemmer
from gensim import models, corpora
from nltk.corpus import stopwords
```

2. Define a function to load the input data. We will use the `data_topic_modeling.txt` text file that is already provided to you:

```
# Load input data
def load_data(input_file):
    data = []
    with open(input_file, 'r') as f:
        for line in f.readlines():
            data.append(line[:-1])

    return data
```

3. Let's define a class to preprocess text. This preprocessor takes care of creating the required objects and extracting the relevant features from input text:

```
# Class to preprocess text
class Preprocessor(object):
    # Initialize various operators
    def __init__(self):
        # Create a regular expression tokenizer
        self.tokenizer = RegexpTokenizer(r'\w+')
```

4. We need a list of stop words so that we can exclude them from analysis. These are common words, such as "in", "the", "is", and so on:

```
        # get the list of stop words
        self.stop_words_english = stopwords.words('english')
```

5. Define a snowball stemmer:

```
        # Create a Snowball stemmer
        self.stemmer = SnowballStemmer('english')
```

6. Define a processor function that takes care of tokenization, stop word removal, and stemming:

```
    # Tokenizing, stop word removal, and stemming
    def process(self, input_text):
        # Tokenize the string
        tokens = self.tokenizer.tokenize(input_text.lower())
```

7. Remove the stop words from the text:

```
        # Remove the stop words
        tokens_stopwords = [x for x in tokens if not x in
self.stop_words_english]
```

8. Perform stemming on tokens:

```
        # Perform stemming on the tokens
        tokens_stemmed = [self.stemmer.stem(x) for x in
tokens_stopwords]
```

9. Return the processed tokens:

```
        return tokens_stemmed
```

10. We are now ready to define the `main` function. Load the input data from the text file:

```
if __name__=='__main__':
    # File containing linewise input data
    input_file = 'data_topic_modeling.txt'

    # Load data
    data = load_data(input_file)
```

11. Define an object that is based on the class that we defined:

```
    # Create a preprocessor object
    preprocessor = Preprocessor()
```

12. We need to process the text in the file, and extract the processed tokens:

```
    # Create a list for processed documents
    processed_tokens = [preprocessor.process(x) for x in data]
```

13. Create a dictionary, which is based on tokenized documents so that this can be used for topic modeling:

```
    # Create a dictionary based on the tokenized documents
    dict_tokens = corpora.Dictionary(processed_tokens)
```

14. We need to create a document-term matrix using the processed tokens, as follows:

```
    # Create a document-term matrix
    corpus = [dict_tokens.doc2bow(text) for text in
processed_tokens]
```

15. Let's say we know that the text can be divided into two topics. We will use a technique called **Latent Dirichlet Allocation** (**LDA**) for topic modeling. Define the required parameters and initialize the LDA model object:

```
    # Generate the LDA model based on the corpus we just created
    num_topics = 2
    num_words = 4

    ldamodel = models.ldamodel.LdaModel(corpus,
            num_topics=num_topics, id2word=dict_tokens,
passes=25)
```

16. Once this identifies the two topics, we can see how it's separating these two topics by looking at the most-contributed words:

```
    print "Most contributing words to the topics:"
    for item in ldamodel.print_topics(num_topics=num_topics,
num_words=num_words):
        print "\nTopic", item[0], "==>", item[1]
```

17. The full code is in the `topic_modeling.py` file. If you run this code, you will see the following printed on your Terminal:

```
Most contributing words to the topics:

Topic 0 ==> 0.049*need + 0.030*younger + 0.030*talent + 0.030*train

Topic 1 ==> 0.064*need + 0.063*order + 0.038*encrypt + 0.038*understand
```

How it works…

Topic modeling works by identifying the important words of themes in a document. These words tend to determine what the topic is about. We use a regular expression tokenizer because we just want the words without any punctuation or other kinds of tokens. Hence, we use this to extract the tokens. Stop word removal is another important step because this helps us eliminate the noise caused due to words, such as "is" or "the". After this, we need to stem the words to get to their base forms. This entire thing is packaged as a preprocessing block in text analysis tools. This is what we are doing here as well!

We use a technique called Latent Dirichlet Allocation (LDA) to model the topics. LDA basically represents the documents as a mixture of different topics that tend to spit out words. These words are spat out with certain probabilities. The goal is to find these topics! This is a generative model that tries

to find the set of topics that are responsible for the generation of the given set of documents. You can learn more about it at http://blog.echen.me/2011/08/22/introduction-to-latent-dirichlet-allocation.

As you can see from the output, we have words such as "talent" and "train" to characterize the sports topic, whereas we have "encrypt" to characterize the cryptography topic. We are working with a really small text file, which is the reason why some words might seem less relevant. Obviously, the accuracy will improve if you work with a larger dataset.

Chapter 7. Speech Recognition

In this chapter, we will cover the following recipes:

- Reading and plotting audio data
- Transforming audio signals into the frequency domain
- Generating audio signals with custom parameters
- Synthesizing music
- Extracting frequency domain features
- Building Hidden Markov Models
- Building a speech recognizer

Introduction

Speech recognition refers to the process of recognizing and understanding spoken language. Input comes in the form of audio data, and the speech recognizers will process this data to extract meaningful information from it. This has a lot of practical uses, such as voice controlled devices, transcription of spoken language into words, security systems, and so on.

Speech signals are very versatile in nature. There are many variations of speech in the same language. There are different elements to speech, such as language, emotion, tone, noise, accent, and so on. It's difficult to rigidly define a set of rules that can constitute speech. Even with all these variations, humans are really good at understanding all of this with relative ease. Hence, we need machines to understand speech in the same way.

Over the last couple of decades, researchers have worked on various aspects of speech, such as identifying the speaker, understanding words, recognizing accents, translating speech, and so on. Among all these tasks, automatic speech recognition has been the focal point of attention for many researchers. In this chapter, we will learn how to build a **speech recognizer**.

Reading and plotting audio data

Let's take a look at how to read an audio file and visualize the signal. This will be a good starting point, and it will give us a good understanding about the basic structure of audio signals. Before we start, we need to understand that audio files are digitized versions of actual audio signals. Actual audio signals are complex continuous-valued waves. In order to save a digital version, we sample the signal and convert it into numbers. For example, speech is commonly sampled at 44100 Hz. This means that each second of the signal is broken down into 44100 parts, and the values at these timestamps are stored. In other words, you store a value every 1/44100 seconds. As the sampling rate is high, we feel that the signal is continuous when we listen to it on our media players.

How to do it…

1. Create a new Python file, and import the following packages:

   ```
   import numpy as np
   import matplotlib.pyplot as plt
   from scipy.io import wavfile
   ```

2. We will use the `wavfile` package to read the audio file from the `input_read.wav` input file that is already provided to you:

   ```
   # Read the input file
   sampling_freq, audio = wavfile.read('input_read.wav')
   ```

3. Let's print out the parameters of this signal:

   ```
   # Print the params
   print '\nShape:', audio.shape
   print 'Datatype:', audio.dtype
   print 'Duration:', round(audio.shape[0] / float(sampling_freq),
   3), 'seconds'
   ```

4. The audio signal is stored as 16-bit signed integer data. We need to normalize these values:

   ```
   # Normalize the values
   audio = audio / (2.**15)
   ```

5. Let's extract the first 30 values to plot, as follows:

   ```
   # Extract first 30 values for plotting
   audio = audio[:30]
   ```

6. The X-axis is the time axis. Let's build this axis, considering the fact that it should be scaled using the sampling frequency factor:

   ```
   # Build the time axis
   x_values = np.arange(0, len(audio), 1) / float(sampling_freq)
   ```

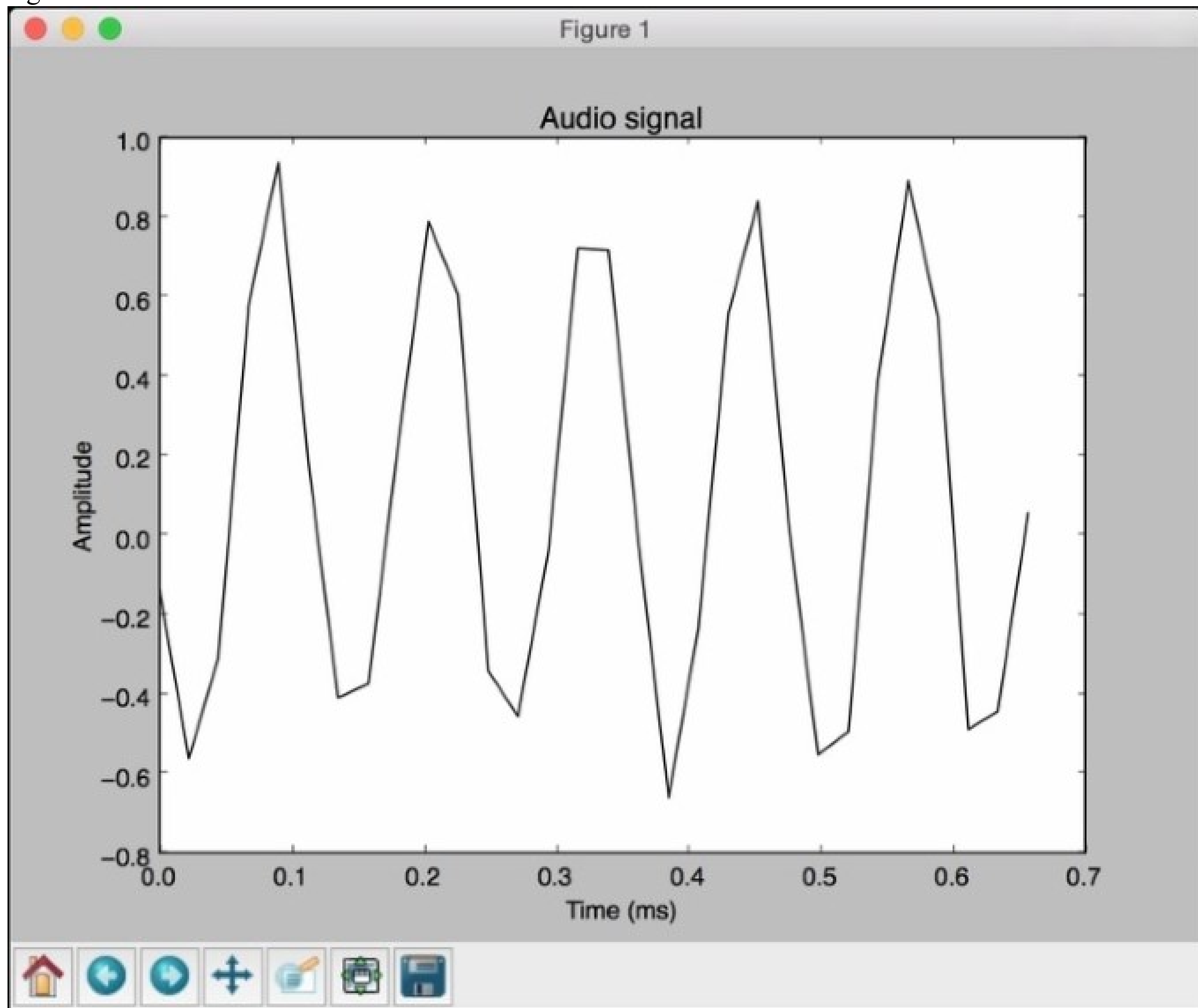
7. Convert the units to seconds:

```
# Convert to seconds
x_values *= 1000
```

8. Let's plot this as follows:

```
# Plotting the chopped audio signal
plt.plot(x_values, audio, color='black')
plt.xlabel('Time (ms)')
plt.ylabel('Amplitude')
plt.title('Audio signal')
plt.show()
```

9. The full code is in the `read_plot.py` file. If you run this code, you will see the following signal:



10. You will also see the following printed on your Terminal:

```
Shape: (132300,)
Datatype: int16
Duration: 3.0 seconds
```

Transforming audio signals into the frequency domain

Audio signals consist of a complex mixture of sine waves of different frequencies, amplitudes, and phases. Sine waves are also referred to as **sinusoids**. There is a lot of information that is hidden in the frequency content of an audio signal. In fact, an audio signal is heavily characterized by its frequency content. The whole world of speech and music is based on this fact. Before you proceed further, you will need some knowledge about **Fourier transforms**. A quick refresher can be found at http://www.thefouriertransform.com. Now, let's take a look at how to transform an audio signal into the frequency domain.

How to do it…

1. Create a new Python file, and import the following package:

```
import numpy as np
from scipy.io import wavfile
import matplotlib.pyplot as plt
```

2. Read the `input_freq.wav` file that is already provided to you:

```
# Read the input file
sampling_freq, audio = wavfile.read('input_freq.wav')
```

3. Normalize the signal, as follows:

```
# Normalize the values
audio = audio / (2.**15)
```

4. The audio signal is just a NumPy array. So, you can extract the length using the following code:

```
# Extract length
len_audio = len(audio)
```

5. Let's apply the Fourier transform. The Fourier transform signal is mirrored along the center, so we just need to take the first half of the transformed signal. Our end goal is to extract the power signal. So, we square the values in the signal in preparation for this:

```
# Apply Fourier transform
transformed_signal = np.fft.fft(audio)
half_length = np.ceil((len_audio + 1) / 2.0)
transformed_signal = abs(transformed_signal[0:half_length])
transformed_signal /= float(len_audio)
transformed_signal **= 2
```

6. Extract the length of the signal:

```
# Extract length of transformed signal
len_ts = len(transformed_signal)
```

7. We need to double the signal according to the length of the signal:

```
# Take care of even/odd cases
if len_audio % 2:
    transformed_signal[1:len_ts] *= 2
else:
    transformed_signal[1:len_ts-1] *= 2
```

8. The power signal is extracted using the following formula:

```
# Extract power in dB
power = 10 * np.log10(transformed_signal)
```

9. The X-axis is the time axis. We need to scale this according the sampling frequency and then convert this into seconds:

```
# Build the time axis
x_values = np.arange(0, half_length, 1) * (sampling_freq /
len_audio) / 1000.0
```

10. Plot the signal, as follows:

```
# Plot the figure
plt.figure()
plt.plot(x_values, power, color='black')
plt.xlabel('Freq (in kHz)')
plt.ylabel('Power (in dB)')
plt.show()
```

11. The full code is in the `freq_transform.py` file. If you run this code, you will see the following figure:

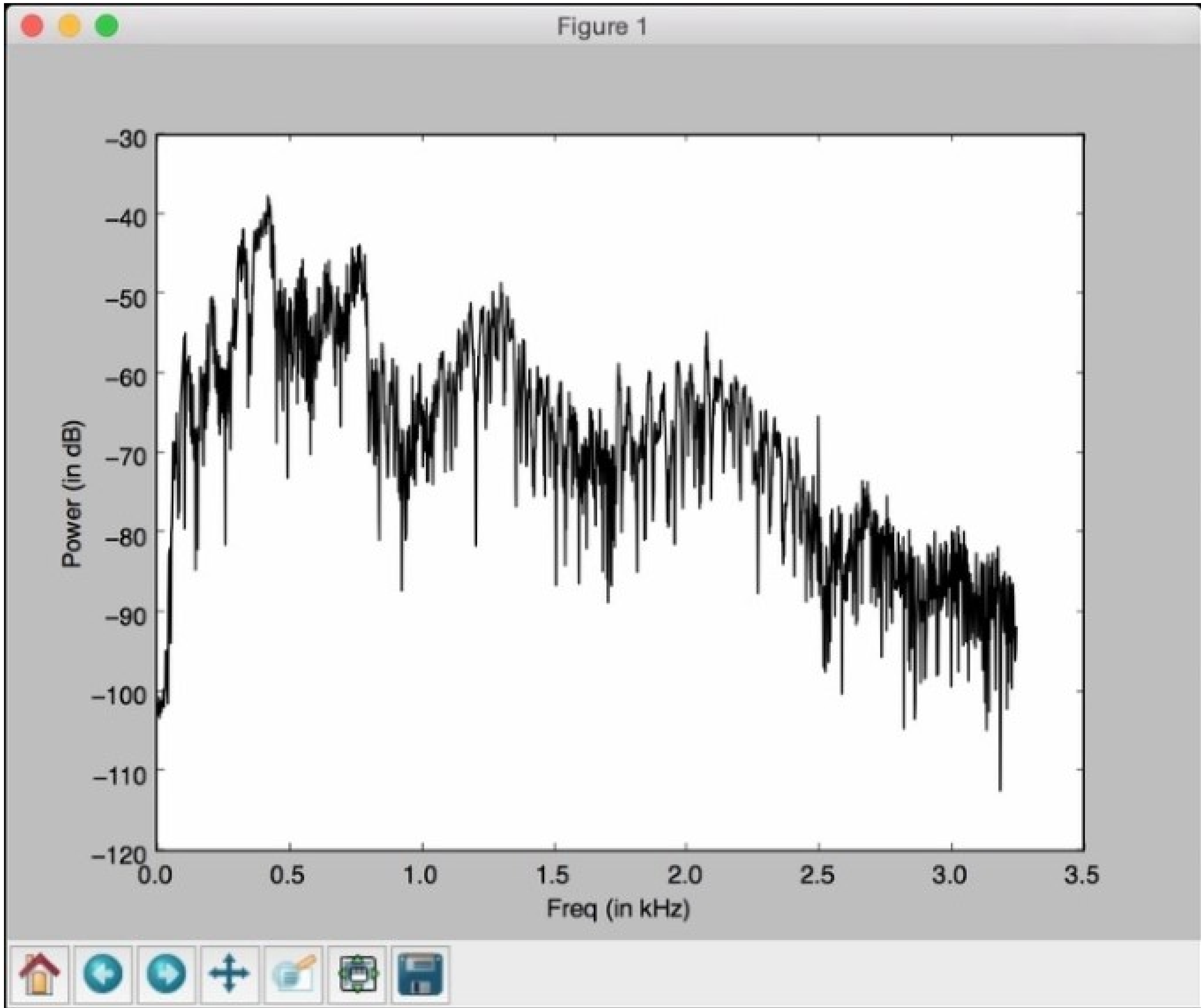


Figure 1
-30
-40
-50
-60
-70
-80
-90
-100
-110
-120
Power (in dB)
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
Freq (in kHz)

Generating audio signals with custom parameters

We can use NumPy to generate audio signals. As we discussed earlier, audio signals are complex mixtures of sinusoids. So, we will keep this in mind when we generate our own audio signal.

How to do it…

1. Create a new Python file, and import the following packages:

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.io.wavfile import write
```

2. We need to define the output file where the generated audio will be stored:

```
# File where the output will be saved
output_file = 'output_generated.wav'
```

3. Let's specify the audio generation parameters. We want to generate a three-second long signal with a sampling frequency of `44100` and a tonal frequency of `587` Hz. The values on the time axis will go from *-2*pi* to *2*pi*:

```
# Specify audio parameters
duration = 3  # seconds
sampling_freq = 44100  # Hz
tone_freq = 587
min_val = -2 * np.pi
max_val = 2 * np.pi
```

4. Let's generate the time axis and the audio signal. The audio signal is a simple sinusoid with the previously mentioned parameters:

```
# Generate audio
t = np.linspace(min_val, max_val, duration * sampling_freq)
audio = np.sin(2 * np.pi * tone_freq * t)
```

5. Let's add some noise to the signal:

```
# Add some noise
noise = 0.4 * np.random.rand(duration * sampling_freq)
audio += noise
```

6. We need to scale the values to 16-bit integers before we store them:

```
# Scale it to 16-bit integer values
scaling_factor = pow(2,15) - 1
```

```
audio_normalized = audio / np.max(np.abs(audio))
audio_scaled = np.int16(audio_normalized * scaling_factor)
```

7. Write this signal to the output file:

```
# Write to output file
write(output_file, sampling_freq, audio_scaled)
```

8. Plot the signal using the first 100 values:

```
# Extract first 100 values for plotting
audio = audio[:100]
```

9. Generate the time axis:

```
# Build the time axis
x_values = np.arange(0, len(audio), 1) / float(sampling_freq)
```

10. Convert the time axis into seconds:

```
# Convert to seconds
x_values *= 1000
```

11. Plot the signal, as follows:

```
# Plotting the chopped audio signal
plt.plot(x_values, audio, color='black')
plt.xlabel('Time (ms)')
plt.ylabel('Amplitude')
plt.title('Audio signal')
plt.show()
```

12. The full code is in the `generate.py` file. If you run this code, you will get the following figure:

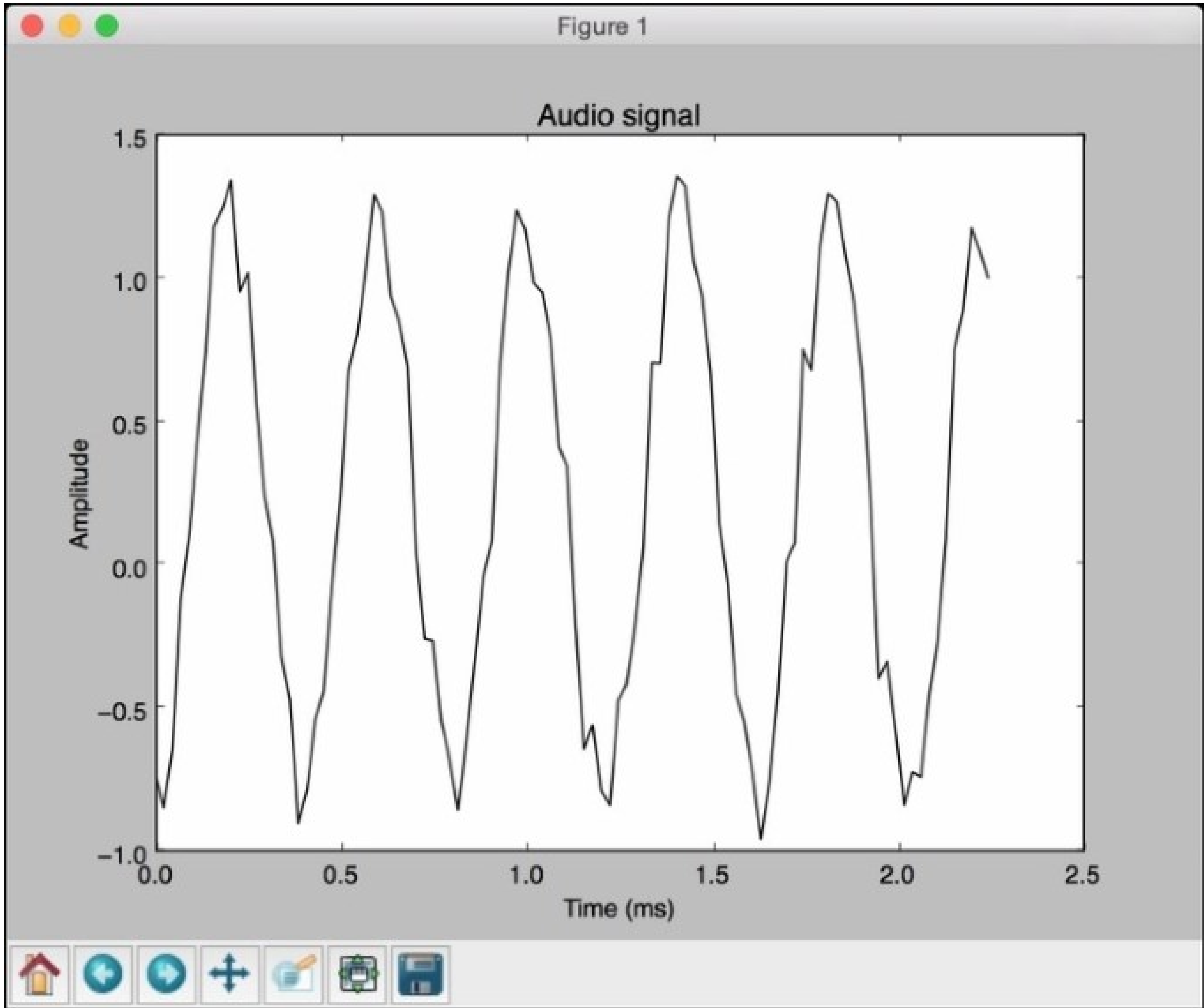


Figure 1
Audio signal
Amplitude
Time (ms)
1.5
1.0
0.5
0.0
−0.5
−1.0
0.0
0.5
1.0
1.5
2.0
2.5

Synthesizing music

Now that we know how to generate audio, let's use this principle to synthesize some music. You can check out this link, http://www.phy.mtu.edu/~suits/notefreqs.html. This link lists various notes, such as *A*, *G*, *D*, and so on, along with their corresponding frequencies. We will use this to generate some simple music.

How to do it…

1. Create a new Python file, and import the following packages:

```
import json
import numpy as np
from scipy.io.wavfile import write
import matplotlib.pyplot as plt
```

2. Define a function to synthesize a tone, based on input parameters:

```
# Synthesize tone
def synthesizer(freq, duration, amp=1.0, sampling_freq=44100):
```

3. Build the time axis values:

```
    # Build the time axis
    t = np.linspace(0, duration, duration * sampling_freq)
```

4. Construct the audio sample using the input arguments, such as amplitude and frequency:

```
    # Construct the audio signal
    audio = amp * np.sin(2 * np.pi * freq * t)

    return audio.astype(np.int16)
```

5. Let's define the `main` function. You have been provided with a JSON file called `tone_freq_map.json`, which contains some notes along with their frequencies:

```
if __name__=='__main__':
    tone_map_file = 'tone_freq_map.json'
```

6. Load that file:

```
    # Read the frequency map
    with open(tone_map_file, 'r') as f:
        tone_freq_map = json.loads(f.read())
```

7. Let's assume that we want to generate a *G* note for a duration of `2` seconds:

```
    # Set input parameters to generate 'G' tone
    input_tone = 'G'
    duration = 2     # seconds
```

```
amplitude = 10000
sampling_freq = 44100    # Hz
```

8. Call the function with the following parameters:

```
# Generate the tone
synthesized_tone = synthesizer(tone_freq_map[input_tone],
duration, amplitude, sampling_freq)
```

9. Write the generated signal into the output file:

```
# Write to the output file
write('output_tone.wav', sampling_freq, synthesized_tone)
```

10. Open this file in a media player and listen to it. That's the *G* note! Let's do something more interesting. Let's generate some notes in sequence to give it a musical feel. Define a note sequence along with their durations in seconds:

```
# Tone-duration sequence
tone_seq = [('D', 0.3), ('G', 0.6), ('C', 0.5), ('A', 0.3),
('Asharp', 0.7)]
```

11. Iterate through this list and call the synthesizer function for each of them:

```
# Construct the audio signal based on the chord sequence
output = np.array([])
for item in tone_seq:
    input_tone = item[0]
    duration = item[1]
    synthesized_tone =
synthesizer(tone_freq_map[input_tone], duration, amplitude,
sampling_freq)
    output = np.append(output, synthesized_tone, axis=0)
```

12. Write the signal to the output file:

```
# Write to the output file
write('output_tone_seq.wav', sampling_freq, output)
```

13. The full code is in the `synthesize_music.py` file. You can open the `output_tone_seq.wav` file in your media player and listen to it. You can feel the music!

Extracting frequency domain features

We discussed earlier how to convert a signal into the frequency domain. In most modern speech recognition systems, people use frequency-domain features. After you convert a signal into the frequency domain, you need to convert it into a usable form. **Mel Frequency Cepstral Coefficients** (**MFCC**) is a good way to do this. MFCC takes the power spectrum of a signal and then uses a combination of filter banks and discrete cosine transform to extract features. If you need a quick refresher, you can check out http://practicalcryptography.com/miscellaneous/machine-learning/guide-mel-frequency-cepstral-coefficients-mfccs. Make sure that the `python_speech_features` package is installed before you start. You can find the installation instructions at http://python-speech-features.readthedocs.org/en/latest. Let's take a look at how to extract MFCC features.

How to do it…

1. Create a new Python file, and import the following packages:

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.io import wavfile
from features import mfcc, logfbank
```

2. Read the `input_freq.wav` input file that is already provided to you:

```
# Read input sound file
sampling_freq, audio = wavfile.read("input_freq.wav")
```

3. Extract the MFCC and filter bank features:

```
# Extract MFCC and Filter bank features
mfcc_features = mfcc(audio, sampling_freq)
filterbank_features = logfbank(audio, sampling_freq)
```

4. Print the parameters to see how many windows were generated:

```
# Print parameters
print '\nMFCC:\nNumber of windows =', mfcc_features.shape[0]
print 'Length of each feature =', mfcc_features.shape[1]
print '\nFilter bank:\nNumber of windows =',
filterbank_features.shape[0]
print 'Length of each feature =', filterbank_features.shape[1]
```

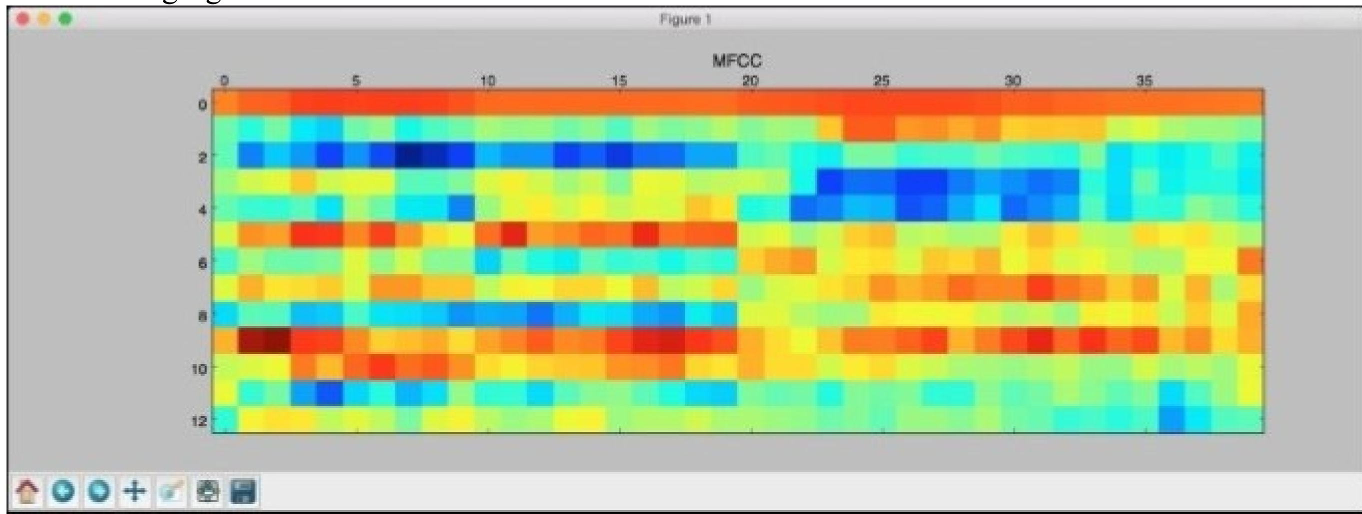
5. Let's visualize the MFCC features. We need to transform the matrix so that the time domain is horizontal:

```
# Plot the features
mfcc_features = mfcc_features.T
plt.matshow(mfcc_features)
plt.title('MFCC')
```

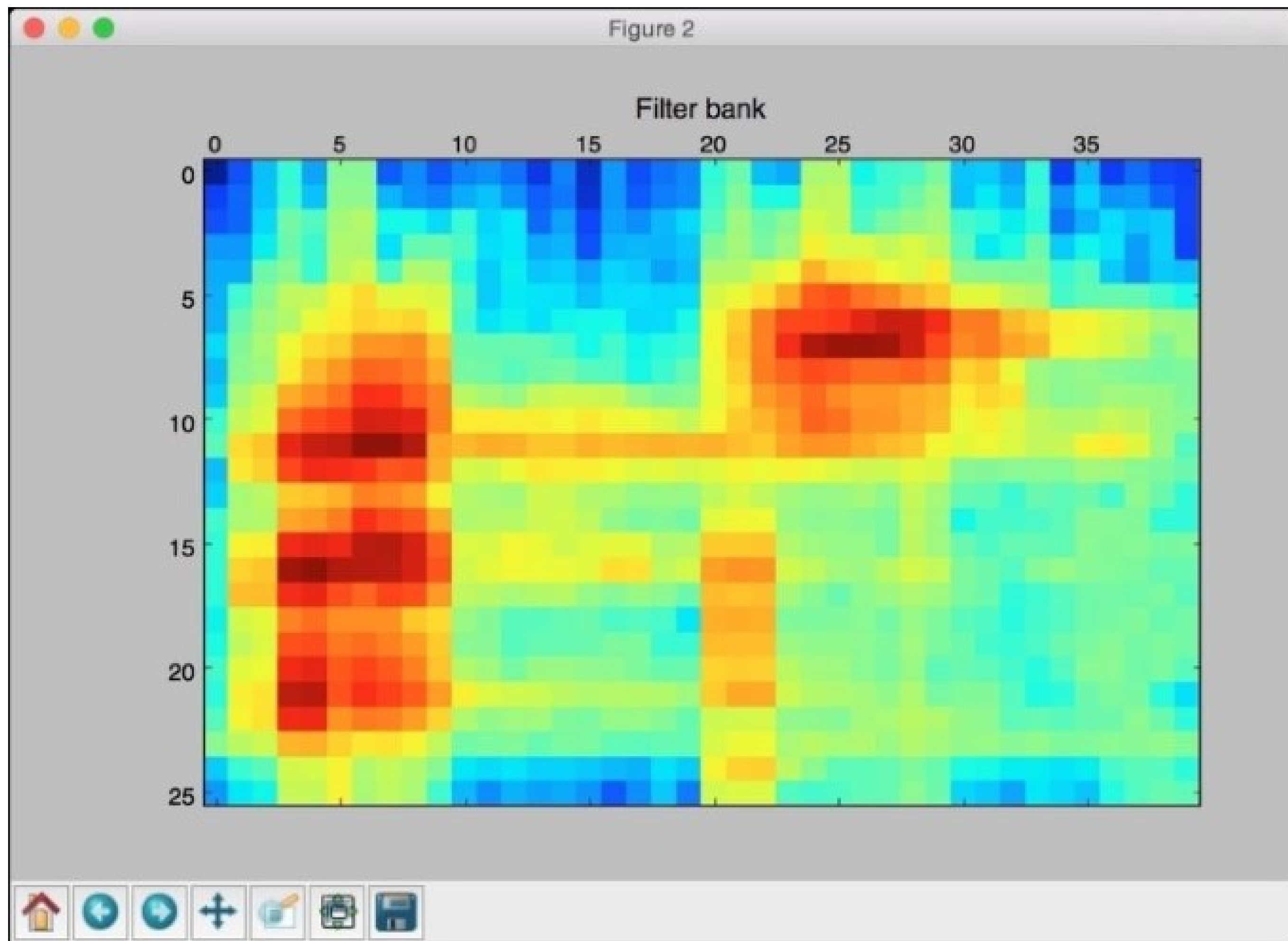
6. Let's visualize the filter bank features. Again, we need to transform the matrix so that the time domain is horizontal:

```
filterbank_features = filterbank_features.T
plt.matshow(filterbank_features)
plt.title('Filter bank')

plt.show()
```

7. The full code is in the `extract_freq_features.py` file. If you run this code, you will get the following figure for MFCC features:



8. The filter bank features will look like the following:



9. You will get the following output on your Terminal:

```
MFCC:
Number of windows = 40
Length of each feature = 13

Filter bank:
Number of windows = 40
Length of each feature = 26
```

Building Hidden Markov Models

We are now ready to discuss speech recognition. We will use **Hidden Markov Models** (**HMMs**) to perform speech recognition. HMMs are great at modeling time series data. As an audio signal is a time series signal, HMMs perfectly suit our needs. An HMM is a model that represents probability distributions over sequences of observations. We assume that the outputs are generated by hidden states. So, our goal is to find these hidden states so that we can model the signal. You can learn more about it at https://www.robots.ox.ac.uk/~vgg/rg/slides/hmm.pdf. Before you proceed, you need to install the `hmmlearn` package. You can find the installation instructions at http://hmmlearn.readthedocs.org/en/latest. Let's take a look at how to build HMMs.

How to do it…

1. Create a new Python file. Let's define a class to model HMMs:

```
# Class to handle all HMM related processing
class HMMTrainer(object):
```

2. Let's initialize the class. We will use Gaussian HMMs to model our data. The `n_components` parameter defines the number of hidden states. The `cov_type` defines the type of covariance in our transition matrix, and `n_iter` indicates the number of iterations it will go through before it stops training:

```
    def __init__(self, model_name='GaussianHMM',
n_components=4, cov_type='diag', n_iter=1000):
```

   The choice of the preceding parameters depends on the problem at hand. You need to have an understanding of your data in order to select these parameters in a smart way.

3. Initialize the variables:

```
        self.model_name = model_name
        self.n_components = n_components
        self.cov_type = cov_type
        self.n_iter = n_iter
        self.models = []
```

4. Define the model with the following parameters:

```
        if self.model_name == 'GaussianHMM':
            self.model =
hmm.GaussianHMM(n_components=self.n_components,
                    covariance_type=self.cov_type,
n_iter=self.n_iter)
        else:
            raise TypeError('Invalid model type')
```

5. The input data is a NumPy array, where each element is a feature vector consisting of *k*-dimensions:

```
# X is a 2D numpy array where each row is 13D
def train(self, X):
    np.seterr(all='ignore')
    self.models.append(self.model.fit(X))
```

6. Define a method to extract the score, based on the model:

```
# Run the model on input data
def get_score(self, input_data):
    return self.model.score(input_data)
```

7. We built a class to handle HMM training and prediction, but we need some data to see it in action. We will use it in the next recipe to build a speech recognizer. The full code is in the `speech_recognizer.py` file.

Building a speech recognizer

We need a database of speech files to build our speech recognizer. We will use the database available at https://code.google.com/archive/p/hmm-speech-recognition/downloads. This contains seven different words, where each word has 15 audio files associated with it. This is a small dataset, but this is sufficient to understand how to build a speech recognizer that can recognize seven different words. We need to build an HMM model for each class. When we want to identify the word in a new input file, we need to run all the models on this file and pick the one with the best score. We will use the HMM class that we built in the previous recipe.

How to do it…

1. Create a new Python file, and import the following packages:

```
import os
import argparse

import numpy as np
from scipy.io import wavfile
from hmmlearn import hmm
from features import mfcc
```

2. Define a function to parse the input arguments in the command line:

```
# Function to parse input arguments
def build_arg_parser():
    parser = argparse.ArgumentParser(description='Trains the
HMM classifier')
    parser.add_argument("--input-folder", dest="input_folder",
required=True,
            help="Input folder containing the audio files in
subfolders")
    return parser
```

3. Define the `main` function, and parse the input arguments:

```
if __name__=='__main__':
    args = build_arg_parser().parse_args()
    input_folder = args.input_folder
```

4. Initiate the variable that will hold all the HMM models:

```
    hmm_models = []
```

5. Parse the input directory that contains all the database's audio files:

```
    # Parse the input directory
    for dirname in os.listdir(input_folder):
```

6. Extract the name of the subfolder:

```
            # Get the name of the subfolder
            subfolder = os.path.join(input_folder, dirname)

            if not os.path.isdir(subfolder):
                continue
```

7. The name of the subfolder is the label of this class. Extract it using the following:

```
            # Extract the label
            label = subfolder[subfolder.rfind('/') + 1:]
```

8. Initialize the variables for training:

```
            # Initialize variables
            X = np.array([])
            y_words = []
```

9. Iterate through the list of audio files in each subfolder:

```
            # Iterate through the audio files (leaving 1 file for
testing in each class)
            for filename in [x for x in os.listdir(subfolder) if
x.endswith('.wav')][:-1]:
```

10. Read each audio file, as follows:

```
                # Read the input file
                filepath = os.path.join(subfolder, filename)
                sampling_freq, audio = wavfile.read(filepath)
```

11. Extract the MFCC features:

```
                # Extract MFCC features
                mfcc_features = mfcc(audio, sampling_freq)
```

12. Keep appending this to the `X` variable:

```
                # Append to the variable X
                if len(X) == 0:
                    X = mfcc_features
                else:
                    X = np.append(X, mfcc_features, axis=0)
```

13. Append the corresponding label too:

```
                # Append the label
                y_words.append(label)
```

14. Once you have extracted features from all the files in the current class, train and save the HMM model. As HMM is a generative model for unsupervised learning, we don't need labels to build HMM models for each class. We explicitly assume that separate HMM models will be built for each class:

```
        # Train and save HMM model
        hmm_trainer = HMMTrainer()
        hmm_trainer.train(X)
        hmm_models.append((hmm_trainer, label))
        hmm_trainer = None
```

15. Get a list of test files that were not used for training:

```
    # Test files
    input_files = [
            'data/pineapple/pineapple15.wav',
            'data/orange/orange15.wav',
            'data/apple/apple15.wav',
            'data/kiwi/kiwi15.wav'
            ]
```

16. Parse the input files, as follows:

```
    # Classify input data
    for input_file in input_files:
```

17. Read in each audio file:

```
        # Read input file
        sampling_freq, audio = wavfile.read(input_file)
```

18. Extract the MFCC features:

```
        # Extract MFCC features
        mfcc_features = mfcc(audio, sampling_freq)
```

19. Define variables to store the maximum score and the output label:

```
        # Define variables
        max_score = None
        output_label = None
```

20. Iterate through all the models and run the input file through each of them:

```
        # Iterate through all HMM models and pick
        # the one with the highest score
        for item in hmm_models:
            hmm_model, label = item
```

21. Extract the score and store the maximum score:

```
                score = hmm_model.get_score(mfcc_features)
                if score > max_score:
                    max_score = score
                    output_label = label
```

22. Print the true and predicted labels:

```
            # Print the output
            print "\nTrue:",
input_file[input_file.find('/')+1:input_file.rfind('/')]
            print "Predicted:", output_label
```

23. The full code is in the `speech_recognizer.py` file. If you run this code, you will see the following on your Terminal:

```
True: pineapple
Predicted: pineapple

True: orange
Predicted: orange

True: apple
Predicted: apple

True: kiwi
Predicted: kiwi
```

Chapter 8. Dissecting Time Series and Sequential Data

In this chapter, we will cover the following recipes:

- Transforming data into the time series format
- Slicing time series data
- Operating on time series data
- Extracting statistics from time series data
- Building Hidden Markov Models for sequential data
- Building Conditional Random Fields for sequential text data
- Analyzing stock market data using Hidden Markov Models

Introduction

Time series data is basically a sequence of measurements that are collected over time. These measurements are taken with respect to a predetermined variable and at regular time intervals. One of the main characteristics of time series data is that the ordering matters!

The list of observations that we collect is ordered on a timeline, and the order in which they appear says a lot about underlying patterns. If you change the order, this would totally change the meaning of the data. Sequential data is a generalized notion that encompasses any data that comes in a sequential form, including time series data.

Our objective here is to build a model that describes the pattern of the time series or any sequence in general. Such models are used to describe important features of the time series pattern. We can use these models to explain how the past might affect the future. We can also use them to see how two datasets can be correlated, to forecast future values, or to control a given variable that is based on some metric.

In order to visualize time series data, we tend to plot it using line charts or bar graphs. Time series data analysis is frequently used in finance, signal processing, weather prediction, trajectory forecasting, predicting earthquakes, or any field where we have to deal with temporal data. The models that we build in time series and sequential data analysis should take into account the ordering of data and extract the relationships between neighbors. Let's go ahead and check out a few recipes to analyze time series and sequential data in Python.

Transforming data into the time series format

We will start by understanding how to convert a sequence of observations into time series data and visualize it. We will use a library called **pandas** to analyze time series data. Make sure that you install pandas before you proceed further. You can find the installation instructions at http://pandas.pydata.org/pandas-docs/stable/install.html.

How to do it…

1. Create a new Python file, and import the following packages:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
```

2. Let's define a function to read an input file that converts sequential observations into time-indexed data:

```
def convert_data_to_timeseries(input_file, column,
verbose=False):
```

3. We will use a text file consisting of four columns. The first column denotes the year, the second column denotes the month, and the third and fourth columns denote data. Let's load this into a NumPy array:

```
    # Load the input file
    data = np.loadtxt(input_file, delimiter=',')
```

4. As this is arranged chronologically, the first row contains the start date and the last row contains the end date. Let's extract the starting and ending dates of this dataset:

```
    # Extract the start and end dates
    start_date = str(int(data[0,0])) + '-' + str(int(data[0,1]))
    end_date = str(int(data[-1,0] + 1)) + '-' +
str(int(data[-1,1] % 12 + 1))
```

5. There is also a verbose mode for this function. So if this is set to true, it will print a few things. Let's print out the start and end dates:

```
    if verbose:
        print "\nStart date =", start_date
        print "End date =", end_date
```

6. Let's create a pandas variable, which contains the date sequence with monthly intervals:

```
    # Create a date sequence with monthly intervals
    dates = pd.date_range(start_date, end_date, freq='M')
```

7. Our next step is to convert the given column into time series data. You can access this data using the month and year (as opposed to index):

```
# Convert the data into time series data
data_timeseries = pd.Series(data[:,column], index=dates)
```

8. Use the verbose mode to print out the first ten elements:

```
if verbose:
    print "\nTime series data:\n", data_timeseries[:10]
```

9. Return the time-indexed variable, as follows:

```
return data_timeseries
```

10. Define the `main` function, as follows:

```
if __name__=='__main__':
```

11. We will use the `data_timeseries.txt` file that is already provided to you:

```
# Input file containing data
input_file = 'data_timeseries.txt'
```

12. Load the third column from this text file and convert it to time series data:

```
    # Load input data
    column_num = 2
    data_timeseries = convert_data_to_timeseries(input_file,
column_num)
```

13. The pandas library provides a nice plotting function that you can run directly on the variable:

```
# Plot the time series data
data_timeseries.plot()
plt.title('Input data')

plt.show()
```

14. The full code is given in the `convert_to_timeseries.py` file that is provided to you. If you run the code, you will see the following image:

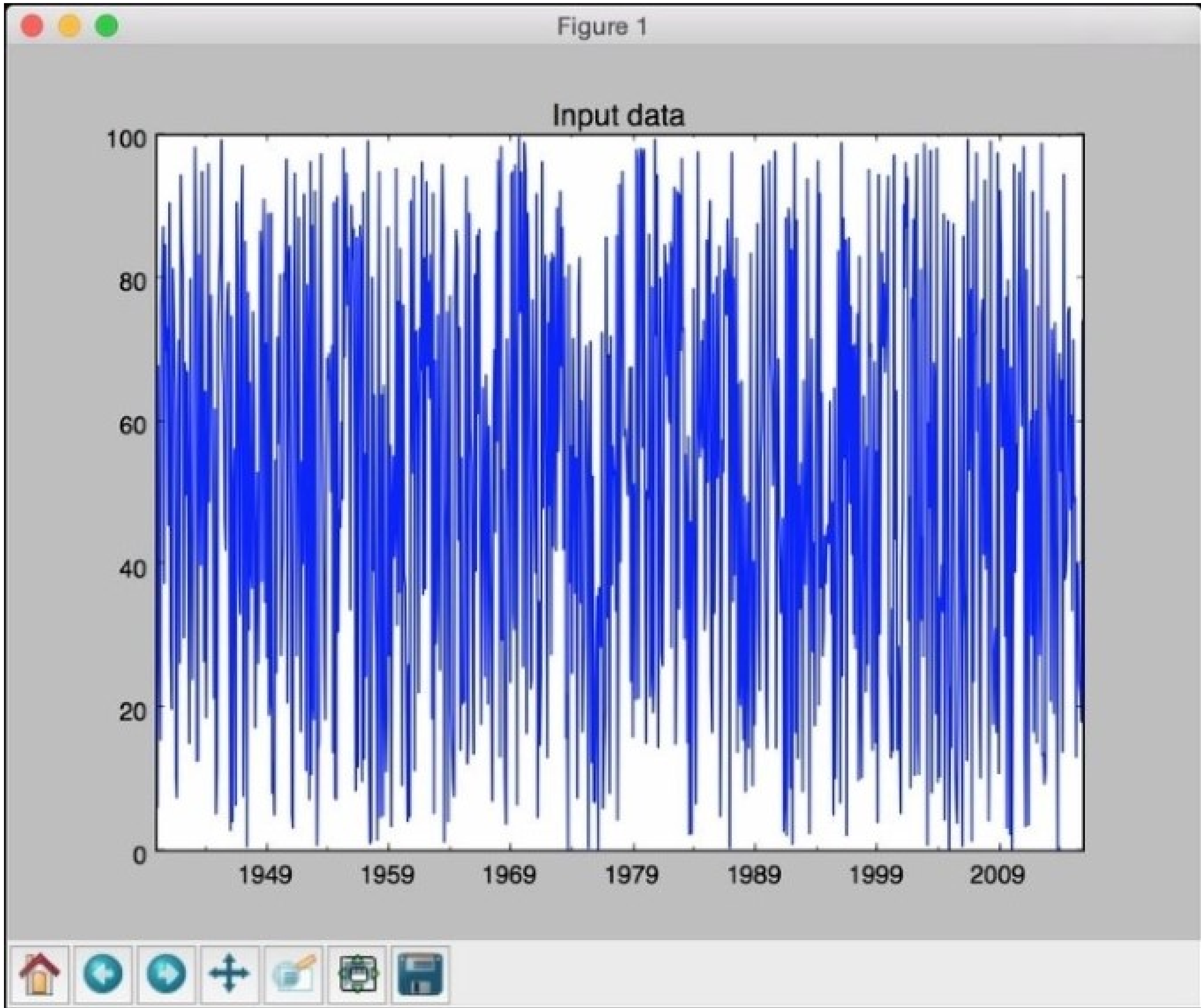


Figure 1
Input data
100
80
60
40
20
0
1949
1959
1969
1979
1989
1999
2009

Slicing time series data

In this recipe, we will learn how to slice time series data using pandas. This will help you extract information from various intervals in the time series data. We will learn how to use dates to handle subsets of our data.

How to do it…

1. Create a new Python file, and import the following packages:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

from convert_to_timeseries import convert_data_to_timeseries
```

2. We will use the same text file that we used in the previous recipe to slice and dice the data:

```
# Input file containing data
input_file = 'data_timeseries.txt'
```

3. We will use the third column again:

```
# Load data
column_num = 2
data_timeseries = convert_data_to_timeseries(input_file,
column_num)
```

4. Let's assume that we want to extract the data between given start and end years. Let's define these, as follows:

```
# Plot within a certain year range
start = '2008'
end = '2015'
```

5. Plot the data between the given year range:

```
plt.figure()
data_timeseries[start:end].plot()
plt.title('Data from ' + start + ' to ' + end)
```

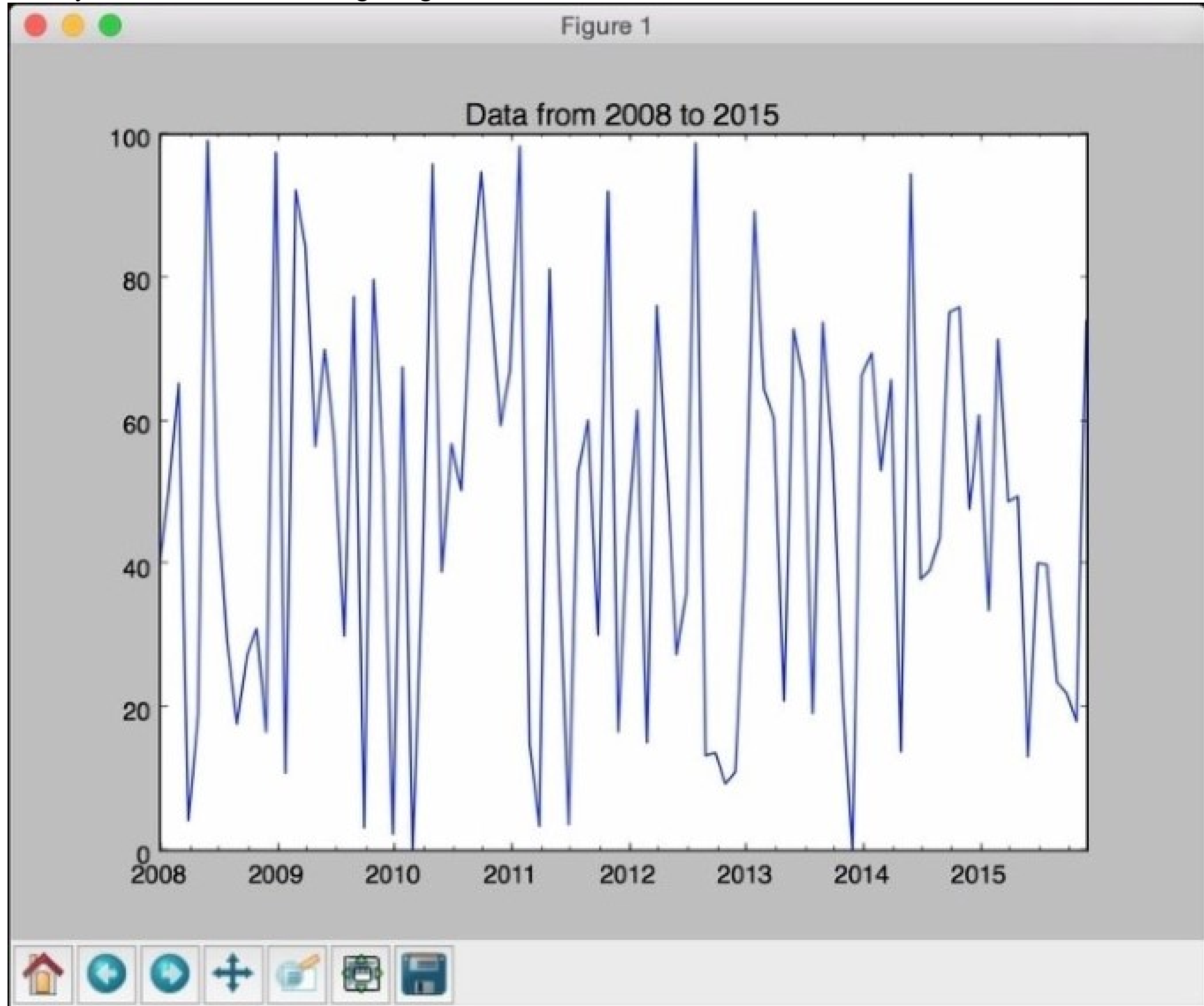
6. We can also slice the data based on a certain range of months:

```
# Plot within a certain range of dates
start = '2007-2'
end = '2007-11'
```

7. Plot the data, as follows:

```
plt.figure()
data_timeseries[start:end].plot()
plt.title('Data from ' + start + ' to ' + end)

plt.show()
```

8. The full code is given in the `slicing_data.py` file that is provided to you. If you run the code, you will see the following image:



9. The next figure will display a smaller time frame; hence, it looks like we have zoomed into it:

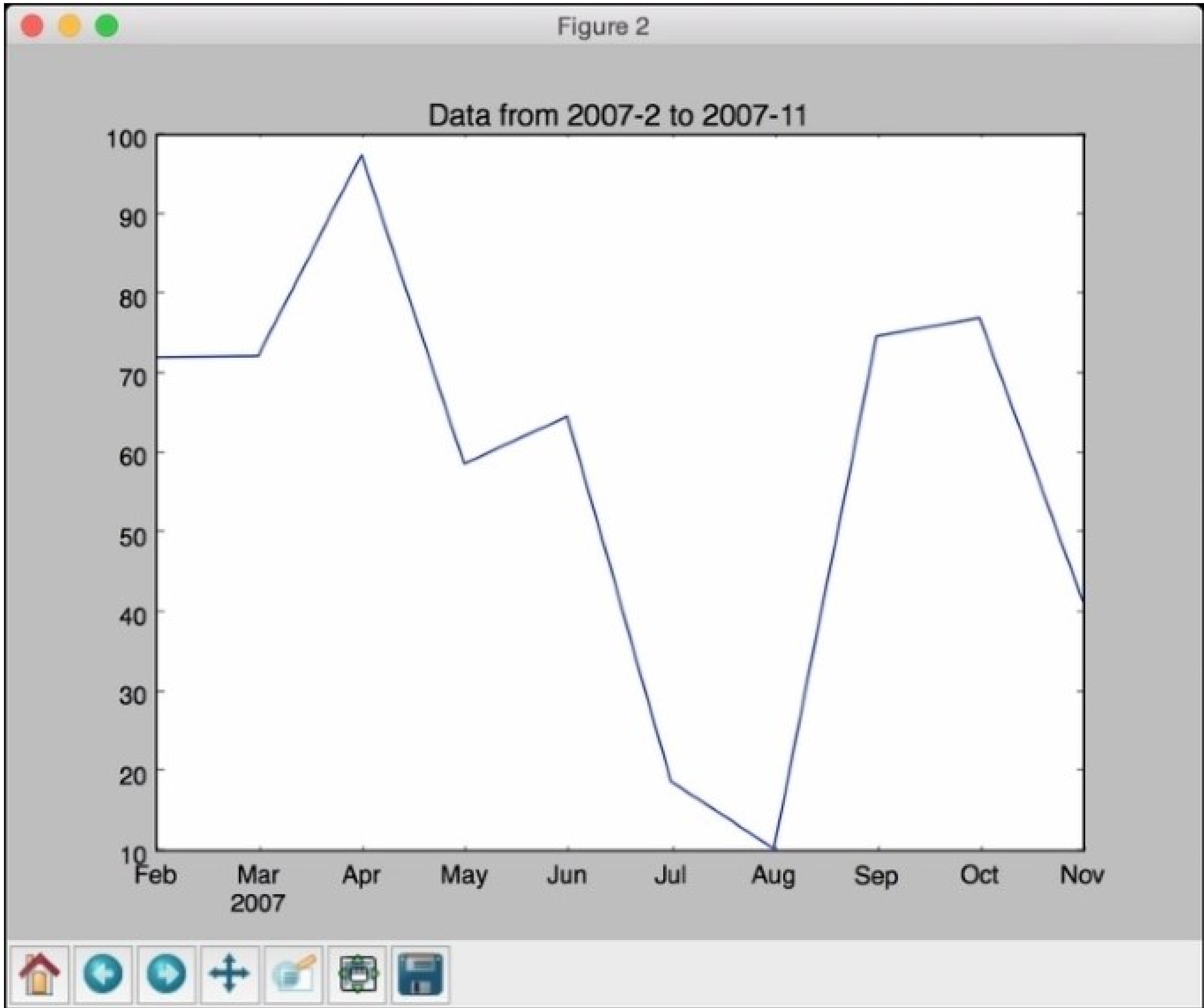


Figure 2
Data from 2007-2 to 2007-11
100
90
80
70
60
50
40
30
20
10
Feb
Mar
2007
Apr
May
Jun
Jul
Aug
Sep
Oct
Nov

Operating on time series data

Now that we know how to slice data and extract various subsets, let's discuss how to operate on time series data. You can filter the data in many different ways. The pandas library allows you to operate on time series data in any way that you want.

How to do it…

1. Create a new Python file, and import the following packages:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

from convert_to_timeseries import convert_data_to_timeseries
```

2. We will use the same text file that we used in the previous recipe:

```
# Input file containing data
input_file = 'data_timeseries.txt'
```

3. We will use both the third and fourth columns in this text file:

```
# Load data
data1 = convert_data_to_timeseries(input_file, 2)
data2 = convert_data_to_timeseries(input_file, 3)
```

4. Convert the data into a pandas data frame:

```
dataframe = pd.DataFrame({'first': data1, 'second': data2})
```

5. Plot the data in the given year range:

```
# Plot data
dataframe['1952':'1955'].plot()
plt.title('Data overlapped on top of each other')
```

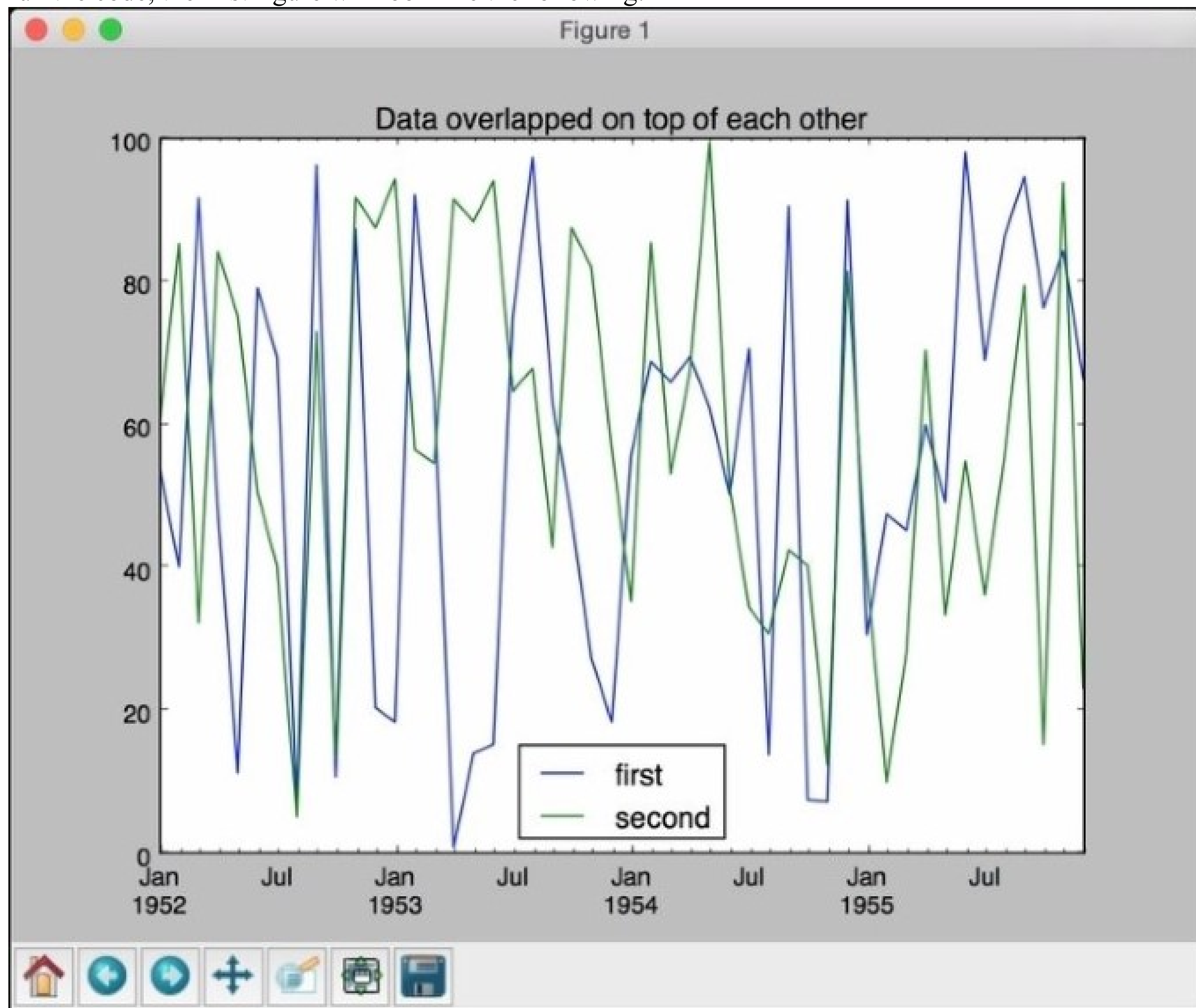
6. Let's assume that we want to plot the difference between the two columns that we just loaded in the given year range. We can do this using the following lines:

```
# Plot the difference
plt.figure()
difference = dataframe['1952':'1955']['first'] -
dataframe['1952':'1955']['second']
difference.plot()
plt.title('Difference (first - second)')
```

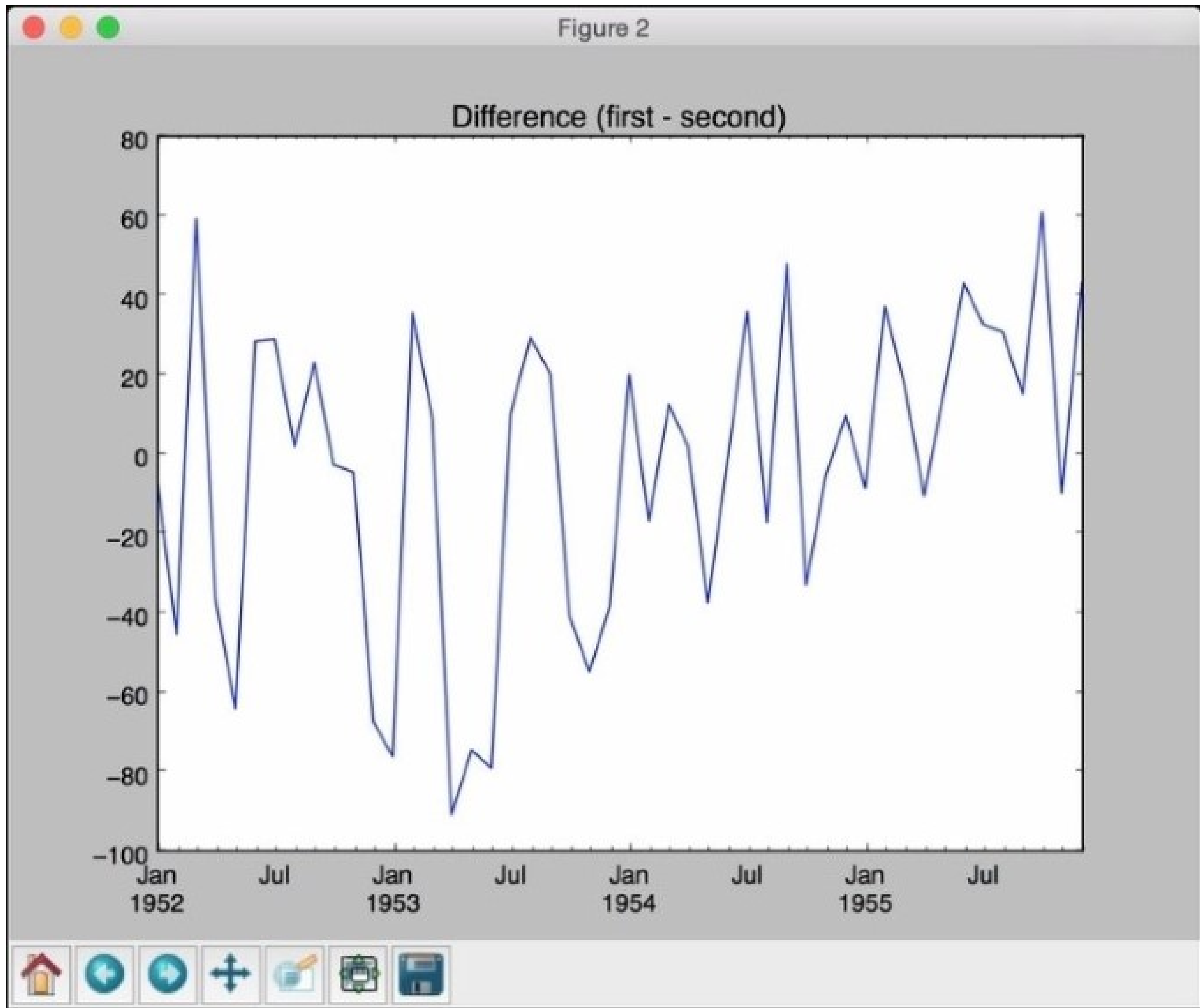
7. If we want to filter the data based on different conditions for the first and second column, we can just specify these conditions and plot this:

```
# When 'first' is greater than a certain threshold
# and 'second' is smaller than a certain threshold
dataframe[(dataframe['first'] > 60) & (dataframe['second'] <
20)].plot()
plt.title('first > 60 and second < 20')

plt.show()
```

8. The full code is in the `operating_on_data.py` file that is already provided to you. If you run the code, the first figure will look like the following:



9. The second output figure denotes the difference, as follows:



10. The third output figure denotes the filtered data, as follows:

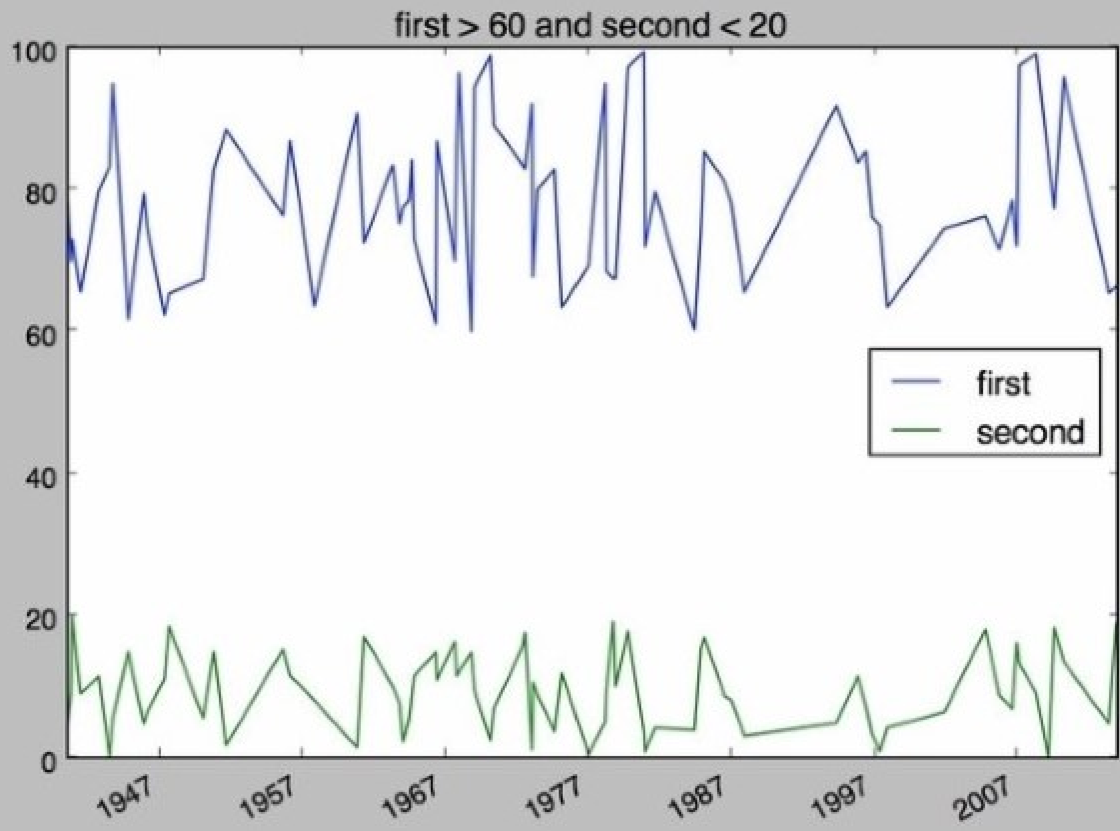
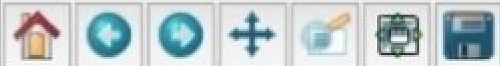


Figure 3
first > 60 and second < 20
100
80
60
40
20
0
1947
1957
1967
1977
1987
1997
2007
first
second


Extracting statistics from time series data

One of the main reasons that we want to analyze time series data is to extract interesting statistics from it. This provides a lot of information regarding the nature of the data. In this recipe, we will take a look at how to extract these stats.

How to do it…

1. Create a new Python file, and import the following packages:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

from convert_to_timeseries import convert_data_to_timeseries
```

2. We will use the same text file that we used in the previous recipes for analysis:

```
# Input file containing data
input_file = 'data_timeseries.txt'
```

3. Load both the data columns (third and fourth columns):

```
# Load data
data1 = convert_data_to_timeseries(input_file, 2)
data2 = convert_data_to_timeseries(input_file, 3)
```

4. Create a pandas data structure to hold this data. This dataframe is like a dictionary that has keys and values:

```
dataframe = pd.DataFrame({'first': data1, 'second': data2})
```

5. Let's start extracting some stats now. To extract the maximum and minimum values, use the following code:

```
# Print max and min
print '\nMaximum:\n', dataframe.max()
print '\nMinimum:\n', dataframe.min()
```

6. To print the mean values of your data or just the row-wise mean, use the following code:

```
# Print mean
print '\nMean:\n', dataframe.mean()
print '\nMean row-wise:\n', dataframe.mean(1)[:10]
```

7. The rolling mean is an important statistic that's used a lot in time series processing. One of the most famous applications is smoothing a signal to remove noise. Rolling mean refers to computing the mean of a signal in a window that keeps sliding on the time scale. Let's consider a window size of `24` and plot this, as follows:

```
# Plot rolling mean
pd.rolling_mean(dataframe, window=24).plot()
```

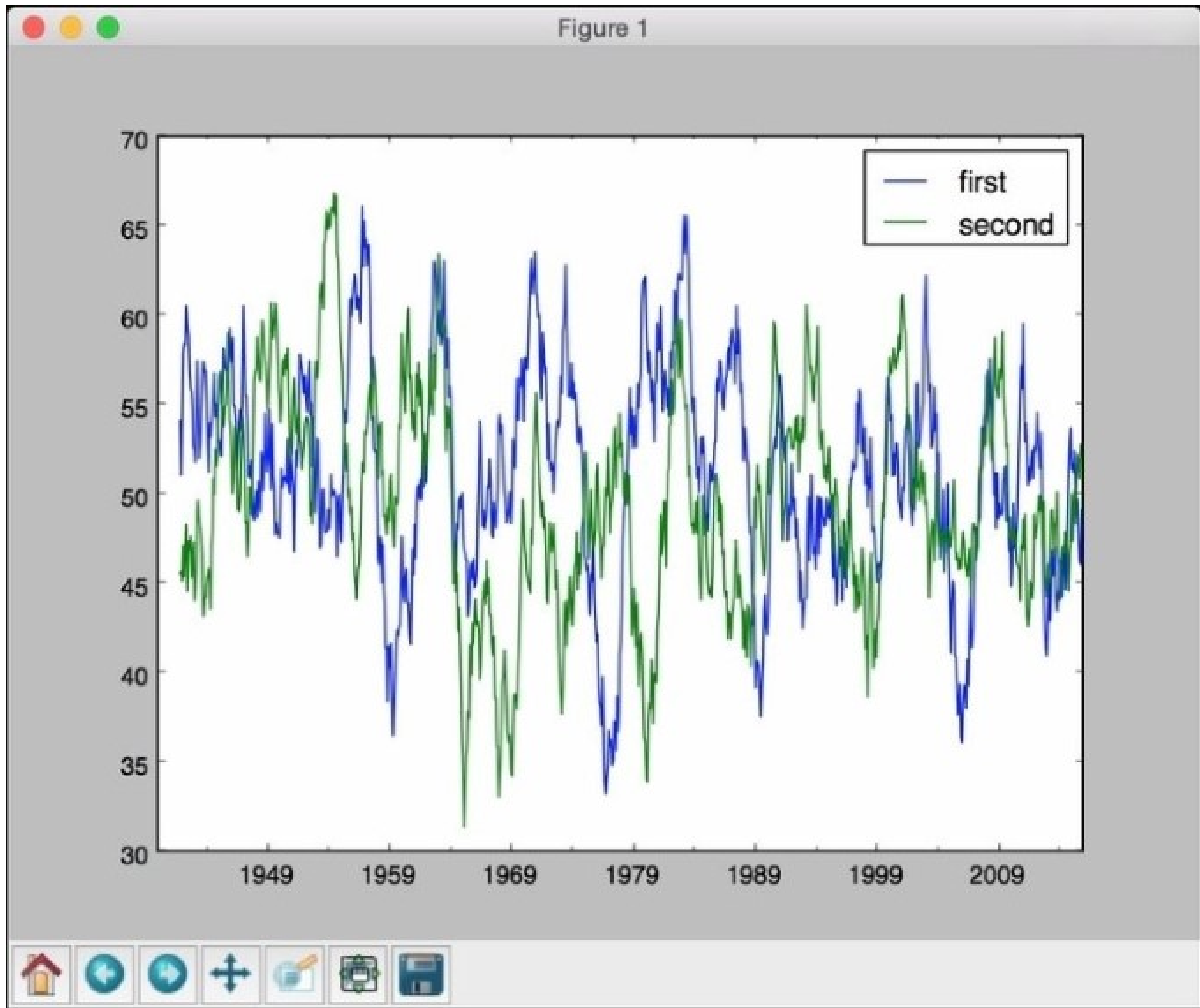
8. Correlation coefficients are useful in understanding the nature of the data, as follows:

```
# Print correlation coefficients
print '\nCorrelation coefficients:\n', dataframe.corr()
```

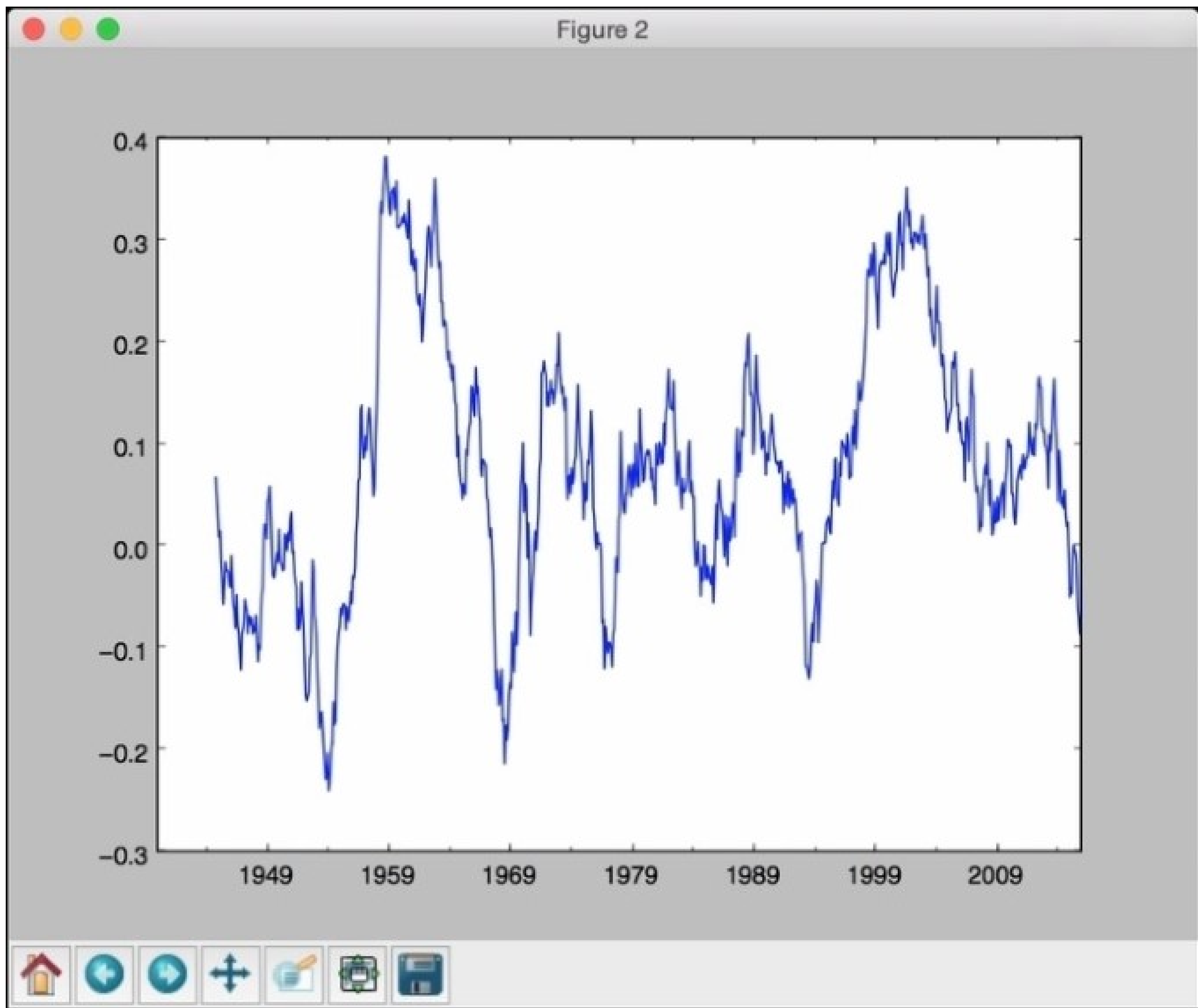
9. Let's plot this using a window size of `60`:

```
# Plot rolling correlation
plt.figure()
pd.rolling_corr(dataframe['first'], dataframe['second'],
window=60).plot()

plt.show()
```

10. The full code is given in the `extract_stats.py` file that is already provided to you. If you run the code, the rolling mean will look like the following:



11. The second output figure indicates the rolling correlation:



12. In the upper half of the Terminal, you will see max, min, and mean values printed, as shown in the following image:

```
Maximum:
first     99.82
second    99.97
dtype: float64

Minimum:
first     0.07
second    0.00
dtype: float64

Mean:
first     51.264529
second    49.695417
dtype: float64
```

13. In the lower half of the Terminal, you will see the row-wise mean stats and correlation coefficients printed, as seen in the following image:

```
Mean row-wise:
1940-01-31    81.885
1940-02-29    41.135
1940-03-31    10.305
1940-04-30    83.545
1940-05-31    18.395
1940-06-30    16.695
1940-07-31    86.875
1940-08-31    42.255
1940-09-30    55.880
1940-10-31    34.720
Freq: M, dtype: float64

Correlation coefficients:
           first    second
first   1.000000  0.077607
second  0.077607  1.000000
```

Building Hidden Markov Models for sequential data

The **Hidden Markov Models** (**HMMs**) are really powerful when it comes to sequential data analysis. They are used extensively in finance, speech analysis, weather forecasting, sequencing of words, and so on. We are often interested in uncovering hidden patterns that appear over time.

Any source of data that produces a sequence of outputs could produce patterns. Note that HMMs are generative models, which means that they can generate the data once they learn the underlying structure. HMMs cannot discriminate between classes in their base forms. This is in contrast to discriminative models that can learn to discriminate between classes but cannot generate data.

Getting ready

For example, let's say that we want to predict whether the weather will be sunny, chilly, or rainy tomorrow. To do this, we look at all the parameters, such as temperature, pressure, and so on, whereas the underlying state is hidden. Here, the underlying state refers to the three available options: sunny, chilly, or rainy. If you wish to learn more about HMMs, check out this tutorial at https://www.robots.ox.ac.uk/~vgg/rg/slides/hmm.pdf.

We will use `hmmlearn` to build and train HMMs. Make sure that you install this before you proceed. You can find the installation instructions at http://hmmlearn.readthedocs.org/en/latest.

How to do it…

1. Create a new Python file, and import the following packages:

```
import datetime

import numpy as np
import matplotlib.pyplot as plt
from hmmlearn.hmm import GaussianHMM

from convert_to_timeseries import convert_data_to_timeseries
```

2. We will use the data from a file named `data_hmm.txt` that is already provided to you. This file contains comma-separated lines. Each line contains three values: a year, a month, and a floating point data. Let's load this into a NumPy array:

```
# Load data from input file
input_file = 'data_hmm.txt'
data = np.loadtxt(input_file, delimiter=',')
```

3. Let's stack the data column-wise for analysis. We don't need to technically column-stack this because it's only one column. However, if you had more than one column to analyze, you can use this structure:

```
# Arrange data for training
X = np.column_stack([data[:,2]])
```

4. Create and train the HMM using four components. The number of components is a hyperparameter that we have to choose. Here, by selecting four, we say that the data is being generated using four underlying states. We will see how the performance varies with this parameter soon:

```
# Create and train Gaussian HMM
print "\nTraining HMM...."
num_components = 4
model = GaussianHMM(n_components=num_components,
covariance_type="diag", n_iter=1000)
model.fit(X)
```

5. Run the predictor to get the hidden states:

```
# Predict the hidden states of HMM
hidden_states = model.predict(X)
```

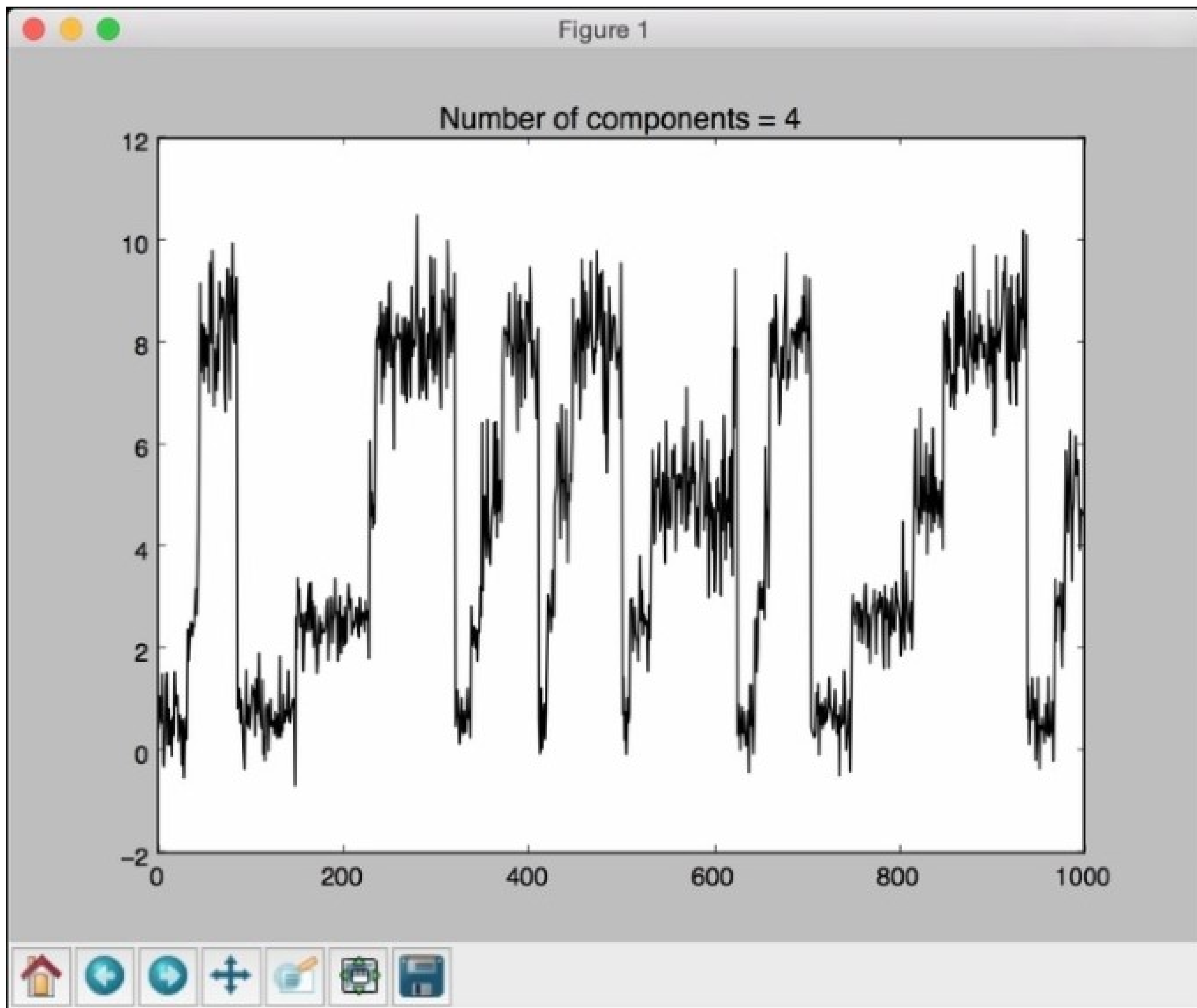
6. Compute the mean and variance of the hidden states:

```
print "\nMeans and variances of hidden states:"
for i in range(model.n_components):
    print "\nHidden state", i+1
    print "Mean =", round(model.means_[i][0], 3)
    print "Variance =", round(np.diag(model.covars_[i])[0], 3)
```

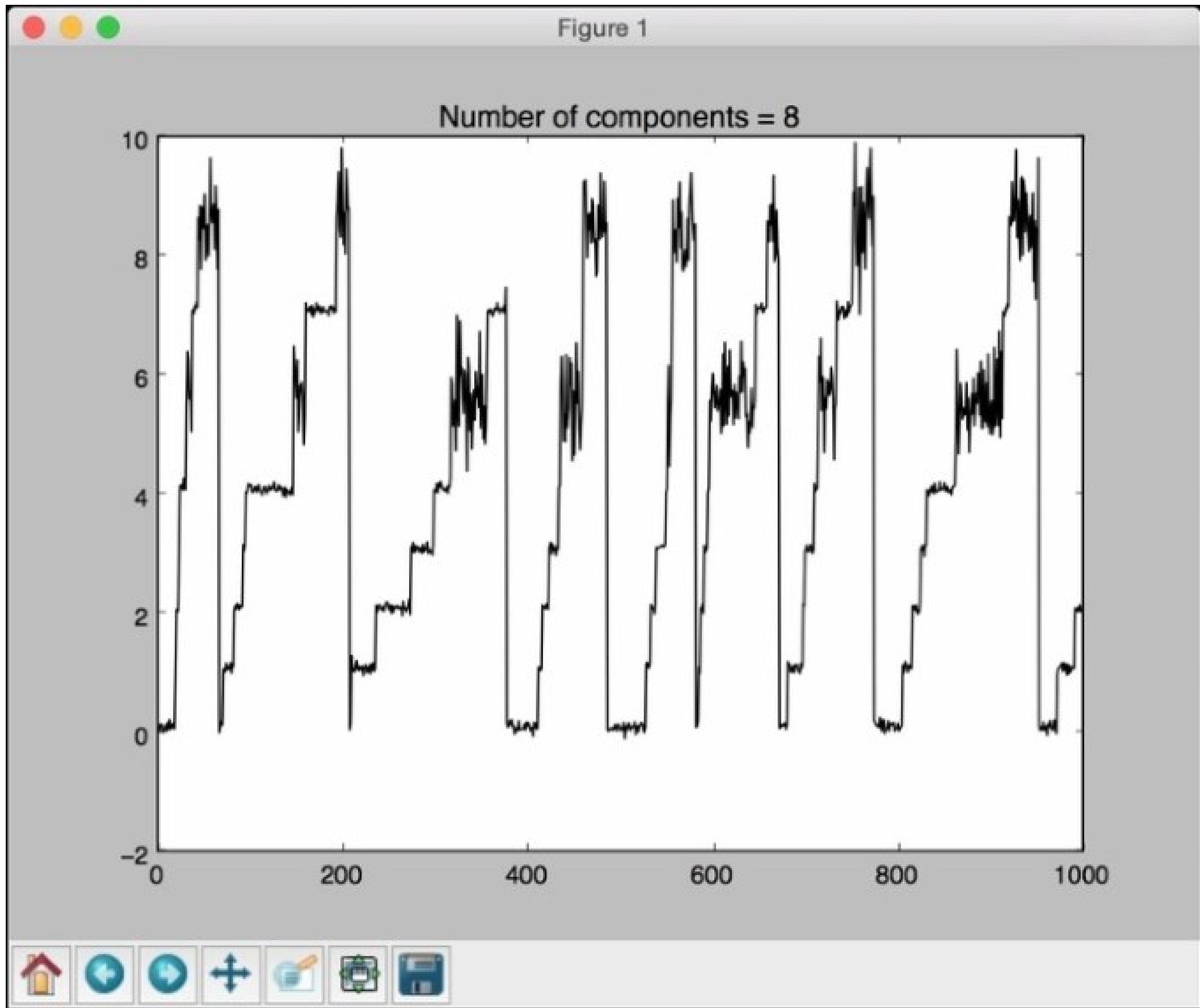
7. As we discussed earlier, HMMs are generative models. So, let's generate, for example, `1000` samples and plot this:

```
# Generate data using model
num_samples = 1000
samples, _ = model.sample(num_samples)
plt.plot(np.arange(num_samples), samples[:,0], c='black')
plt.title('Number of components = ' + str(num_components))

plt.show()
```

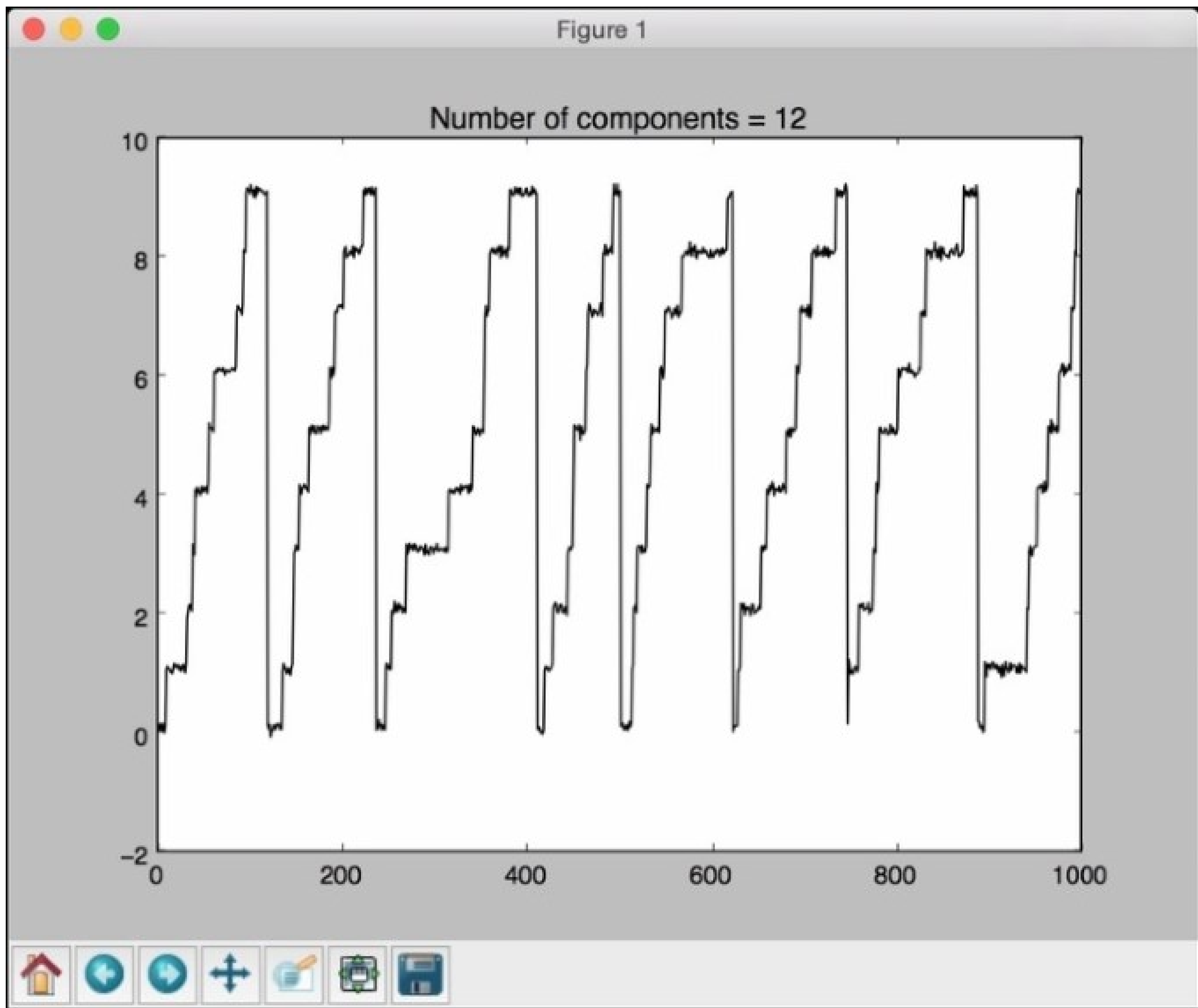
8. The full code is given in the `hmm.py` file that is already provided to you. If you run the code, you will see the following figure:



9. You can experiment with the `n_components` parameter to see how the curve gets nicer as you increase it. You can basically give it more freedom to train and customize by allowing a larger number of hidden states. If you increase it to `8`, you will see the following figure:



10. If you increase this to `12`, it will get even smoother:



11. In the Terminal, you will get the following output:

```
Training HMM....

Means and variances of hidden states:

Hidden state 1
Mean = 5.092
Variance = 0.677

Hidden state 2
Mean = 0.6
Variance = 0.254

Hidden state 3
Mean = 8.099
Variance = 0.678

Hidden state 4
Mean = 2.601
Variance = 0.257
```

Building Conditional Random Fields for sequential text data

The **Conditional Random Fields** (**CRFs**) are probabilistic models used to analyze structured data. They are frequently used to label and segment sequential data. CRFs are discriminative models as opposed to HMMs, which are generative models. CRFs are used extensively to analyze sequences, stocks, speech, words, and so on. In these models, given a particular labeled observation sequence, we define a conditional probability distribution over this sequence. This is in contrast with HMMs where we define a joint distribution over the label and the observed sequence.

Getting ready

HMMs assume that the current output is statistically independent of the previous outputs. This is needed by HMMs to ensure that the inference works in a robust way. However, this assumption need not always be true! The current output in a time series setup, more often than not, depends on previous outputs. One of the main advantages of CRFs over HMMs is that they are conditional by nature, which means that we are not assuming any independence between output observations. There are a few other advantages of using CRFs over HMMs. CRFs tend to outperform HMMs in a number of applications, such as linguistics, bioinformatics, speech analysis, and so on. In this recipe, we will learn how to use CRFs to analyze sequences of letters.

We will use a library called `pystruct` to build and train CRFs. Make sure that you install this before you proceed. You can find the installation instructions at https://pystruct.github.io/installation.html.

How to do it…

1. Create a new Python file, and import the following packages:

```
import os
import argparse
import cPickle as pickle

import numpy as np
import matplotlib.pyplot as plt
from pystruct.datasets import load_letters
from pystruct.models import ChainCRF
from pystruct.learners import FrankWolfeSSVM
```

2. Define an argument parser to take the `C` value as an input argument. `C` is a hyperparameter that controls how specific you want your model to be without losing the power to generalize:

```
def build_arg_parser():
    parser = argparse.ArgumentParser(description='Trains the
CRF classifier')
    parser.add_argument("--c-value", dest="c_value",
required=False, type=float,
```

```
            default=1.0, help="The C value that will be used
for training")
    return parser
```

3. Define a class to handle all CRF-related processing:

```
class CRFTrainer(object):
```

4. Define an `init` function to initialize the values:

```
    def __init__(self, c_value, classifier_name='ChainCRF'):
        self.c_value = c_value
        self.classifier_name = classifier_name
```

5. We will use chain CRF to analyze the data. We need to add an error check for this, as follows:

```
        if self.classifier_name == 'ChainCRF':
            model = ChainCRF()
```

6. Define the classifier that we will use with our CRF model. We will use a type of **Support Vector Machine** to achieve this:

```
            self.clf = FrankWolfeSSVM(model=model,
C=self.c_value, max_iter=50)
        else:
            raise TypeError('Invalid classifier type')
```

7. Load the letters dataset. This dataset consists of segmented letters and their associated feature vectors. We will not analyze the images because we already have the feature vectors. The first letter from each word has been removed, so all we have are lowercase letters:

```
    def load_data(self):
        letters = load_letters()
```

8. Load the data and labels into their respective variables:

```
        X, y, folds = letters['data'], letters['labels'],
letters['folds']
        X, y = np.array(X), np.array(y)
        return X, y, folds
```

9. Define a training method, as follows:

```
    # X is a numpy array of samples where each sample
    # has the shape (n_letters, n_features)
    def train(self, X_train, y_train):
        self.clf.fit(X_train, y_train)
```

10. Define a method to evaluate the performance of the model:

```
    def evaluate(self, X_test, y_test):
        return self.clf.score(X_test, y_test)
```

11. Define a method to classify new data:

```
    # Run the classifier on input data
    def classify(self, input_data):
        return self.clf.predict(input_data)[0]
```

12. The letters are indexed in a numbered array. In order to check the output and make it readable, we need to transform these numbers into alphabets. Define a function to do this:

```
def decoder(arr):

    alphabets = 'abcdefghijklmnopqrstuvwxyz'
    output = ''
    for i in arr:
        output += alphabets[i]

    return output
```

13. Define the `main` function and parse the input arguments:

```
if __name__=='__main__':
    args = build_arg_parser().parse_args()
    c_value = args.c_value
```

14. Initialize the variable with the class and the `C` value:

```
    crf = CRFTrainer(c_value)
```

15. Load the letters data:

```
    X, y, folds = crf.load_data()
```

16. Separate the data into training and testing datasets:

```
    X_train, X_test = X[folds == 1], X[folds != 1]
    y_train, y_test = y[folds == 1], y[folds != 1]
```

17. Train the CRF model, as follows:

```
    print "\nTraining the CRF model..."
    crf.train(X_train, y_train)
```

18. Evaluate the performance of the CRF model:

```
    score = crf.evaluate(X_test, y_test)
    print "\nAccuracy score =", str(round(score*100, 2)) + '%'
```

19. Let's take a random test vector and predict the output using the model:

```
print "\nTrue label =", decoder(y_test[0])
predicted_output = crf.classify([X_test[0]])
print "Predicted output =", decoder(predicted_output)
```

20. The full code is given in the `crf.py` file that is already provided to you. If you run this code, you will get the following output on your Terminal. As we can see, the word is supposed to be "commanding". The CRF does a pretty good job of predicting all the letters:

```
Training the CRF model...

Accuracy score = 78.05%

True label = ommanding
Predicted output = ommanging
```

Analyzing stock market data using Hidden Markov Models

Let's analyze stock market data using Hidden Markov Models. Stock market data is a good example of time series data where the data is organized in the form of dates. In the dataset that we will use, we can see how the stock values of various companies fluctuate over time. Hidden Markov Models are generative models that are used to analyze such time series data. In this recipe, we will use these models to analyze stock values.

How to do it…

1. Create a new Python file, and import the following packages:

   ```
   import datetime

   import numpy as np
   import matplotlib.pyplot as plt
   from matplotlib.finance import quotes_historical_yahoo_ochl
   from hmmlearn.hmm import GaussianHMM
   ```

2. Get the stock quotes from Yahoo finance. There is a method available in `matplotlib` to load this directly:

   ```
   # Get quotes from Yahoo finance
   quotes = quotes_historical_yahoo_ochl("INTC",
           datetime.date(1994, 4, 5), datetime.date(2015, 7, 3))
   ```

3. There are six values in each quote. Let's extract the relevant data such as the closing value of the stock and the volume of stock that is traded along with their corresponding dates:

   ```
   # Extract the required values
   dates = np.array([quote[0] for quote in quotes], dtype=np.int)
   closing_values = np.array([quote[2] for quote in quotes])
   volume_of_shares = np.array([quote[5] for quote in quotes])[1:]
   ```

4. Let's compute the percentage change in the closing value of each type of data. We will use this as one of the features:

   ```
   # Take diff of closing values and computing rate of change
   diff_percentage = 100.0 * np.diff(closing_values) /
   closing_values[:-1]

   dates = dates[1:]
   ```

5. Stack the two arrays column-wise for training:

```
# Stack the percentage diff and volume values column-wise for
training
X = np.column_stack([diff_percentage, volume_of_shares])
```

6. Train the HMM using five components:

```
# Create and train Gaussian HMM
print "\nTraining HMM...."
model = GaussianHMM(n_components=5, covariance_type="diag",
n_iter=1000)

model.fit(X)
```

7. Generate `500` samples using the trained HMM and plot this, as follows:

```
# Generate data using model
num_samples = 500
samples, _ = model.sample(num_samples)
plt.plot(np.arange(num_samples), samples[:,0], c='black')

plt.show()
```

8. The full code is given in `hmm_stock.py` that is already provided to you. If you run this code, you will see the following figure:

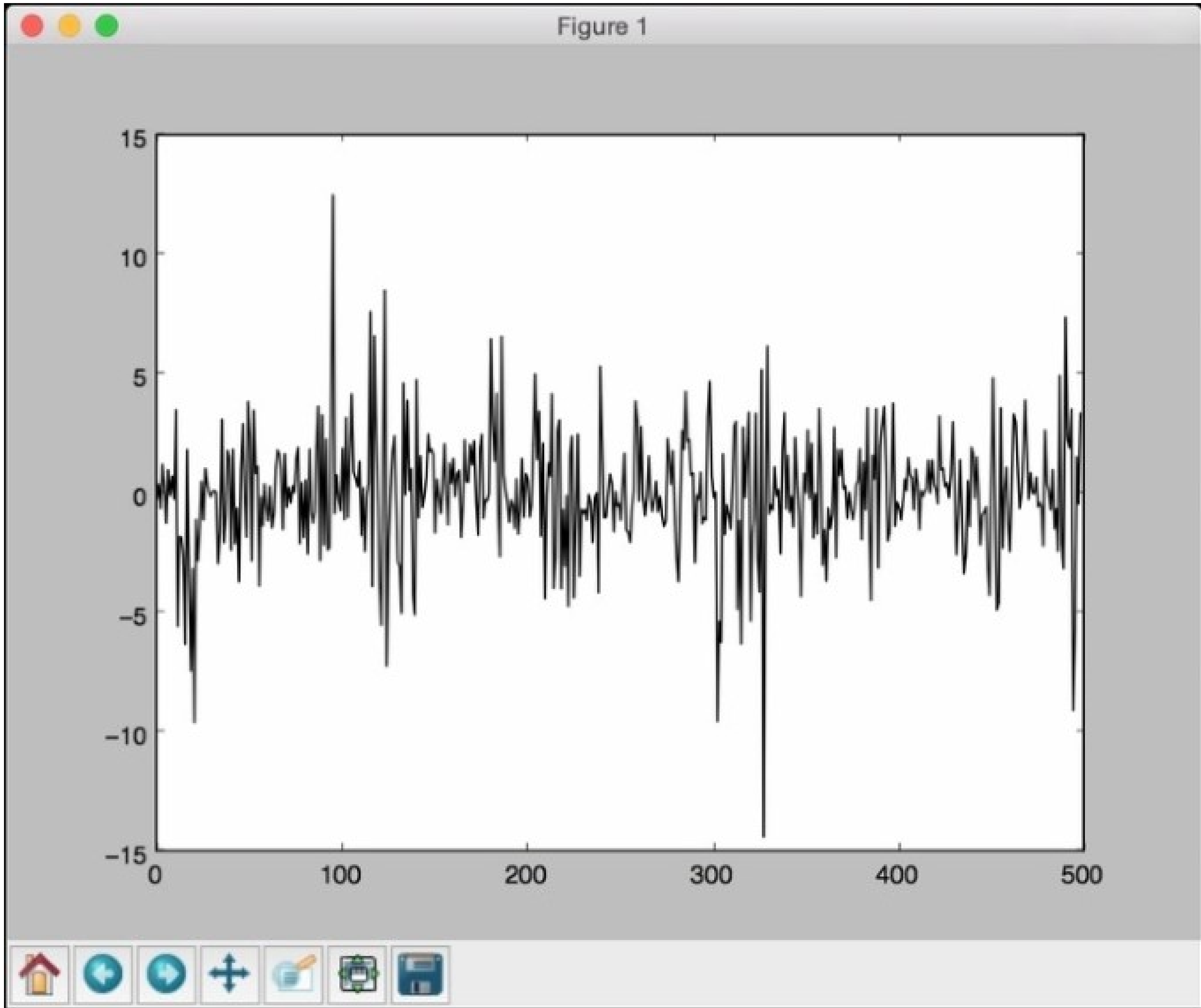


Figure 1
15
10
5
0
−5
−10
−15
0
100
200
300
400
500

Chapter 9. Image Content Analysis

In this chapter, we will cover the following recipes:

- Operating on images using OpenCV-Python
- Detecting edges
- Histogram equalization
- Detecting corners
- Detecting SIFT feature points
- Building Star feature detector
- Creating features using visual codebook and vector quantization
- Training an image classifier using Extremely Random Forests
- Building an object recognizer

Introduction

Computer Vision is a field that studies how to process, analyze, and understand the contents of visual data. In image content analysis, we use a lot of Computer Vision algorithms to build our understanding of the objects in the image. Computer Vision covers various aspects of image analysis, such as object recognition, shape analysis, pose estimation, 3D modeling, visual search, and so on. Humans are really good at identifying and recognizing things around them! The ultimate goal of Computer Vision is to accurately model the human vision system using computers.

Computer Vision consists of various levels of analysis. In low-level vision, we deal with pixel processing tasks, such as edge detection, morphological processing, and optical flow. In middle-level and high-level vision, we deal with things, such as object recognition, 3D modeling, motion analysis, and various other aspects of visual data. As we go higher, we tend to delve deeper into the conceptual aspects of our visual system and try to extract a description of visual data, based on activities and intentions. One thing to note is that higher levels tend to rely on the outputs of the lower levels for analysis.

One of the most common questions here is, "How is Computer Vision different from Image Processing?" Image Processing studies image transformations at the pixel level. Both the input and output of an Image Processing system are images. Some common examples are edge detection, histogram equalization, or image compression. Computer Vision algorithms heavily rely on Image Processing algorithms to perform their duties. In Computer Vision, we deal with more complex things that include understanding the visual data at a conceptual level. The reason for this is because we want to construct meaningful descriptions of the objects in the images. The output of a Computer Vision system is an interpretation of the 3D scene in the given image. This interpretation can come in various forms, depending on the task at hand.

In this chapter, we will use a library, called **OpenCV**, to analyze images. OpenCV is the world's most popular library for Computer Vision. As it has been highly optimized for many different platforms, it has become the de facto standard in the industry. Before you proceed, make sure that you install the library with Python support. You can download and install OpenCV at http://opencv.org. For detailed installation instructions on various operating systems, you can refer to the documentation section on the website.

Operating on images using OpenCV-Python

Let's take a look at how to operate on images using OpenCV-Python. In this recipe, we will see how to load and display an image. We will also look at how to crop, resize, and save an image to an output file.

How to do it…

1. Create a new Python file, and import the following packages:

   ```
   import sys

   import cv2
   import numpy as np
   ```

2. Specify the input image as the first argument to the file, and read it using the image read function. We will use `forest.jpg`, as follows:

   ```
   # Load and display an image -- 'forest.jpg'
   input_file = sys.argv[1]
   img = cv2.imread(input_file)
   ```

3. Display the input image, as follows:

   ```
   cv2.imshow('Original', img)
   ```

4. We will now crop this image. Extract the height and width of the input image, and then specify the boundaries:

   ```
   # Cropping an image
   h, w = img.shape[:2]
   start_row, end_row = int(0.21*h), int(0.73*h)
   start_col, end_col= int(0.37*w), int(0.92*w)
   ```

5. Crop the image using NumPy style slicing and display it:

   ```
   img_cropped = img[start_row:end_row, start_col:end_col]
   cv2.imshow('Cropped', img_cropped)
   ```

6. Resize the image to `1.3` times its original size and display it:

   ```
   # Resizing an image
   scaling_factor = 1.3
   img_scaled = cv2.resize(img, None, fx=scaling_factor,
   fy=scaling_factor,
   interpolation=cv2.INTER_LINEAR)
   cv2.imshow('Uniform resizing', img_scaled)
   ```

7. The previous method will uniformly scale the image on both dimensions. Let's assume that we want to skew the image based on specific output dimensions. We use the following code:

```
img_scaled = cv2.resize(img, (250, 400),
interpolation=cv2.INTER_AREA)
cv2.imshow('Skewed resizing', img_scaled)
```

8. Save the image to an output file:

```
# Save an image
output_file = input_file[:-4] + '_cropped.jpg'
cv2.imwrite(output_file, img_cropped)

cv2.waitKey()
```

9. The `waitKey()` function displays the images until you hit a key on the keyboard.
10. The full code is given in the `operating_on_images.py` file that is already provided to you. If you run the code, you will see the following input image:

11. The second output is the cropped image:

12. The third output is the uniformly resized image:

Uniform resizing

13. The fourth output is the skewed image:

Skewed resizing

Detecting edges

Edge detection is one of the most popular techniques in Computer Vision. It is used as a preprocessing step in many applications. Let's look at how to use different edge detectors to detect edges in the input image.

How to do it…

1. Create a new Python file, and import the following packages:

```
import sys

import cv2
import numpy as np
```

2. Load the input image. We will use `chair.jpg`:

```
# Load the input image -- 'chair.jpg'
# Convert it to grayscale
input_file = sys.argv[1]
img = cv2.imread(input_file, cv2.IMREAD_GRAYSCALE)
```

3. Extract the height and width of the image:

```
h, w = img.shape
```

4. **Sobel filter** is a type of edge detector that uses a *3x3* kernel to detect horizontal and vertical edges separately. You can learn more about it at http://www.tutorialspoint.com/dip/sobel_operator.htm. Let's start with the horizontal detector:

```
sobel_horizontal = cv2.Sobel(img, cv2.CV_64F, 1, 0, ksize=5)
```

5. Run the vertical Sobel detector:

```
sobel_vertical = cv2.Sobel(img, cv2.CV_64F, 0, 1, ksize=5)
```

6. **Laplacian edge detector** detects edges in both the directions. You can learn more about it at http://homepages.inf.ed.ac.uk/rbf/HIPR2/log.htm. We use it as follows:

```
laplacian = cv2.Laplacian(img, cv2.CV_64F)
```

7. Even though Laplacian addresses the shortcomings of Sobel, the output is still very noisy. **Canny edge detector** outperforms all of them because of the way it treats the problem. It is a multistage process, and it uses hysteresis to come up with clean edges. You can learn more about it at http://homepages.inf.ed.ac.uk/rbf/HIPR2/canny.htm:

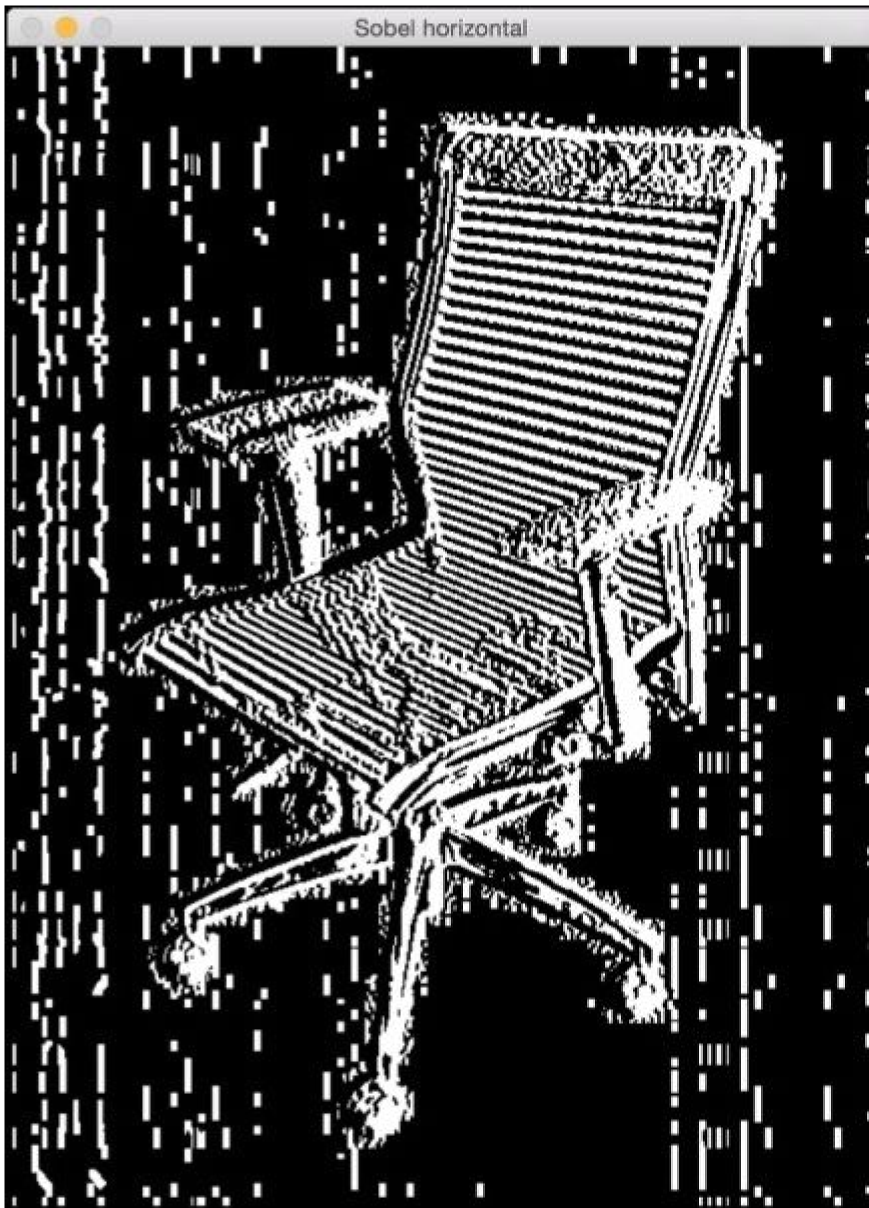
```
canny = cv2.Canny(img, 50, 240)
```

8. Display all the output images:

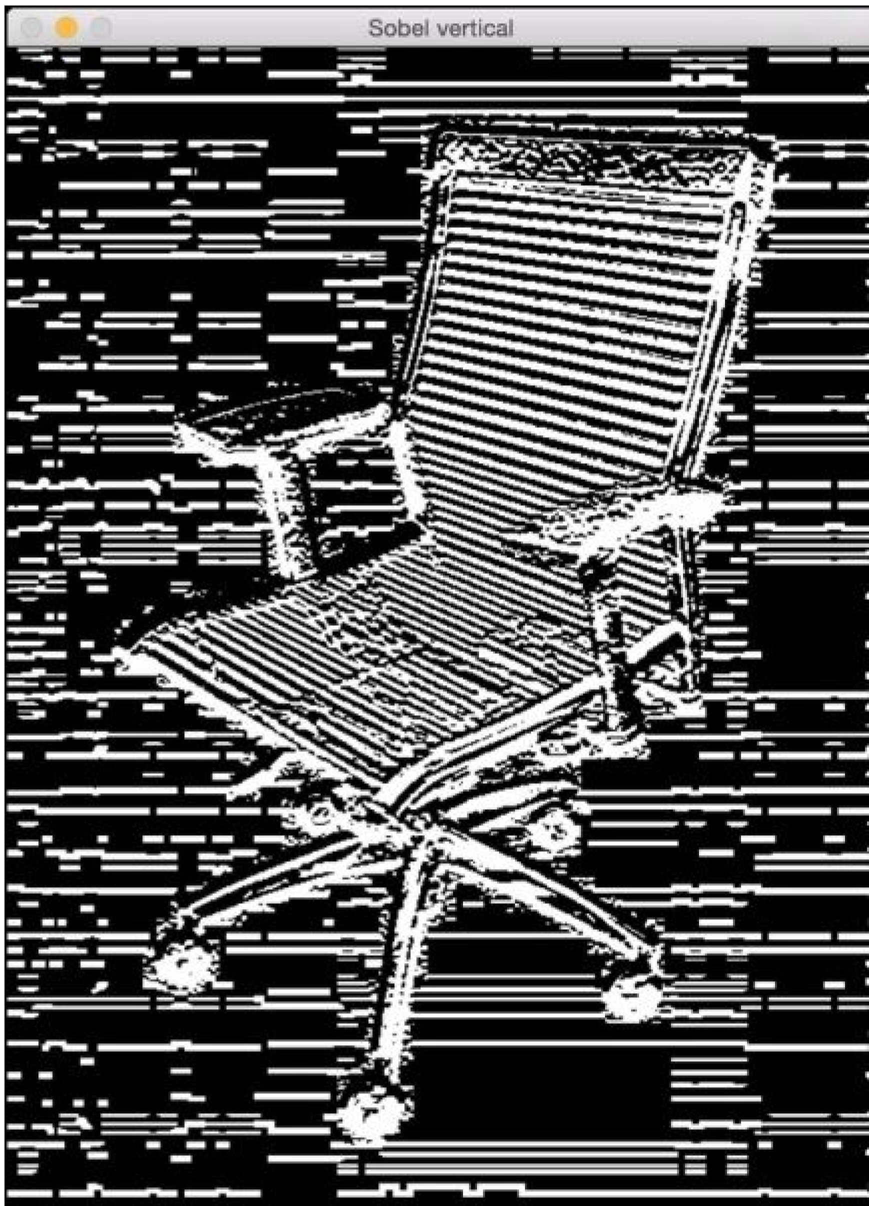
```
cv2.imshow('Original', img)
cv2.imshow('Sobel horizontal', sobel_horizontal)
cv2.imshow('Sobel vertical', sobel_vertical)
cv2.imshow('Laplacian', laplacian)
cv2.imshow('Canny', canny)

cv2.waitKey()
```

9. The full code is given in the `edge_detector.py` file that is already provided to you. The original input image looks like the following:

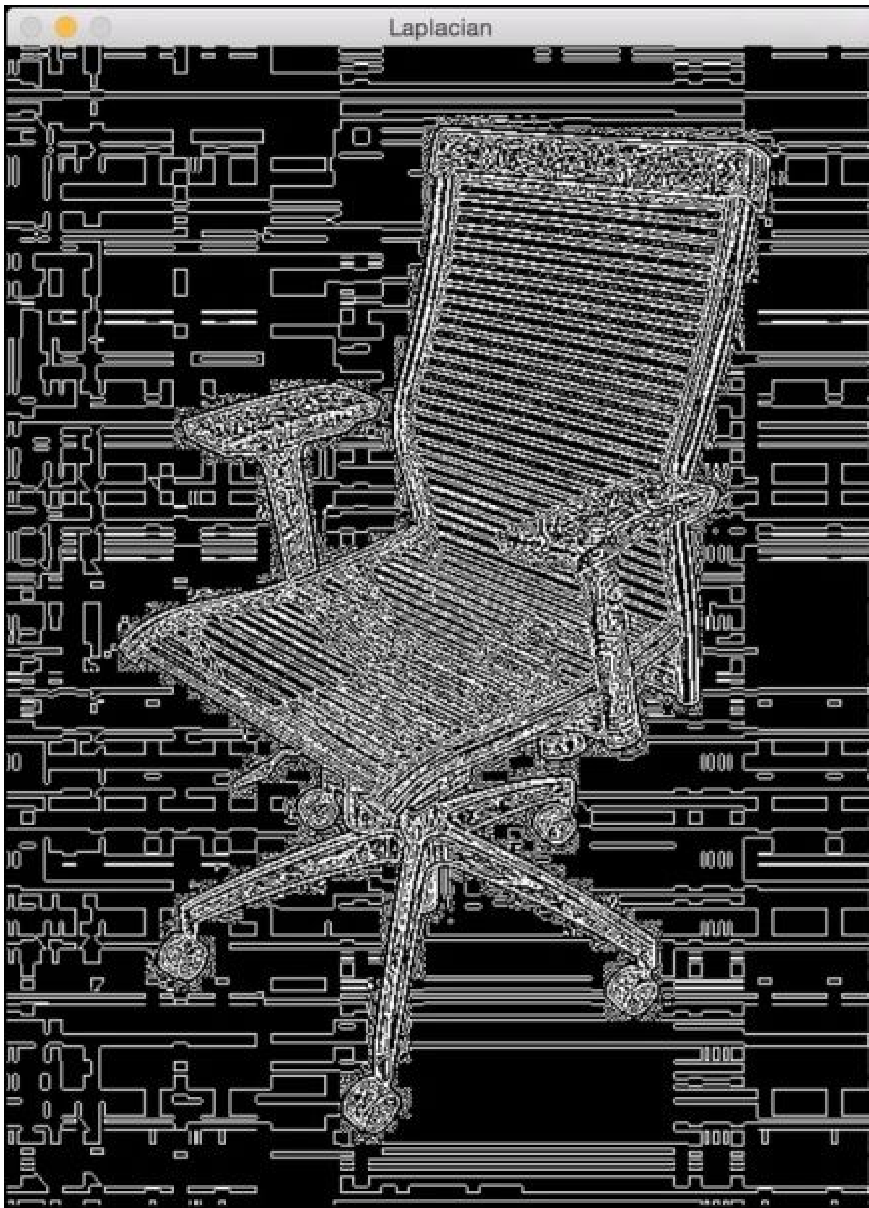
10. Here is the horizontal Sobel edge detector output. Note how the detected lines tend to be vertical. This is due the fact that it's a horizontal edge detector, and it tends to detect changes in this direction:



11. The vertical Sobel edge detector output looks like the following image:



12. Here is the Laplacian edge detector output:



13. Canny edge detector detects all the edges nicely, as shown in the following image:

Canny

Histogram equalization

Histogram equalization is the process of modifying the intensities of the image pixels to enhance the contrast. The human eye likes contrast! This is the reason that almost all camera systems use histogram equalization to make images look nice. The interesting thing is that the histogram equalization process is different for grayscale and color images. There's a catch when dealing with color images, and we'll see it in this recipe. Let's see how to do it.

How to do it…

1. Create a new Python file, and import the following packages:

```
import sys

import cv2
import numpy as np
```

2. Load the input image. We will use the image, `sunrise.jpg`:

```
# Load input image -- 'sunrise.jpg'
input_file = sys.argv[1]
img = cv2.imread(input_file)
```

3. Convert the image to grayscale and display it:

```
# Convert it to grayscale
img_gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
cv2.imshow('Input grayscale image', img_gray)
```

4. Equalize the histogram of the grayscale image and display it:

```
# Equalize the histogram
img_gray_histeq = cv2.equalizeHist(img_gray)
cv2.imshow('Histogram equalized - grayscale', img_gray_histeq)
```

5. In order to equalize the histogram of the color images, we need to follow a different procedure. Histogram equalization only applies to the intensity channel. An RGB image consists of three color channels, and we cannot apply the histogram equalization process on these channels separately. We need to separate the intensity information from the color information before we do anything. So, we convert it to YUV colorspace first, equalize the Y channel, and then convert it back to RGB to get the output. You can learn more about YUV colorspace at http://softpixel.com/~cwright/programming/colorspace/yuv. OpenCV loads images in the BGR format by default, so let's convert it from BGR to YUV first:

```
# Histogram equalization of color images
img_yuv = cv2.cvtColor(img, cv2.COLOR_BGR2YUV)
```

6. Equalize the Y channel, as follows:

```
img_yuv[:,:,0] = cv2.equalizeHist(img_yuv[:,:,0])
```

7. Convert it back to BGR:

```
img_histeq = cv2.cvtColor(img_yuv, cv2.COLOR_YUV2BGR)
```

8. Display the input and output images:

```
cv2.imshow('Input color image', img)
cv2.imshow('Histogram equalized - color', img_histeq)

cv2.waitKey()
```

9. The full code is given in the `histogram_equalizer.py` file that is already provided to you. The input image is shown, as follows:

10. The histogram equalized image looks like the following:

Histogram equalized - grayscale

Detecting corners

Corner detection is an important process in Computer Vision. It helps us identify the salient points in the image. This was one of the earliest feature extraction techniques that was used to develop image analysis systems.

How to do it…

1. Create a new Python file, and import the following packages:

```
import sys

import cv2
import numpy as np
```

2. Load the input image. We will use `box.png`:

```
# Load input image -- 'box.png'
input_file = sys.argv[1]
img = cv2.imread(input_file)
cv2.imshow('Input image', img)
```

3. Convert the image to grayscale and cast it to floating point values. We need the floating point values for the corner detector to work:

```
img_gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
img_gray = np.float32(img_gray)
```

4. Run the **Harris corner detector** function on the grayscale image. You can learn more about Harris corner detector at http://docs.opencv.org/3.0-beta/doc/py_tutorials/py_feature2d/py_features_harris/py_features_harris.html:

```
# Harris corner detector
img_harris = cv2.cornerHarris(img_gray, 7, 5, 0.04)
```

5. In order to mark the corners, we need to dilate the image, as follows:

```
# Resultant image is dilated to mark the corners
img_harris = cv2.dilate(img_harris, None)
```

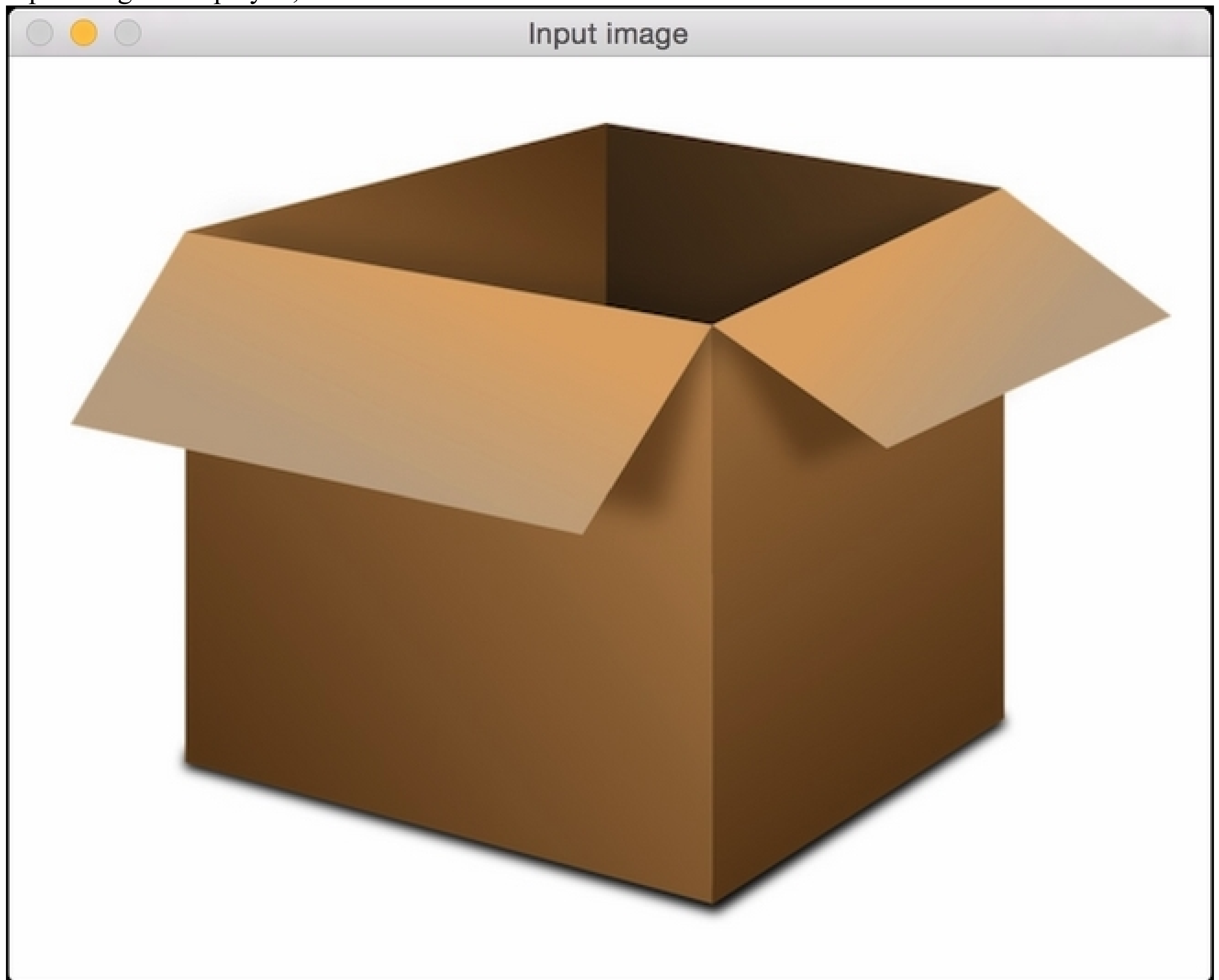
6. Let's threshold the image to display the important points:

```
# Threshold the image
img[img_harris > 0.01 * img_harris.max()] = [0, 0, 0]
```

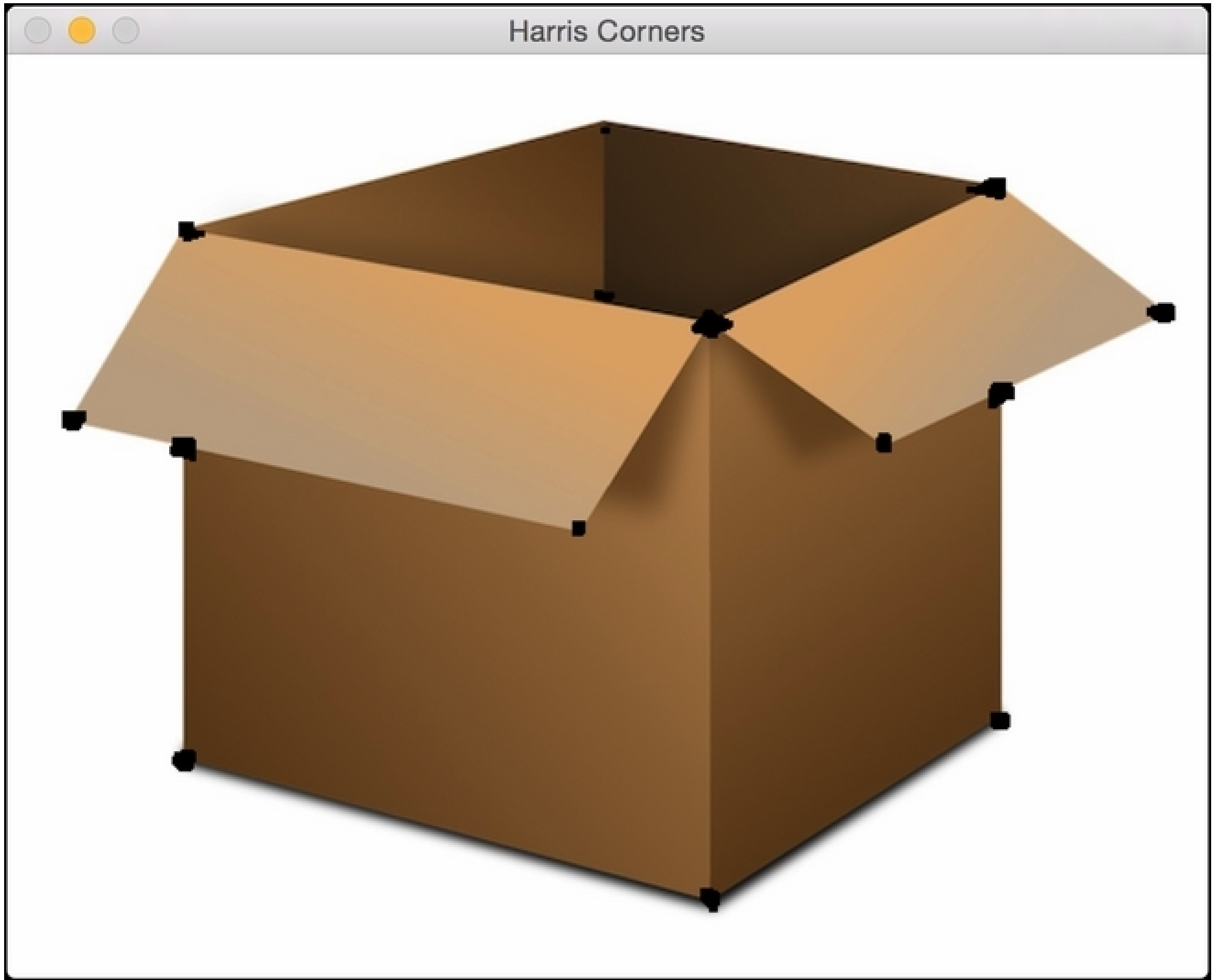
7. Display the output image:

```
cv2.imshow('Harris Corners', img)
cv2.waitKey()
```

8. The full code is given in the `corner_detector.py` file that is already provided to you. The input image is displayed, as follows:



9. The output image after detecting corners is as follows:


Harris Corners

Detecting SIFT feature points

Scale Invariant Feature Transform (**SIFT**) is one of the most popular features in the field of Computer Vision. David Lowe first proposed this in his seminal paper, which is available at https://www.cs.ubc.ca/~lowe/papers/ijcv04.pdf. It has since become one of the most effective features to use for image recognition and content analysis. It is robust against scale, orientation, intensity, and so on. This forms the basis of our object recognition system. Let's take a look at how to detect these feature points.

How to do it…

1. Create a new Python file, and import the following packages:

```
import sys

import cv2
import numpy as np
```

2. Load the input image. We will use `table.jpg`:

```
# Load input image -- 'table.jpg'
input_file = sys.argv[1]
img = cv2.imread(input_file)
```

3. Convert this image to grayscale:

```
img_gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
```

4. Initialize the SIFT detector object and extract the keypoints:

```
sift = cv2.xfeatures2d.SIFT_create()
keypoints = sift.detect(img_gray, None)
```

5. The keypoints are the salient points, but they are not the features. This basically gives us the location of the salient points. SIFT also functions as a very effective feature extractor, but we will see this aspect of it in one of the later recipes.
6. Draw the keypoints on top of the input image, as follows:

```
img_sift = np.copy(img)
cv2.drawKeypoints(img, keypoints, img_sift,
flags=cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)
```

7. Display the input and output images:

```
cv2.imshow('Input image', img)
cv2.imshow('SIFT features', img_sift)
cv2.waitKey()
```

8. The full code is given in the `feature_detector.py` file that is already provided to you. The input image is as follows:

9. The output image looks like the following:

SIFT features

Building a Star feature detector

SIFT feature detector is good in many cases. However, when we build object recognition systems, we may want to use a different feature detector before we extract features using SIFT. This will give us the flexibility to cascade different blocks to get the best possible performance. So, we will use the **Star feature detector** in this case to see how to do it.

How to do it…

1. Create a new Python file, and import the following packages:

```
import sys

import cv2
import numpy as np
```

2. Define a class to handle all the functions that are related to Star feature detection:

```
class StarFeatureDetector(object):
    def __init__(self):
        self.detector = cv2.xfeatures2d.StarDetector_create()
```

3. Define a function to run the detector on the input image:

```
    def detect(self, img):
        return self.detector.detect(img)
```

4. Load the input image in the `main` function. We will use `table.jpg`:

```
if __name__=='__main__':
    # Load input image -- 'table.jpg'
    input_file = sys.argv[1]
    input_img = cv2.imread(input_file)
```

5. Convert the image to grayscale:

```
    # Convert to grayscale
    img_gray = cv2.cvtColor(input_img, cv2.COLOR_BGR2GRAY)
```

6. Detect features using the Star feature detector:

```
    # Detect features using Star feature detector
    keypoints = StarFeatureDetector().detect(input_img)
```

7. Draw keypoints on top of the input image:

```
    cv2.drawKeypoints(input_img, keypoints, input_img,
               flags=cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)
```

8. Display the output image:

```
cv2.imshow('Star features', input_img)
cv2.waitKey()
```

9. The full code is given in the `star_detector.py` file that is already provided to you. The output image looks like the following:

Creating features using visual codebook and vector quantization

In order to build an object recognition system, we need to extract feature vectors from each image. Each image needs to have a signature that can be used for matching. We use a concept called **visual codebook** to build image signatures. This codebook is basically the dictionary that we will use to come up with a representation for the images in our training dataset. We use vector quantization to cluster many feature points and come up with centroids. These centroids will serve as the elements of our visual codebook. You can learn more about this at http://mi.eng.cam.ac.uk/~cipolla/lectures/PartIB/old/IB-visualcodebook.pdf.

Before you start, make sure that you have some training images. You were provided with a sample training dataset that contains three classes, where each class has 20 images. These images were downloaded from http://www.vision.caltech.edu/html-files/archive.html.

To build a robust object recognition system, you need tens of thousands of images. There is a dataset called `Caltech256` that's very popular in this field! It contains 256 classes of images, where each class contains thousands of samples. You can download this dataset at http://www.vision.caltech.edu/Image_Datasets/Caltech256.

How to do it…

1. This is a lengthy recipe, so we will only look at the important functions. The full code is given in the `build_features.py` file that is already provided to you. Let's look at the class defined to extract features:

```
class FeatureBuilder(object):
```

2. Define a method to extract features from the input image. We will use the Star detector to get the keypoints and then use SIFT to extract descriptors from these locations:

```
    def extract_ features(self, img):
        keypoints = StarFeatureDetector().detect(img)
        keypoints, feature_vectors = compute_sift_features(img,
keypoints)
        return feature_vectors
```

3. We need to extract centroids from all the descriptors:

```
    def get_codewords(self, input_map, scaling_size,
max_samples=12):
        keypoints_all = []

        count = 0
        cur_label = ''
```

4. Each image will give rise to a large number of descriptors. We will just use a small number of images because the centroids won't change much after this:

```
        for item in input_map:
            if count >= max_samples:
                if cur_class != item['object_class']:
                    count = 0
            else:
                continue

        count += 1
```

5. The print progress is as follows:

```
        if count == max_samples:
            print "Built centroids for", item['object_class']
```

6. Extract the current label:

```
        cur_class = item['object_class']
```

7. Read the image and resize it:

```
        img = cv2.imread(item['image_path'])
        img = resize_image(img, scaling_size)
```

8. Set the number of dimensions to 128 and extract the features:

```
        num_dims = 128
        feature_vectors = self.extract_image_features(img)
        keypoints_all.extend(feature_vectors)
```

9. Use vector quantization to quantize the feature points. **Vector quantization** is the *N*-dimensional version of "rounding off". You can learn more about it at http://www.data-compression.com/vq.shtml.

```
        kmeans, centroids = BagOfWords().cluster(keypoints_all)
        return kmeans, centroids
```

10. Define the class to handle bag of words model and vector quantization:

```
class BagOfWords(object):
    def __init__(self, num_clusters=32):
        self.num_dims = 128
        self.num_clusters = num_clusters
        self.num_retries = 10
```

11. Define a method to quantize the datapoints. We will use **k-means clustering** to achieve this:

```
def cluster(self, datapoints):
    kmeans = KMeans(self.num_clusters,
```

```
        n_init=max(self.num_retries, 1),
        max_iter=10, tol=1.0)
```

12. Extract the centroids, as follows:

```
    res = kmeans.fit(datapoints)
    centroids = res.cluster_centers_
    return kmeans, centroids
```

13. Define a method to normalize the data:

```
def normalize(self, input_data):
    sum_input = np.sum(input_data)

    if sum_input > 0:
        return input_data / sum_input
    else:
        return input_data
```

14. Define a method to get the feature vector:

```
def construct_feature(self, img, kmeans, centroids):
    keypoints = StarFeatureDetector().detect(img)
    keypoints, feature_vectors = compute_sift_features(img,
keypoints)
    labels = kmeans.predict(feature_vectors)
    feature_vector = np.zeros(self.num_clusters)
```

15. Build a histogram and normalize it:

```
    for i, item in enumerate(feature_vectors):
        feature_vector[labels[i]] += 1

        feature_vector_img = np.reshape(feature_vector,
((1, feature_vector.shape[0])))
        return self.normalize(feature_vector_img)
```

16. Define a method the extract the SIFT features:

```
# Extract SIFT features
def compute_sift_features(img, keypoints):
    if img is None:
        raise TypeError('Invalid input image')

    img_gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    keypoints, descriptors =
cv2.xfeatures2d.SIFT_create().compute(img_gray, keypoints)
    return keypoints, descriptors
```

As mentioned earlier, please refer to build_features.py for the complete code. You should run the code in the following way:

```
$ python build_features.py --data-folder /path/to/
training_images/ --codebook-file codebook.pkl
--feature-map-file feature_map.pkl
```

This will generate two files called `codebook.pkl` and `feature_map.pkl`. We will use these files in the next recipe.

Training an image classifier using Extremely Random Forests

We will use **Extremely Random Forests** (**ERFs**) to train our image classifier. An object recognition system uses an image classifier to classify the images into known categories. ERFs are very popular in the field of machine learning because of their speed and accuracy. We basically construct a bunch of decision trees that are based on our image signatures, and then train the forest to make the right decision. You can learn more about random forests at https://www.stat.berkeley.edu/~breiman/RandomForests/cc_home.htm. You can learn about ERFs at http://www.montefiore.ulg.ac.be/~ernst/uploads/news/id63/extremely-randomized-trees.pdf.

How to do it…

1. Create a new Python file, and import the following packages:

```
import argparse
import cPickle as pickle

import numpy as np
from sklearn.ensemble import ExtraTreesClassifier
from sklearn import preprocessing
```

2. Define an argument parser:

```
def build_arg_parser():
    parser = argparse.ArgumentParser(description='Trains the
classifier')
    parser.add_argument("--feature-map-file",
dest="feature_map_file", required=True,
help="Input pickle file containing the feature map")
    parser.add_argument("--model-file", dest="model_file",
required=False,
help="Output file where the trained model will be stored")
    return parser
```

3. Define a class to handle ERF training. We will use a label encoder to encode our training labels:

```
class ERFTrainer(object):
    def __init__(self, X, label_words):
        self.le = preprocessing.LabelEncoder()
        self.clf = ExtraTreesClassifier(n_estimators=100,
max_depth=16, random_state=0)
```

4. Encode the labels and train the classifier:

```
        y = self.encode_labels(label_words)
        self.clf.fit(np.asarray(X), y)
```

5. Define a function to encode the labels:

```
def encode_labels(self, label_words):
    self.le.fit(label_words)
    return np.array(self.le.transform(label_words),
dtype=np.float32)
```

6. Define a function to classify an unknown datapoint:

```
def classify(self, X):
    label_nums = self.clf.predict(np.asarray(X))
    label_words = self.le.inverse_transform([int(x) for x in
label_nums])
    return label_words
```

7. Define the `main` function and parse the input arguments:

```
if __name__=='__main__':
    args = build_arg_parser().parse_args()
    feature_map_file = args.feature_map_file
    model_file = args.model_file
```

8. Load the feature map that we created in the previous recipe:

```
    # Load the feature map
    with open(feature_map_file, 'r') as f:
        feature_map = pickle.load(f)
```

9. Extract the feature vectors:

```
    # Extract feature vectors and the labels
    label_words = [x['object_class'] for x in feature_map]
    dim_size = feature_map[0]['feature_vector'].shape[1]
    X = [np.reshape(x['feature_vector'], (dim_size,)) for x in
feature_map]
```

10. Train the ERF, which is based on the training data:

```
    # Train the Extremely Random Forests classifier
    erf = ERFTrainer(X, label_words)
```

11. Save the trained ERF model, as follows:

```
    if args.model_file:
        with open(args.model_file, 'w') as f:
            pickle.dump(erf, f)
```

12. The full code is given in the `trainer.py` file that is provided to you. You should run the code in the following way:

```
$ python trainer.py --feature-map-file feature_map.pkl
--model-file erf.pkl
```

This will generate a file called `erf.pkl`. We will use this file in the next recipe.

Building an object recognizer

Now that we trained an ERF model, let's go ahead and build an object recognizer that can recognize the content of unknown images.

How to do it…

1. Create a new Python file, and import the following packages:

```
import argparse
import cPickle as pickle

import cv2
import numpy as np

import build_features as bf
from trainer import ERFTrainer
```

2. Define the argument parser:

```
def build_arg_parser():
    parser = argparse.ArgumentParser(description='Extracts
features \
from each line and classifies the data')
    parser.add_argument("--input-image", dest="input_image",
required=True,
help="Input image to be classified")
    parser.add_argument("--model-file", dest="model_file",
required=True,
help="Input file containing the trained model")
    parser.add_argument("--codebook-file",
dest="codebook_file",
required=True, help="Input file containing the codebook")
    return parser
```

3. Define a class to handle the image tag extraction functions:

```
class ImageTagExtractor(object):
    def __init__(self, model_file, codebook_file):
        with open(model_file, 'r') as f:
            self.erf = pickle.load(f)

        with open(codebook_file, 'r') as f:
            self.kmeans, self.centroids = pickle.load(f)
```

4. Define a function to predict the output using the trained ERF model:

```
    def predict(self, img, scaling_size):
        img = bf.resize_image(img, scaling_size)
        feature_vector = bf.BagOfWords().construct_feature(
img, self.kmeans, self.centroids)
        image_tag = self.erf.classify(feature_vector)[0]
        return image_tag
```

5. Define the `main` function and load the input image:

```
if __name__=='__main__':
    args = build_arg_parser().parse_args()
    model_file = args.model_file
    codebook_file = args.codebook_file
    input_image = cv2.imread(args.input_image)
```

6. Scale the image appropriately, as follows:

```
    scaling_size = 200
```

7. Print the output on the Terminal:

```
    print "\nOutput:", ImageTagExtractor(model_file,
    codebook_file).predict(input_image, scaling_size)
```

8. The full code is given in the `object_recognizer.py` file that is already provided to you. You should run the code in the following way:

```
$ python object_recognizer.py --input-image imagefile.jpg
--model-file erf.pkl --codebook-file codebook.pkl
```

You will see the output class printed on the Terminal.

Chapter 10. Biometric Face Recognition

In this chapter, we will cover the following recipes:

- Capturing and processing video from a webcam
- Building a face detector using Haar cascades
- Building eye and nose detectors
- Performing Principal Components Analysis
- Performing Kernel Principal Components Analysis
- Performing blind source separation
- Building a face recognizer using Local Binary Patterns Histogram

Introduction

Face recognition refers to the task of identifying the person in a given image. This is different from face detection where we locate the face in a given image. During face detection, we don't care who the person is. We just identify the region of the image that contains the face. Therefore, in a typical biometric face-recognition system, we need to determine the location of the face before we can recognize it.

Face recognition is very easy for humans. We seem to do it effortlessly, and we do it all the time! How do we get a machine to do the same thing? We need to understand what parts of the face we can use to uniquely identify a person. Our brain has an internal structure that seems to respond to specific features, such as edges, corners, motion, and so on. The human visual cortex combines all these features into a single coherent inference. If we want our machine to recognize faces with accuracy, we need to formulate the problem in a similar way. We need to extract features from the input image and convert it into a meaningful representation.

Capturing and processing video from a webcam

We will use a webcam in this chapter to capture video data. Let's see how to capture the video from the webcam using OpenCV-Python.

How to do it…

1. Create a new Python file, and import the following packages:

```
import cv2
```

2. OpenCV provides a video capture object that we can use to capture images from the webcam. The `0` input argument specifies the ID of the webcam. If you connect a USB camera, then it will have a different ID:

```
# Initialize video capture object
cap = cv2.VideoCapture(0)
```

3. Define the scaling factor for the frames captured using the webcam:

```
# Define the image size scaling factor
scaling_factor = 0.5
```

4. Start an infinite loop and keep capturing frames until you press the *Esc* key. Read the frame from the webcam:

```
# Loop until you hit the Esc key
while True:
    # Capture the current frame
    ret, frame = cap.read()
```

5. Resizing the frame is optional but still a useful thing to have in your code:

```
    # Resize the frame
    frame = cv2.resize(frame, None, fx=scaling_factor,
fy=scaling_factor,
            interpolation=cv2.INTER_AREA)
```

6. Display the frame:

```
    # Display the image
    cv2.imshow('Webcam', frame)
```

7. Wait for 1 ms before capturing the next frame:

```
    # Detect if the Esc key has been pressed
    c = cv2.waitKey(1)
    if c == 27:
        break
```

8. Release the video capture object:

```
# Release the video capture object
cap.release()
```

9. Close all active windows before exiting the code:

```
# Close all active windows
cv2.destroyAllWindows()
```

10. The full code is given in the `video_capture.py` file that's already provided to you for reference. If you run this code, you will see the video from the webcam, similar to the following screenshot:

Building a face detector using Haar cascades

As we discussed earlier, face detection is the process of determining the location of the face in the input image. We will use **Haar cascades** for face detection. This works by extracting a large number of simple features from the image at multiple scales. The simple features are basically edge, line, and rectangle features that are very easy to compute. It is then trained by creating a cascade of simple classifiers. The **Adaptive Boosting** technique is used to make this process robust. You can learn more about it at http://docs.opencv.org/3.1.0/d7/d8b/tutorial_py_face_detection.html#gsc.tab=0. Let's take a look at how to determine the location of a face in the video frames captured from the webcam.

How to do it…

1. Create a new Python file, and import the following packages:

```
import cv2
import numpy as np
```

2. Load the face detector cascade file. This is a trained model that we can use as a detector:

```
# Load the face cascade file
face_cascade = cv2.CascadeClassifier('cascade_files/
haarcascade_frontalface_alt.xml')
```

3. Check whether the cascade file loaded properly:

```
# Check if the face cascade file has been loaded
if face_cascade.empty():
    raise IOError('Unable to load the face cascade classifier
xml file')
```

4. Create the video capture object:

```
# Initialize the video capture object
cap = cv2.VideoCapture(0)
```

5. Define the scaling factor for image downsampling:

```
# Define the scaling factor
scaling_factor = 0.5
```

6. Keep looping until you hit the *Esc* key:

```
# Loop until you hit the Esc key
while True:
    # Capture the current frame and resize it
    ret, frame = cap.read()
```

7. Resize the frame:

```
    frame = cv2.resize(frame, None, fx=scaling_factor,
fy=scaling_factor,
            interpolation=cv2.INTER_AREA)
```

8. Convert the image to grayscale. We need grayscale images to run the face detector:

```
    # Convert to grayscale
    gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
```

9. Run the face detector on the grayscale image. The `1.3` parameter refers to the scale multiplier for each stage. The `5` parameter refers to the minimum number of neighbors that each candidate rectangle should have so that we can retain it. This candidate rectangle is basically a potential region where there is a chance of a face being detected:

```
    # Run the face detector on the grayscale image
    face_rects = face_cascade.detectMultiScale(gray, 1.3, 5)
```

10. For each detected face region, draw a rectangle around it:

```
    # Draw rectangles on the image
    for (x,y,w,h) in face_rects:
        cv2.rectangle(frame, (x,y), (x+w,y+h), (0,255,0), 3)
```

11. Display the output image:

```
    # Display the image
    cv2.imshow('Face Detector', frame)
```

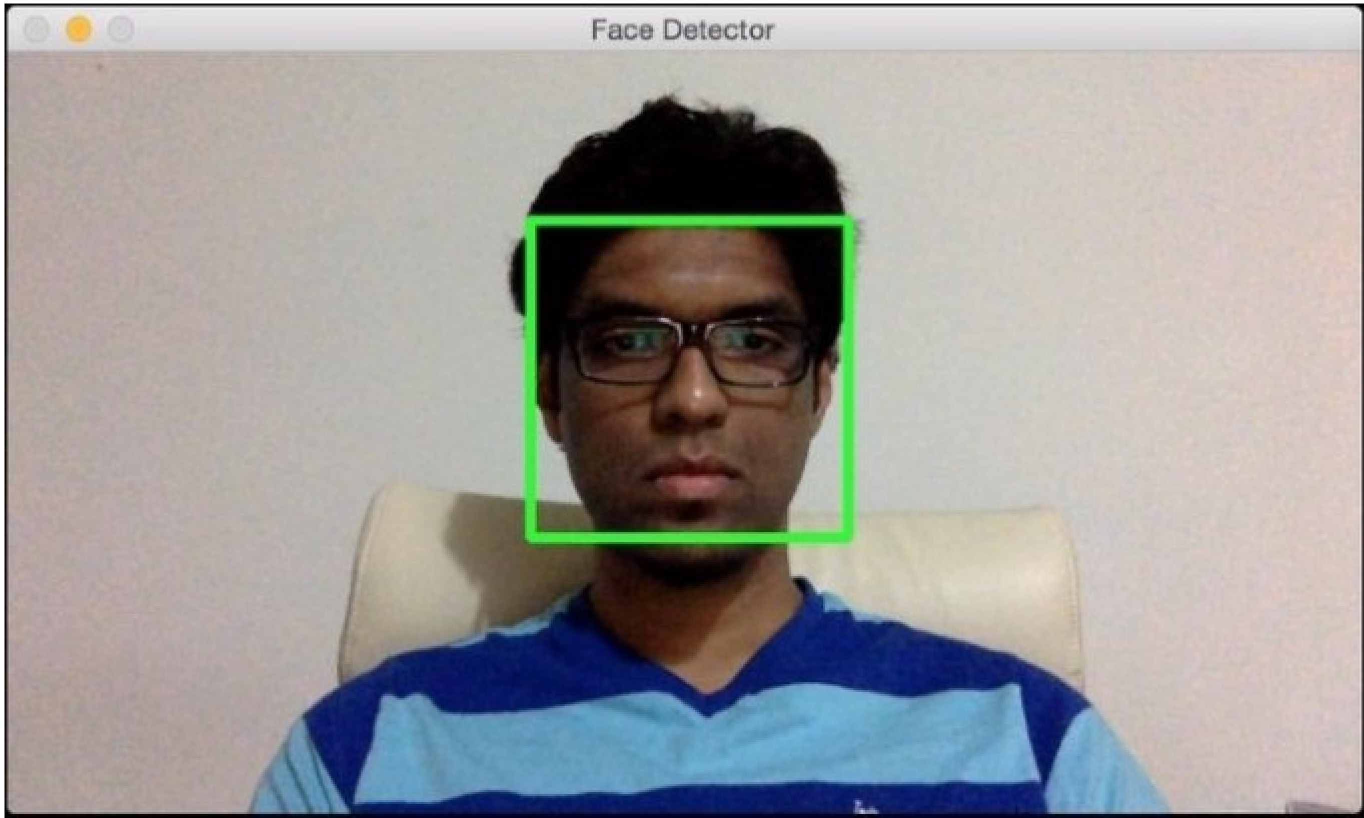
12. Wait for 1 ms before going to the next iteration. If the user presses the *Esc* key, break out of the loop:

```
    # Check if Esc key has been pressed
    c = cv2.waitKey(1)
    if c == 27:
        break
```

13. Release and destroy the objects before exiting the code:

```
# Release the video capture object and close all windows
cap.release()
cv2.destroyAllWindows()
```

14. The full code is given in the `face_detector.py` file that's already provided to you for reference. If you run this code, you will see the face being detected in the webcam video:


Face Detector

Building eye and nose detectors

The Haar cascades method can be extended to detect all types of objects. Let's see how to use it to detect the eyes and nose in the input video.

How to do it…

1. Create a new Python file, and import the following packages:

```
import cv2
import numpy as np
```

2. Load the face, eyes, and nose cascade files:

```
# Load face, eye, and nose cascade files
face_cascade = cv2.CascadeClassifier('cascade_files/
haarcascade_frontalface_alt.xml')
eye_cascade = cv2.CascadeClassifier('cascade_files/
haarcascade_eye.xml')
nose_cascade = cv2.CascadeClassifier('cascade_files/
haarcascade_mcs_nose.xml')
```

3. Check whether the files loaded correctly:

```
# Check if face cascade file has been loaded
if face_cascade.empty():
    raise IOError('Unable to load the face cascade classifier
xml file')

# Check if eye cascade file has been loaded
if eye_cascade.empty():
    raise IOError('Unable to load the eye cascade classifier
xml file')

# Check if nose cascade file has been loaded
if nose_cascade.empty():
    raise IOError('Unable to load the nose cascade classifier
xml file')
```

4. Initialize the video capture object:

```
# Initialize video capture object and define scaling factor
cap = cv2.VideoCapture(0)
```

5. Define the scaling factor:

```
scaling_factor = 0.5
```

6. Keep looping until the user presses the *Esc* key:

```
while True:
    # Read current frame, resize it, and convert it to grayscale
    ret, frame = cap.read()
```

7. Resize the frame:

```
    frame = cv2.resize(frame, None, fx=scaling_factor,
fy=scaling_factor,
            interpolation=cv2.INTER_AREA)
```

8. Convert the image to grayscale:

```
    gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
```

9. Run the face detector on the grayscale image:

```
    # Run face detector on the grayscale image
    faces = face_cascade.detectMultiScale(gray, 1.3, 5)
```

10. As we know that eyes and noses are always on faces, we can run these detectors only in the face region:

```
    # Run eye and nose detectors within each face rectangle
    for (x,y,w,h) in faces:
```

11. Extract the face ROI:

```
        # Grab the current ROI in both color and grayscale
images
        roi_gray = gray[y:y+h, x:x+w]
        roi_color = frame[y:y+h, x:x+w]
```

12. Run the eye detector:

```
        # Run eye detector in the grayscale ROI
        eye_rects = eye_cascade.detectMultiScale(roi_gray)
```

13. Run the nose detector:

```
        # Run nose detector in the grayscale ROI
        nose_rects = nose_cascade.detectMultiScale(roi_gray,
1.3, 5)
```

14. Draw circles around the eyes:

```
        # Draw green circles around the eyes
        for (x_eye, y_eye, w_eye, h_eye) in eye_rects:
            center = (int(x_eye + 0.5*w_eye), int(y_eye +
0.5*h_eye))
            radius = int(0.3 * (w_eye + h_eye))
            color = (0, 255, 0)
```

```
            thickness = 3
            cv2.circle(roi_color, center, radius, color,
thickness)
```

15. Draw a rectangle around the nose:

```
        for (x_nose, y_nose, w_nose, h_nose) in nose_rects:
            cv2.rectangle(roi_color, (x_nose, y_nose),
(x_nose+w_nose,
                y_nose+h_nose), (0,255,0), 3)
            break
```

16. Display the image:

```
    # Display the image
    cv2.imshow('Eye and nose detector', frame)
```

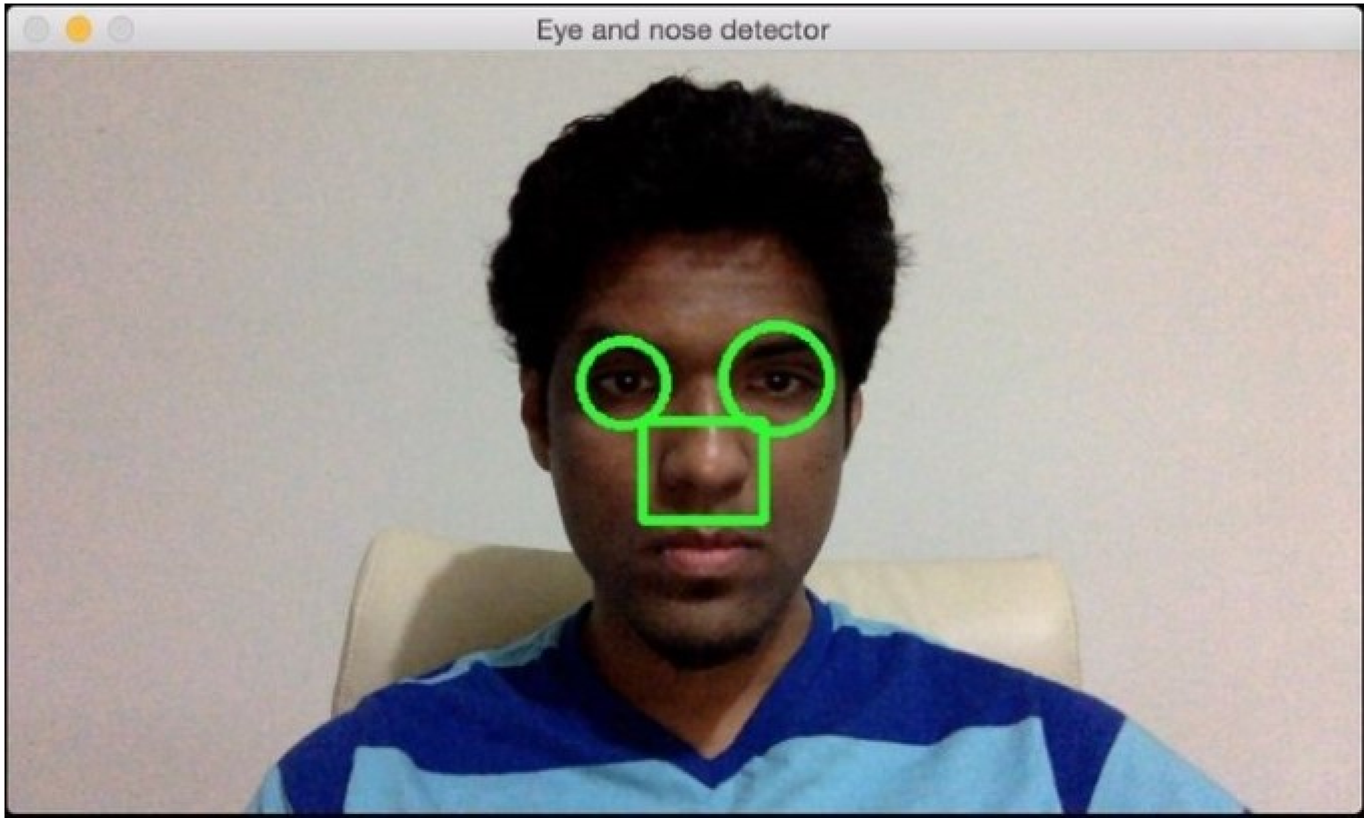
17. Wait for 1 ms before going to the next iteration. If the user presses the *Esc* key, then break the loop.

```
    # Check if Esc key has been pressed
    c = cv2.waitKey(1)
    if c == 27:
        break
```

18. Release and destroy the objects before exiting the code.

```
# Release video capture object and close all windows
cap.release()
cv2.destroyAllWindows()
```

19. The full code is given in the `eye_nose_detector.py` file that's already provided to you for reference. If you run this code, you will see the eyes and nose being detected in the webcam video:


Eye and nose detector

Performing Principal Components Analysis

Principal Components Analysis (**PCA**) is a dimensionality reduction technique that's used very frequently in computer vision and machine learning. When we deal with features with large dimensionalities, training a machine learning system becomes prohibitively expensive. Therefore, we need to reduce the dimensionality of the data before we can train a system. However, when we reduce the dimensionality, we don't want to lose the information present in the data. This is where PCA comes into the picture! PCA identifies the important components of the data and arranges them in the order of importance. You can learn more about it at http://dai.fmph.uniba.sk/courses/ml/sl/PCA.pdf. It is used a lot in face recognition systems. Let's see how to perform PCA on input data.

How to do it…

1. Create a new Python file, and import the following packages:

```
import numpy as np
from sklearn import decomposition
```

2. Let's define five dimensions for our input data. The first two dimensions will be independent, but the next three dimensions will be dependent on the first two dimensions. This basically means that we can live without the last three dimensions because they do not give us any new information:

```
# Define individual features
x1 = np.random.normal(size=250)
x2 = np.random.normal(size=250)
x3 = 2*x1 + 3*x2
x4 = 4*x1 - x2
x5 = x3 + 2*x4
```

3. Let's create a dataset with these features.

```
# Create dataset with the above features
X = np.c_[x1, x3, x2, x5, x4]
```

4. Create a PCA object:

```
# Perform Principal Components Analysis
pca = decomposition.PCA()
```

5. Fit a PCA model on the input data:

```
pca.fit(X)
```

6. Print the variances of the dimensions:

```
# Print variances
variances = pca.explained_variance_
print '\nVariances in decreasing order:\n', variances
```

7. If a particular dimension is useful, then it will have a meaningful value for the variance. Let's set a threshold and identify the important dimensions:

```
# Find the number of useful dimensions
thresh_variance = 0.8
num_useful_dims = len(np.where(variances > thresh_variance)[0])
print '\nNumber of useful dimensions:', num_useful_dims
```

8. Just like we discussed earlier, PCA identified that only two dimensions are important in this dataset:

```
# As we can see, only the 2 first components are useful
pca.n_components = num_useful_dims
```

9. Let's convert the dataset from a five-dimensional set to a two-dimensional set:

```
X_new = pca.fit_transform(X)
print '\nShape before:', X.shape
print 'Shape after:', X_new.shape
```

10. The full code is given in the `pca.py` file that's already provided to you for reference. If you run this code, you will see the following on your Terminal:

```
Variances in decreasing order:
[  1.13489352e+02   1.08125265e+01   3.34017371e-31   4.36320756e-32
   1.49223239e-32]

Number of useful dimensions: 2

Shape before: (250, 5)
Shape after: (250, 2)
```

Performing Kernel Principal Components Analysis

PCA is good at reducing the number of dimensions, but it works in a linear manner. If the data is not organized in a linear fashion, PCA fails to do the required job. This is where Kernel PCA comes into the picture. You can learn more about it at http://www.ics.uci.edu/~welling/classnotes/papers_class/Kernel-PCA.pdf. Let's see how to perform Kernel PCA on the input data and compare it to how PCA performs on the same data.

How to do it…

1. Create a new Python file, and import the following packages:

```
import numpy as np
import matplotlib.pyplot as plt

from sklearn.decomposition import PCA, KernelPCA
from sklearn.datasets import make_circles
```

2. Define the seed value for the random number generator. This is needed to generate the data samples for analysis:

```
# Set the seed for random number generator
np.random.seed(7)
```

3. Generate data that is distributed in concentric circles to demonstrate how PCA doesn't work in this case:

```
# Generate samples
X, y = make_circles(n_samples=500, factor=0.2, noise=0.04)
```

4. Perform PCA on this data:

```
# Perform PCA
pca = PCA()
X_pca = pca.fit_transform(X)
```

5. Perform Kernel PCA on this data:

```
# Perform Kernel PCA
kernel_pca = KernelPCA(kernel="rbf",
fit_inverse_transform=True, gamma=10)
X_kernel_pca = kernel_pca.fit_transform(X)
X_inverse = kernel_pca.inverse_transform(X_kernel_pca)
```

6. Plot the original input data:

```
# Plot original data
class_0 = np.where(y == 0)
class_1 = np.where(y == 1)
plt.figure()
plt.title("Original data")
plt.plot(X[class_0, 0], X[class_0, 1], "ko", mfc='none')
plt.plot(X[class_1, 0], X[class_1, 1], "kx")
plt.xlabel("1st dimension")
plt.ylabel("2nd dimension")
```

7. Plot the PCA-transformed data:

```
# Plot PCA projection of the data
plt.figure()
plt.plot(X_pca[class_0, 0], X_pca[class_0, 1], "ko", mfc='none')
plt.plot(X_pca[class_1, 0], X_pca[class_1, 1], "kx")
plt.title("Data transformed using PCA")
plt.xlabel("1st principal component")
plt.ylabel("2nd principal component")
```

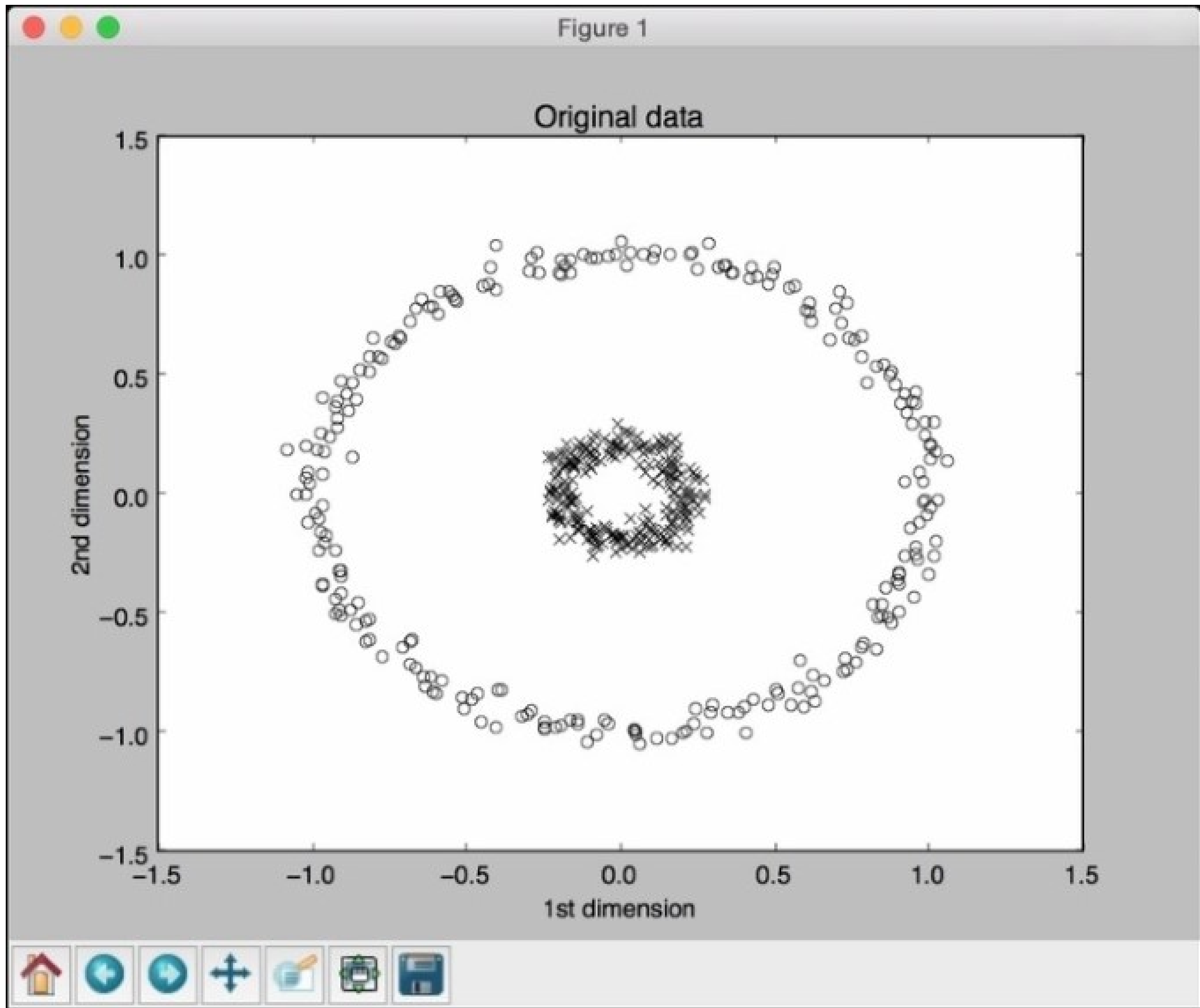
8. Plot Kernel PCA-transformed data:

```
# Plot Kernel PCA projection of the data
plt.figure()
plt.plot(X_kernel_pca[class_0, 0], X_kernel_pca[class_0, 1],
"ko", mfc='none')
plt.plot(X_kernel_pca[class_1, 0], X_kernel_pca[class_1, 1],
"kx")
plt.title("Data transformed using Kernel PCA")
plt.xlabel("1st principal component")
plt.ylabel("2nd principal component")
```

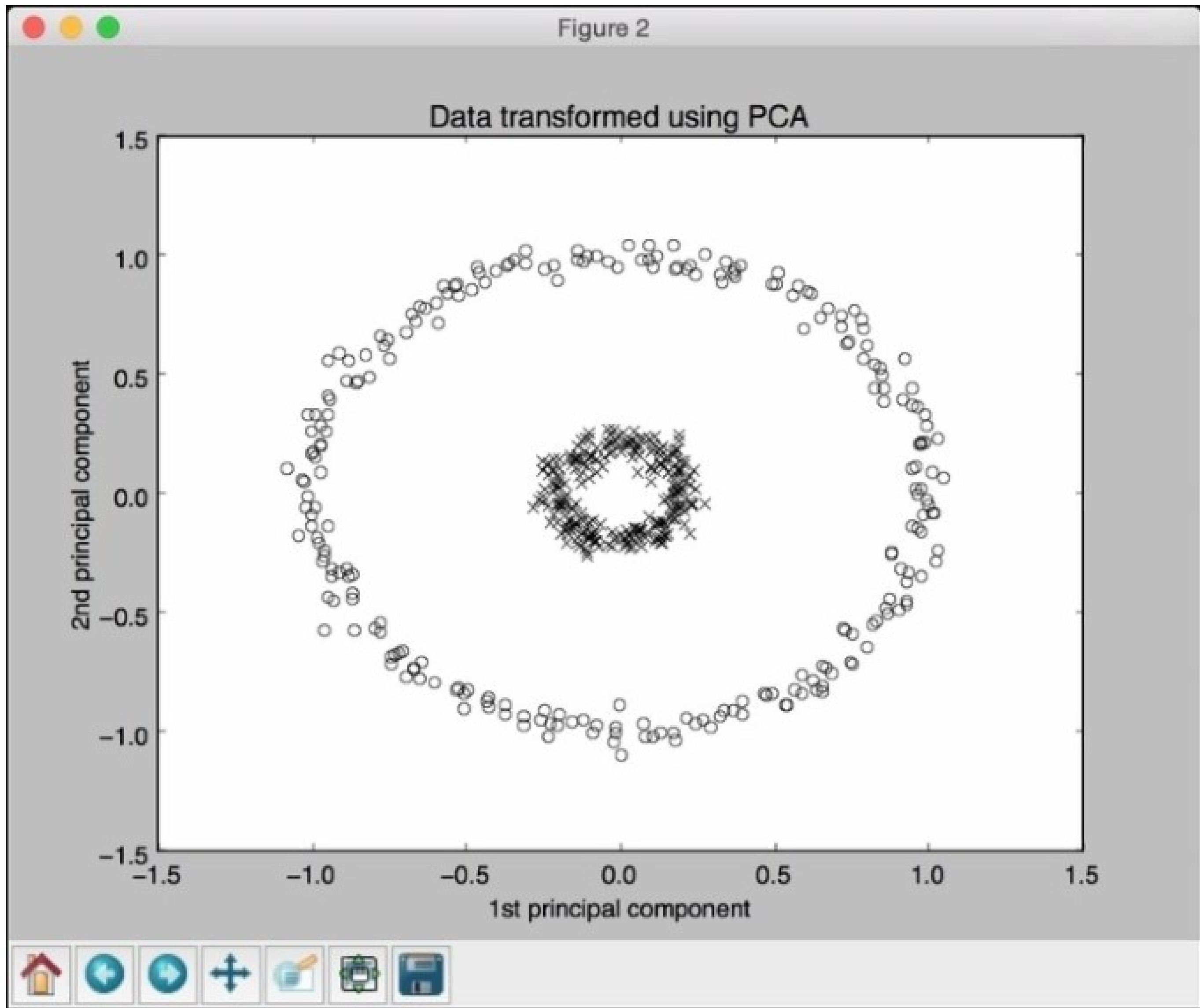
9. Transform the data back to the original space using the Kernel method to show that the inverse is maintained:

```
# Transform the data back to original space
plt.figure()
plt.plot(X_inverse[class_0, 0], X_inverse[class_0, 1], "ko",
mfc='none')
plt.plot(X_inverse[class_1, 0], X_inverse[class_1, 1], "kx")
plt.title("Inverse transform")
plt.xlabel("1st dimension")
plt.ylabel("2nd dimension")

plt.show()
```

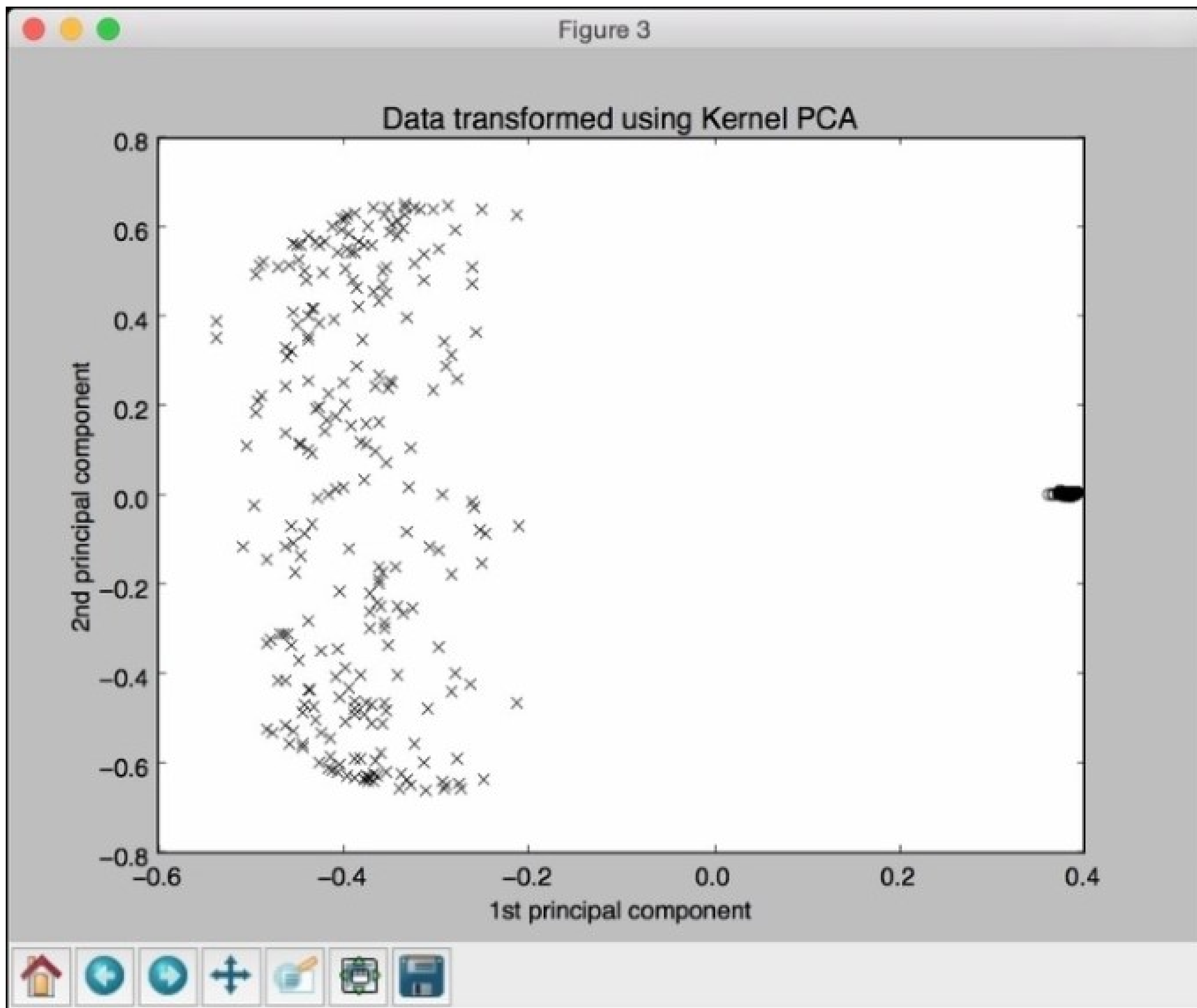
10. The full code is given in the `kpca.py` file that's already provided to you for reference. If you run this code, you will see four figures. The first figure is the original data:



The second figure depicts the data transformed using PCA:



The third figure depicts the data transformed using Kernel PCA. Note how the points are clustered in the right part of the figure:



The fourth figure depicts the inverse transform of the data back to the original space:

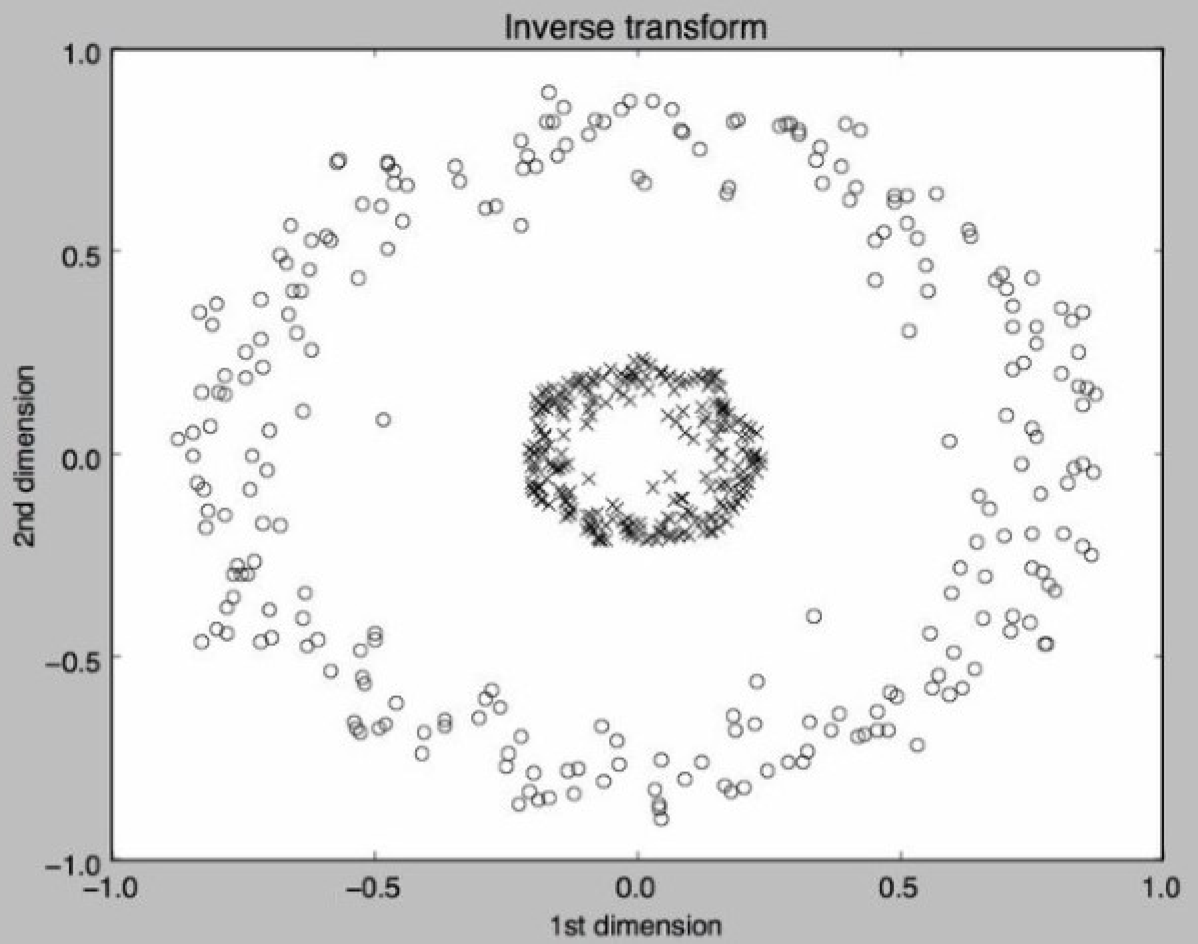
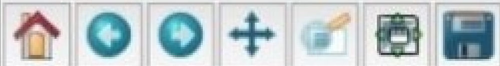


Figure 4
Inverse transform
1.0
0.5
0.0
−0.5
−1.0
2nd dimension
−1.0
−0.5
0.0
0.5
1.0
1st dimension


Performing blind source separation

Blind source separation refers to the process of separating signals from a mixture. Let's say a bunch of different signal generators generate signals and a common receiver receives all of these signals. Now, our job is to separate these signals from this mixture using the properties of these signals. We will use **Independent Components Analysis** (**ICA**) to achieve this. You can learn more about it at http://www.mit.edu/~gari/teaching/6.555/LECTURE_NOTES/ch15_bss.pdf. Let's see how to do it.

How to do it…

1. Create a new Python file, and import the following packages:

```
import numpy as np
import matplotlib.pyplot as plt
from scipy import signal

from sklearn.decomposition import PCA, FastICA
```

2. We will use data from the `mixture_of_signals.txt` file that's already provided to you. Let's load the data:

```
# Load data
input_file = 'mixture_of_signals.txt'
X = np.loadtxt(input_file)
```

3. Create the ICA object:

```
# Compute ICA
ica = FastICA(n_components=4)
```

4. Reconstruct the signals, based on ICA:

```
# Reconstruct the signals
signals_ica = ica.fit_transform(X)
```

5. Extract the mixing matrix:

```
# Get estimated mixing matrix
mixing_mat = ica.mixing_
```

6. Perform PCA for comparison:

```
# Perform PCA
pca = PCA(n_components=4)
signals_pca = pca.fit_transform(X)  # Reconstruct signals based
on orthogonal components
```

7. Define the list of signals to plot them:

```
# Specify parameters for output plots
models = [X, signals_ica, signals_pca]
```

8. Specify the colors of the plots:

```
colors = ['blue', 'red', 'black', 'green']
```

9. Plot the input signal:

```
# Plotting input signal
plt.figure()
plt.title('Input signal (mixture)')
for i, (sig, color) in enumerate(zip(X.T, colors), 1):
    plt.plot(sig, color=color)
```

10. Plot the ICA-separated signals:

```
# Plotting ICA signals
plt.figure()
plt.title('ICA separated signals')
plt.subplots_adjust(left=0.1, bottom=0.05, right=0.94,
        top=0.94, wspace=0.25, hspace=0.45)
```

11. Plot the subplots with different colors:

```
for i, (sig, color) in enumerate(zip(signals_ica.T, colors), 1):
    plt.subplot(4, 1, i)
    plt.title('Signal ' + str(i))
    plt.plot(sig, color=color)
```

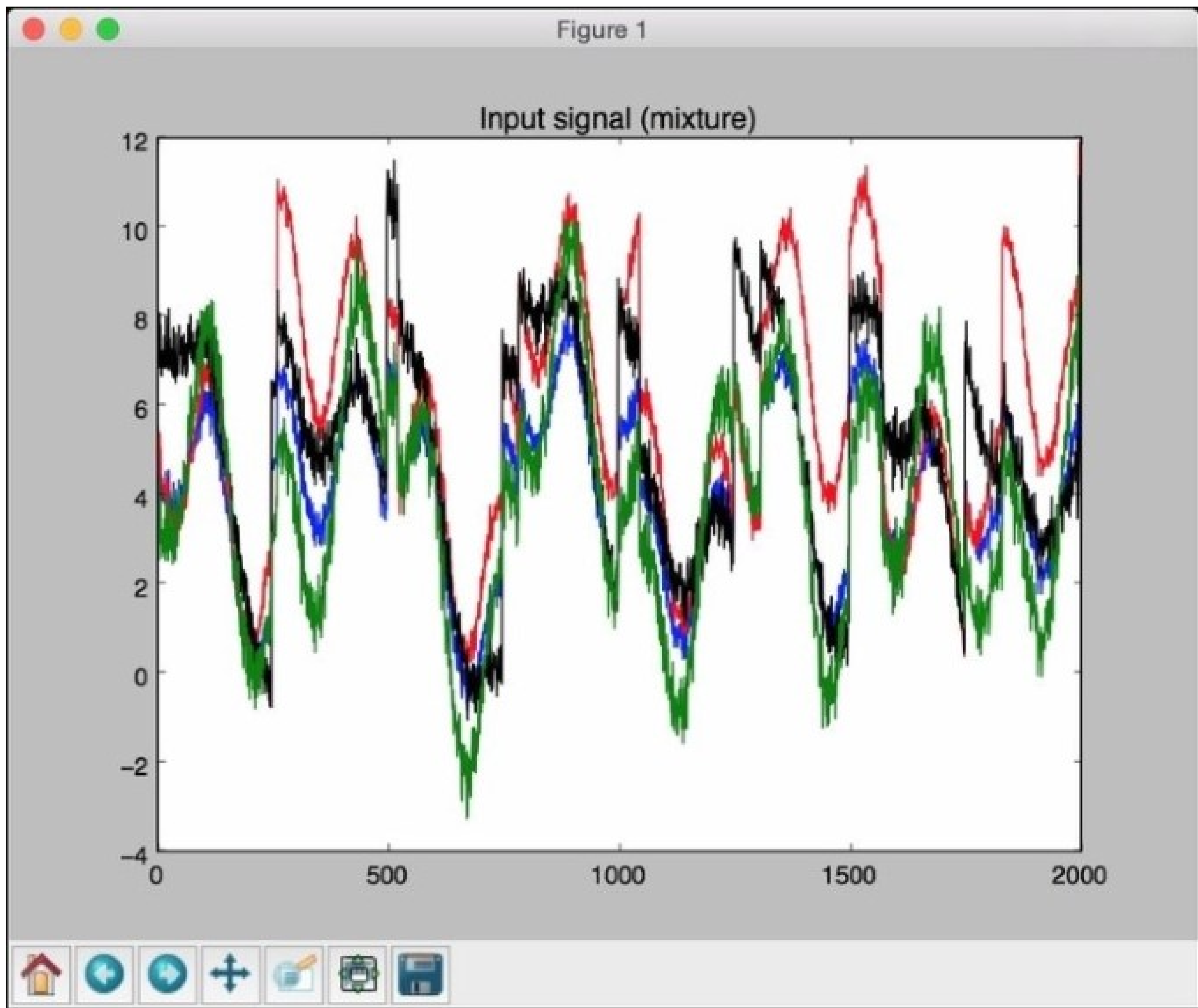
12. Plot the PCA-separated signals:

```
# Plotting PCA signals
plt.figure()
plt.title('PCA separated signals')
plt.subplots_adjust(left=0.1, bottom=0.05, right=0.94,
        top=0.94, wspace=0.25, hspace=0.45)
```

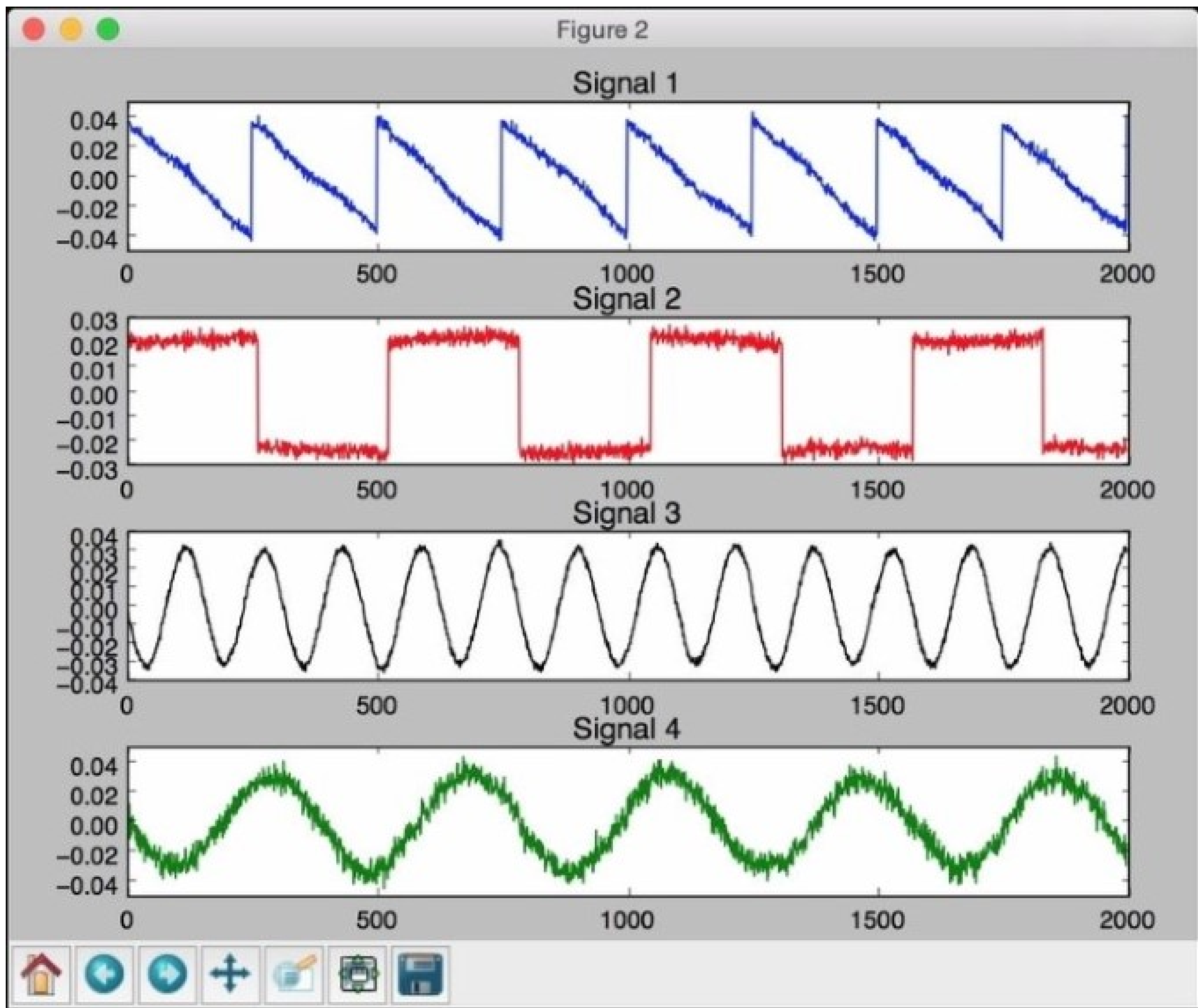
13. Use a different color in each subplot:

```
for i, (sig, color) in enumerate(zip(signals_pca.T, colors), 1):
    plt.subplot(4, 1, i)
    plt.title('Signal ' + str(i))
    plt.plot(sig, color=color)

plt.show()
```

14. The full code is given in the `blind_source_separation.py` file that's already provided to you for reference. If you run this code, you will see three figures. The first figure depicts the input, which is a mixture of signals:



The second figure depicts the signals, separated using ICA:



The third figure depicts the signals, separated using PCA:

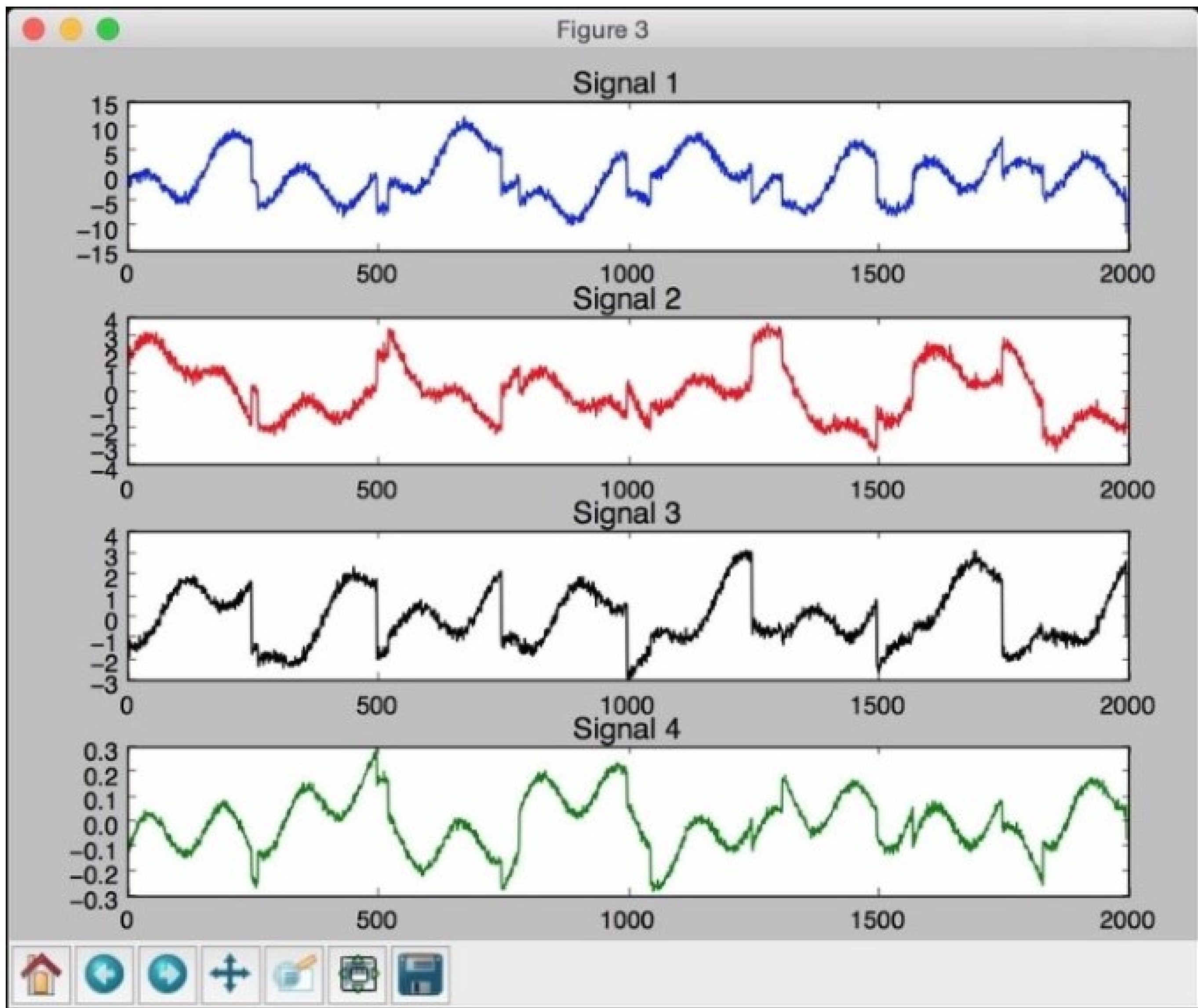


Figure 3
Signal 1
Signal 2
Signal 3
Signal 4

Building a face recognizer using Local Binary Patterns Histogram

We are now ready to build a face recognizer. We need a face dataset for training, so we've provided you a folder called `faces_dataset` that contains a small number of images sufficient for training. This dataset is a subset of the dataset that is available at http://www.vision.caltech.edu/Image_Datasets/faces/faces.tar. This dataset contains a good number of images that we can use to train a face-recognition system.

We will use **Local Binary Patterns Histograms** to build our face-recognition system. In our dataset, you will see different people. Our job is to build a system that can learn to separate these people from one another. When we see an unknown image, this system will assign it to one of the existing classes. You can learn more about Local Binary Patterns Histogram at http://docs.opencv.org/2.4/modules/contrib/doc/facerec/facerec_tutorial.html#local-binary-patterns-histograms. Let's see how to build a face recognizer.

How to do it…

1. Create a new Python file, and import the following packages:

```
import os

import cv2
import numpy as np
from sklearn import preprocessing
```

2. Let's define a class to handle all the tasks that are related to label encoding for the classes:

```
# Class to handle tasks related to label encoding
class LabelEncoder(object):
```

3. Define a method to encode the labels. In the input training data, labels are represented by words. However, we need numbers to train our system. This method will define a preprocessor object that can convert words to numbers in an organized fashion by maintaining the forward and backward mapping:

```
    # Method to encode labels from words to numbers
    def encode_labels(self, label_words):
        self.le = preprocessing.LabelEncoder()
        self.le.fit(label_words)
```

4. Define a method to convert a word to a number:

```
    # Convert input label from word to number
    def word_to_num(self, label_word):
        return int(self.le.transform([label_word])[0])
```

5. Define a method to convert the number back to the original word:

```
    # Convert input label from number to word
    def num_to_word(self, label_num):
        return self.le.inverse_transform([label_num])[0]
```

6. Define a method to extract images and labels from the input folder:

```
# Extract images and labels from input path
def get_images_and_labels(input_path):
    label_words = []
```

7. Recursively iterate through the input folder and extract all the image paths:

```
    # Iterate through the input path and append files
    for root, dirs, files in os.walk(input_path):
        for filename in (x for x in files if
x.endswith('.jpg')):
            filepath = os.path.join(root, filename)
            label_words.append(filepath.split('/')[-2])
```

8. Initialize variables:

```
    # Initialize variables
    images = []
    le = LabelEncoder()
    le.encode_labels(label_words)
    labels = []
```

9. Parse the input directory for training:

```
    # Parse the input directory
    for root, dirs, files in os.walk(input_path):
        for filename in (x for x in files if
x.endswith('.jpg')):
            filepath = os.path.join(root, filename)
```

10. Read the current image in grayscale format:

```
            # Read the image in grayscale format
            image = cv2.imread(filepath, 0)
```

11. Extract the label from the folder path:

```
            # Extract the label
            name = filepath.split('/')[-2]
```

12. Perform face detection on this image:

```
            # Perform face detection
            faces = faceCascade.detectMultiScale(image, 1.1, 2,
minSize=(100,100))
```

13. Extract the ROIs and return them along with the label encoder:

```
            # Iterate through face rectangles
            for (x, y, w, h) in faces:
                images.append(image[y:y+h, x:x+w])
                labels.append(le.word_to_num(name))

    return images, labels, le
```

14. Define the `main` function and define the path to the face cascade file:

```
if __name__=='__main__':
    cascade_path = "cascade_files/
haarcascade_frontalface_alt.xml"
    path_train = 'faces_dataset/train'
    path_test = 'faces_dataset/test'
```

15. Load the face cascade file:

```
    # Load face cascade file
    faceCascade = cv2.CascadeClassifier(cascade_path)
```

16. Create Local Binary Patterns Histogram face recognizer objects:

```
    # Initialize Local Binary Patterns Histogram face recognizer
    recognizer = cv2.face.createLBPHFaceRecognizer()
```

17. Extract the images, labels, and label encoder for this input path:

```
    # Extract images, labels, and label encoder from training
dataset
    images, labels, le = get_images_and_labels(path_train)
```

18. Train the face recognizer using the data that we extracted:

```
    # Train the face recognizer
    print "\nTraining..."
    recognizer.train(images, np.array(labels))
```

19. Test the face recognizer on unknown data:

```
    # Test the recognizer on unknown images
    print '\nPerforming prediction on test images...'
    stop_flag = False
    for root, dirs, files in os.walk(path_test):
        for filename in (x for x in files if
x.endswith('.jpg')):
            filepath = os.path.join(root, filename)
```

20. Load the image:

```
                # Read the image
                predict_image = cv2.imread(filepath, 0)
```

21. Determine the location of the face using the face detector:

```
                # Detect faces
                faces = faceCascade.detectMultiScale(predict_image,
1.1,
                        2, minSize=(100,100))
```

22. For each face ROI, run the face recognizer:

```
                # Iterate through face rectangles
                for (x, y, w, h) in faces:
                    # Predict the output
                    predicted_index, conf = recognizer.predict(
                            predict_image[y:y+h, x:x+w])
```

23. Convert the label to word:

```
                    # Convert to word label
                    predicted_person =
le.num_to_word(predicted_index)
```

24. Overlay the text on the output image and display it:

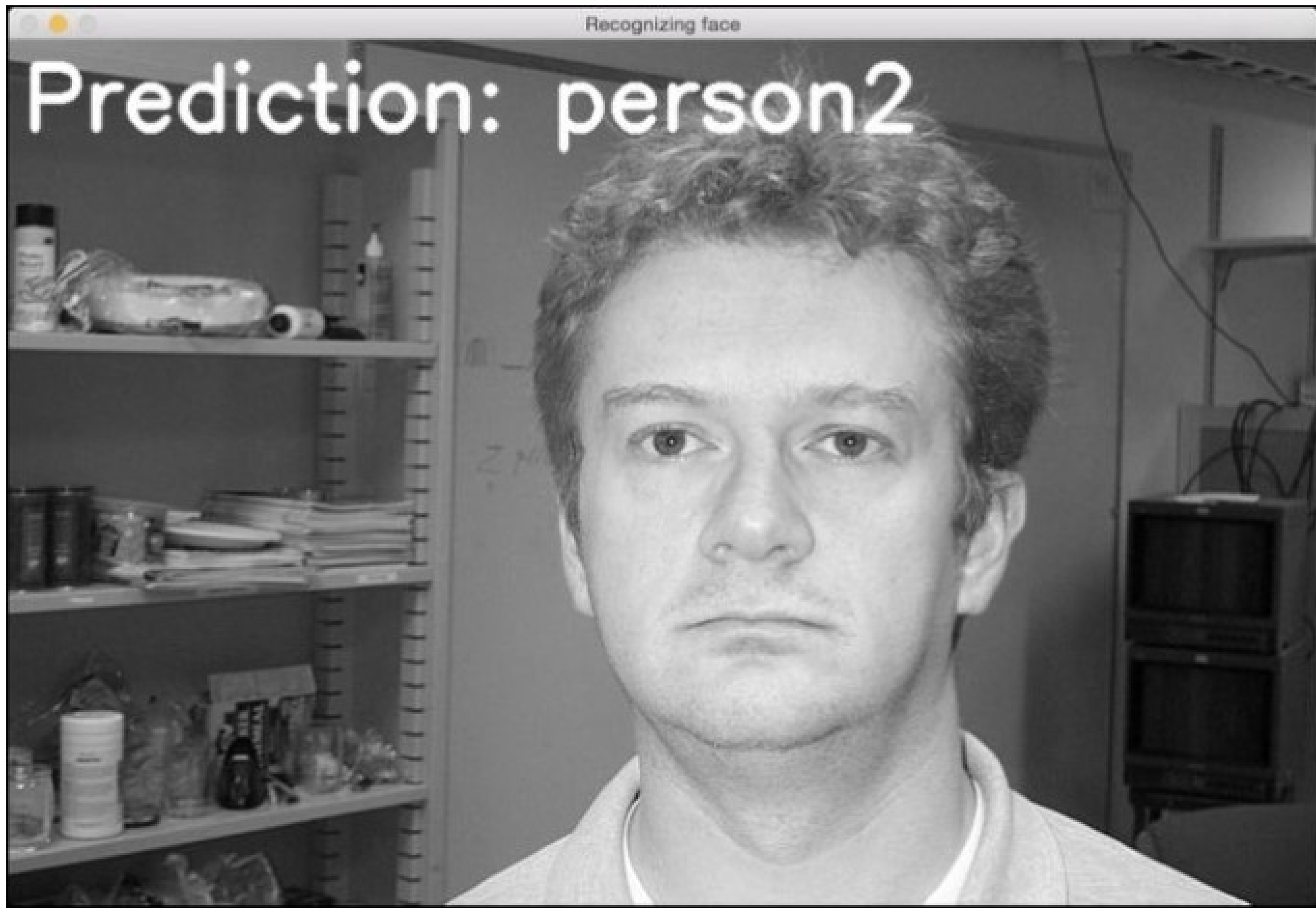
```
                    # Overlay text on the output image and display
it
                    cv2.putText(predict_image, 'Prediction: ' +
predicted_person,
                            (10,60), cv2.FONT_HERSHEY_SIMPLEX, 2,
(255,255,255), 6)
                    cv2.imshow("Recognizing face", predict_image)
```

25. Check whether the user pressed the *Esc* key. If so, break out of the loop:

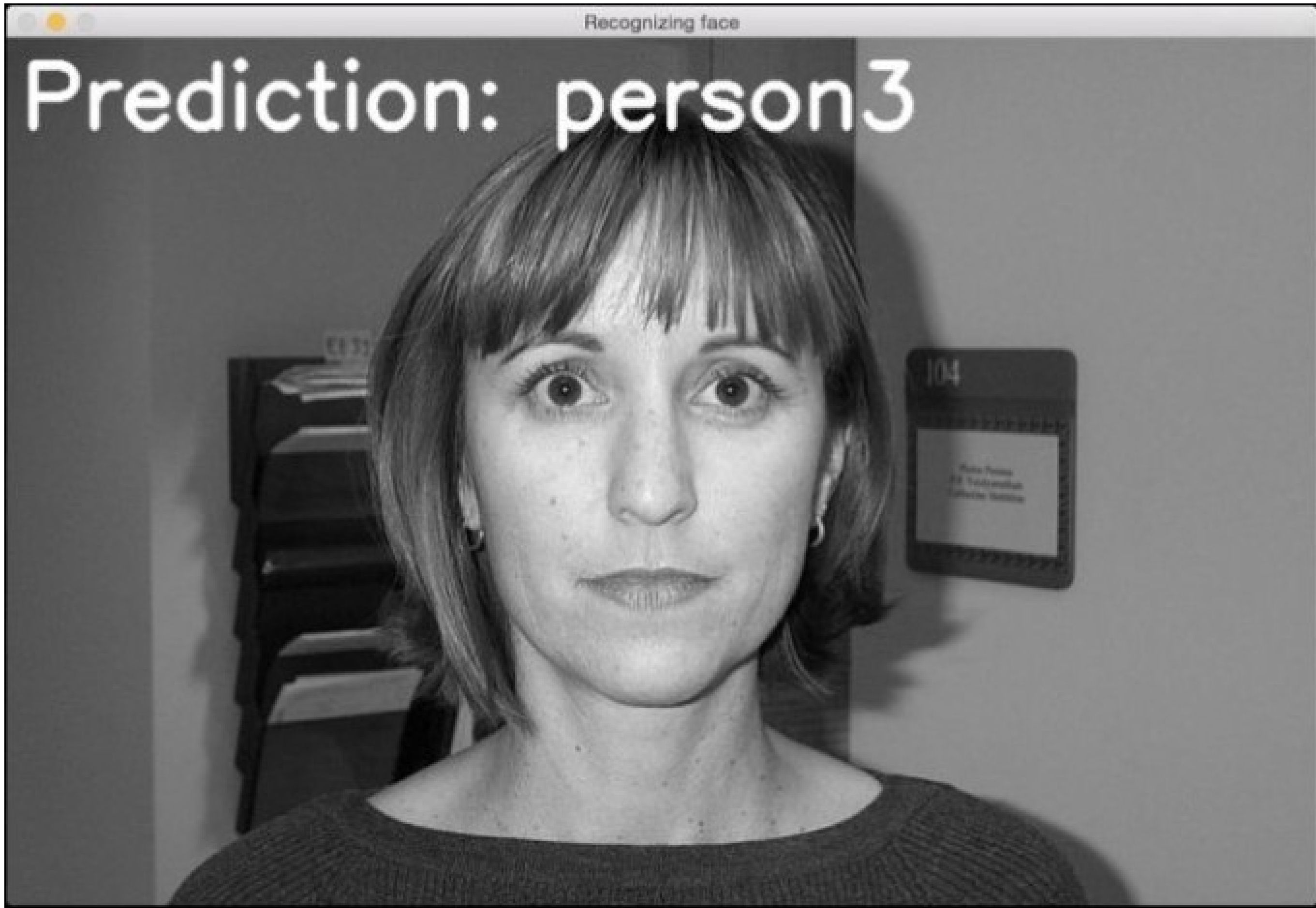
```
                c = cv2.waitKey(0)
                if c == 27:
                    stop_flag = True
                    break

            if stop_flag:
                break
```

26. The full code is in the `face_recognizer.py` file that's already provided to you for reference. If you run this code, you will get an output window, which displays the predicted outputs for test images. You can press the *Space* button to keep looping. There are three different people in the test images. The output for the first person looks like the following:

The output for the second person looks like the following:



The output for the third person looks like the following:


Recognizing face
Prediction: person3
104

Chapter 11. Deep Neural Networks

In this chapter, we will cover the following recipes:

- Building a perceptron
- Building a single layer neural network
- Building a deep neural network
- Creating a vector quantizer
- Building a recurrent neural network for sequential data analysis
- Visualizing the characters in an optical character recognition database
- Building an optical character recognizer using neural networks

Introduction

Our brain is really good at identifying and recognizing things. We want the machines to be able to do the same. A neural network is a framework that is modeled after the human brain to simulate our learning processes. Neural networks are designed to learn from data and recognize the underlying patterns. As with all learning algorithms, neural networks deal with numbers. Therefore, if we want to achieve any real world task involving images, text, sensors, and so on, we have to convert them into the numerical form before we feed them into a neural network. We can use a neural network for classification, clustering, generation, and many other related tasks.

A neural network consists of layers of **neurons**. These neurons are modeled after the biological neurons in the human brain. Each layer is basically a set of independent neurons that are connected to the neurons the adjacent layers. The input layer corresponds to the input data that we provide, and the output layer consists of the output that we desire. All the layers in between are called **hidden layers**. If we design a neural network with more hidden layers, then we give it more freedom to train itself with higher accuracy.

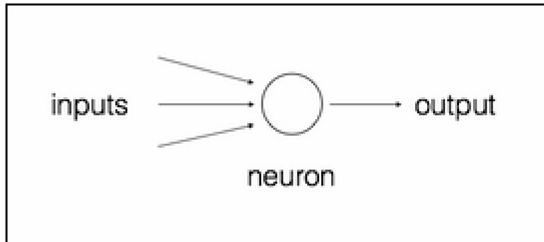
Let's say that we want the neural network to classify data, based on our needs. In order for a neural network to work accordingly, we need to provide labeled training data. The neural network will then train itself by optimizing the cost function. This cost function is the error between actual labels and the predicted labels from the neural network. We keep iterating until the error goes below a certain threshold.

What exactly are *deep* neural networks? Deep neural networks are neural networks that consist of many hidden layers. In general, this falls under the realm of deep learning. This is a field that is dedicated to the study of these neural networks, which are composed of multiple layers that are used across many verticals.

You can check out a tutorial on neural networks to learn more at http://pages.cs.wisc.edu/~bolo/shipyard/neural/local.html. We will use a library called **NeuroLab** throughout this chapter. Before you proceed, make sure that you install it. You can find the installation instructions at https://pythonhosted.org/neurolab/install.html. Let's go ahead and look at how to design and develop these neural networks.

Building a perceptron

Let's start our neural network adventure with a perceptron. A **perceptron** is a single neuron that performs all the computation. It is a very simple model, but it forms the basis of building up complex neural networks. Here is what it looks like:



The neuron combines the inputs using different weights, and it then adds a bias value to compute the output. It's a simple linear equation relating input values with the output of the perceptron.

How to do it…

1. Create a new Python file, and import the following packages:

```
import numpy as np
import neurolab as nl
import matplotlib.pyplot as plt
```

2. Define some input data and their corresponding labels:

```
# Define input data
data = np.array([[0.3, 0.2], [0.1, 0.4], [0.4, 0.6], [0.9,
0.5]])
labels = np.array([[0], [0], [0], [1]])
```

3. Let's plot this data to see where the datapoints are located:

```
# Plot input data
plt.figure()
plt.scatter(data[:,0], data[:,1])
plt.xlabel('X-axis')
plt.ylabel('Y-axis')
plt.title('Input data')
```

4. Let's define a `perceptron` with two inputs. This function also needs us to specify the minimum and maximum values in the input data:

```
# Define a perceptron with 2 inputs;
# Each element of the list in the first argument
```

```
# specifies the min and max values of the inputs
perceptron = nl.net.newp([[0, 1],[0, 1]], 1)
```

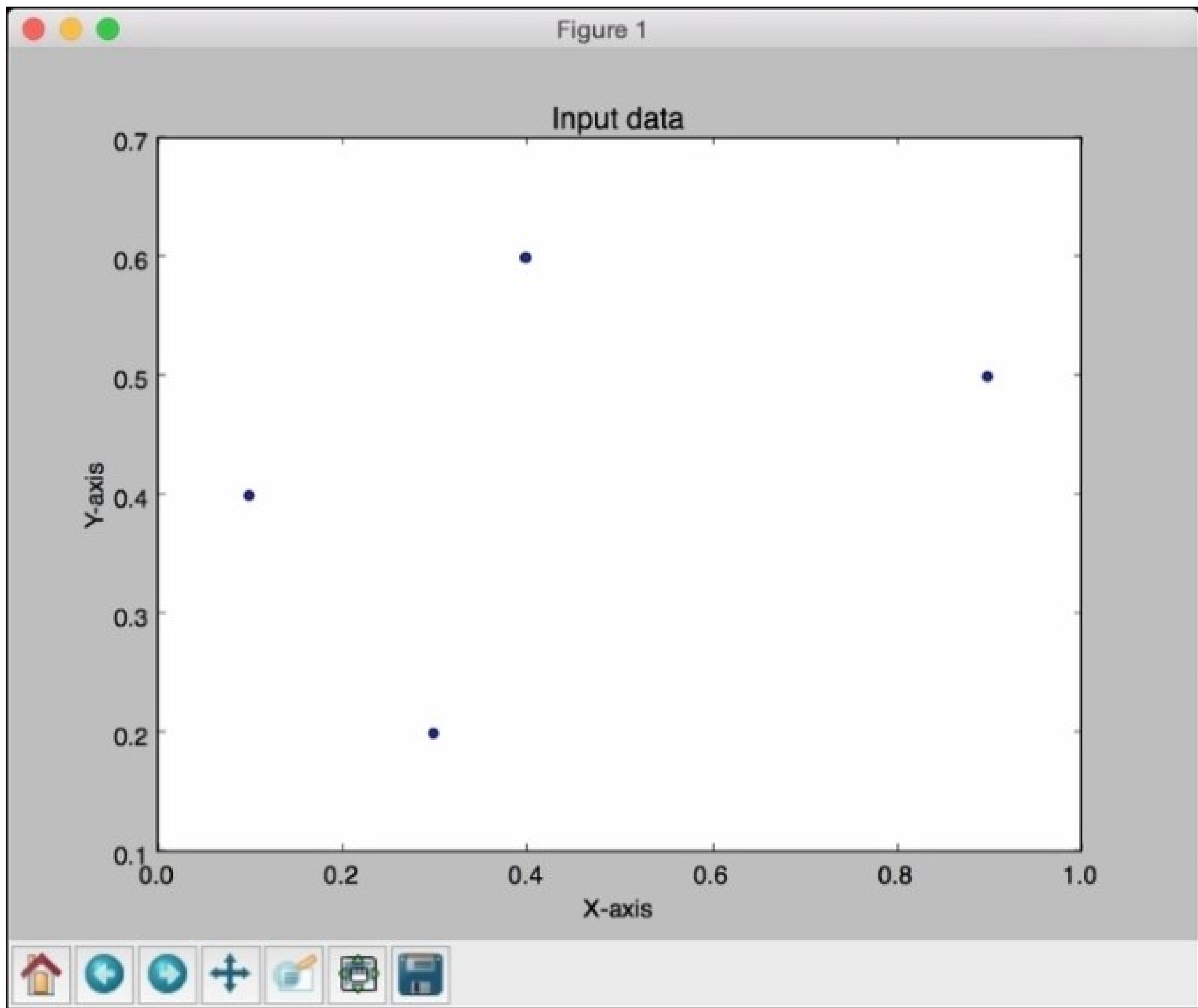
5. Let's train the perceptron. The number of epochs specifies the number of complete passes through our training dataset. The `show` parameter specifies how frequently we want to display the progress. The `lr` parameter specifies the learning rate of the perceptron. It is the step size for the algorithm to search through the parameter space. If this is large, then the algorithm may move faster, but it might miss the optimum value. If this is small, then the algorithm will hit the optimum value, but it will be slow. So it's a trade-off; hence, we choose a value of `0.01`:

```
# Train the perceptron
error = perceptron.train(data, labels, epochs=50, show=15,
lr=0.01)
```

6. Let's plot the results, as follows:

```
# plot results
plt.figure()
plt.plot(error)
plt.xlabel('Number of epochs')
plt.ylabel('Training error')
plt.grid()
plt.title('Training error progress')

plt.show()
```

7. The full code is given in the `perceptron.py` file that's already provided to you. If you run this code, you will see two figures. The first figure displays the input data:



The second figure displays the training error progress:

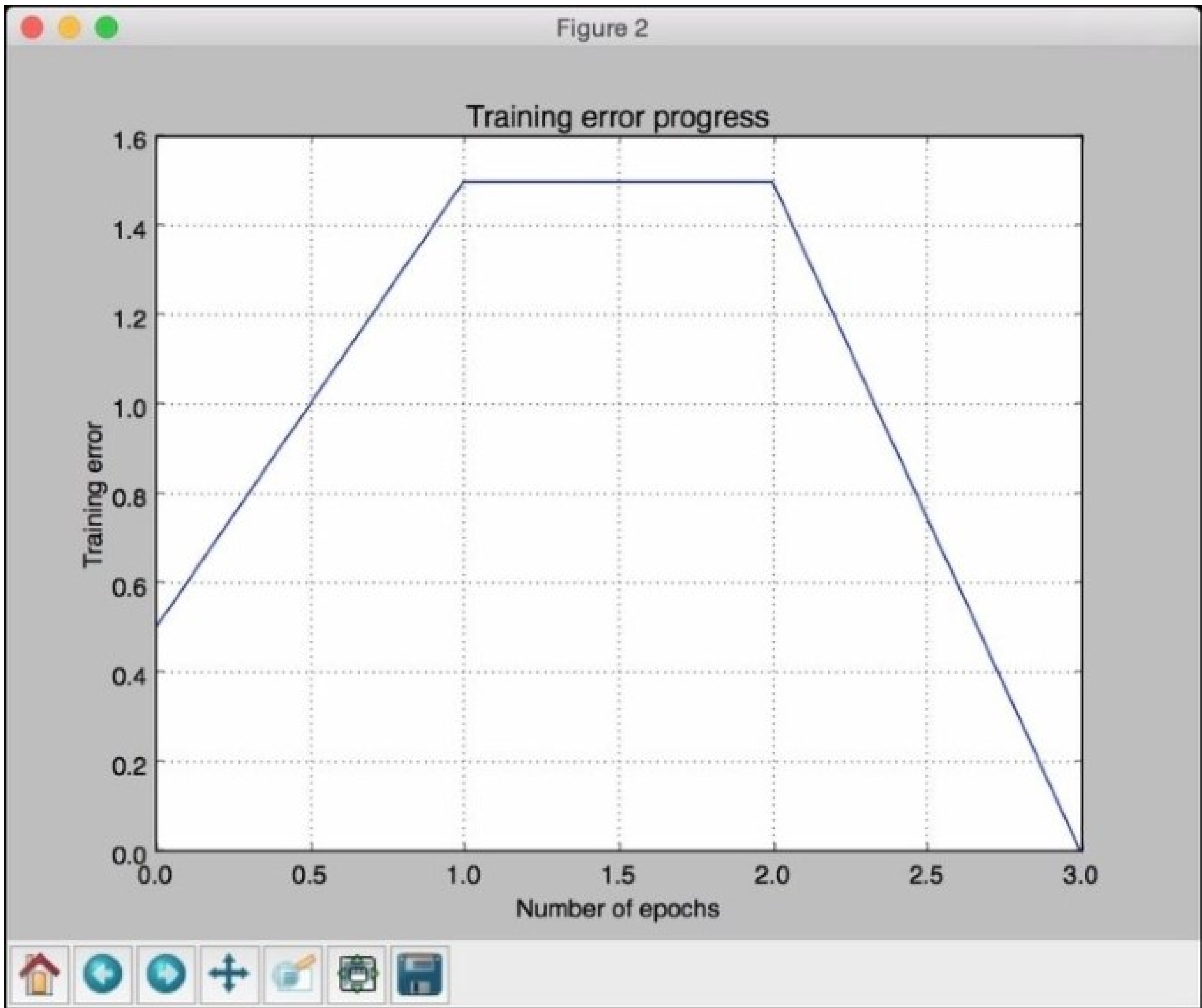


Figure 2
Training error progress
Training error
Number of epochs
1.6
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.5
1.0
1.5
2.0
2.5
3.0

Building a single layer neural network

Now that we know how to create a perceptron, let's create a single layer neural network. A single layer neural network consists of multiple neurons in a single layer. Overall, we will have an input layer, a hidden layer, and an output layer.

How to do it…

1. Create a new Python file, and import the following packages:

```
import numpy as np
import matplotlib.pyplot as plt
import neurolab as nl
```

2. We will use the data in the `data_single_layer.txt` file. Let's load this:

```
# Define input data
input_file = 'data_single_layer.txt'
input_text = np.loadtxt(input_file)
data = input_text[:, 0:2]
labels = input_text[:, 2:]
```

3. Let's plot the input data:

```
# Plot input data
plt.figure()
plt.scatter(data[:,0], data[:,1])
plt.xlabel('X-axis')
plt.ylabel('Y-axis')
plt.title('Input data')
```

4. Let's extract the minimum and maximum values:

```
# Min and max values for each dimension
x_min, x_max = data[:,0].min(), data[:,0].max()
y_min, y_max = data[:,1].min(), data[:,1].max()
```

5. Let's define a single layer neural network with two neurons in the hidden layer:

```
# Define a single-layer neural network with 2 neurons;
# Each element in the list (first argument) specifies the
# min and max values of the inputs
single_layer_net = nl.net.newp([[x_min, x_max], [y_min,
y_max]], 2)
```

6. Train the neural network until 50 epochs:

```
# Train the neural network
error = single_layer_net.train(data, labels, epochs=50,
show=20, lr=0.01)
```

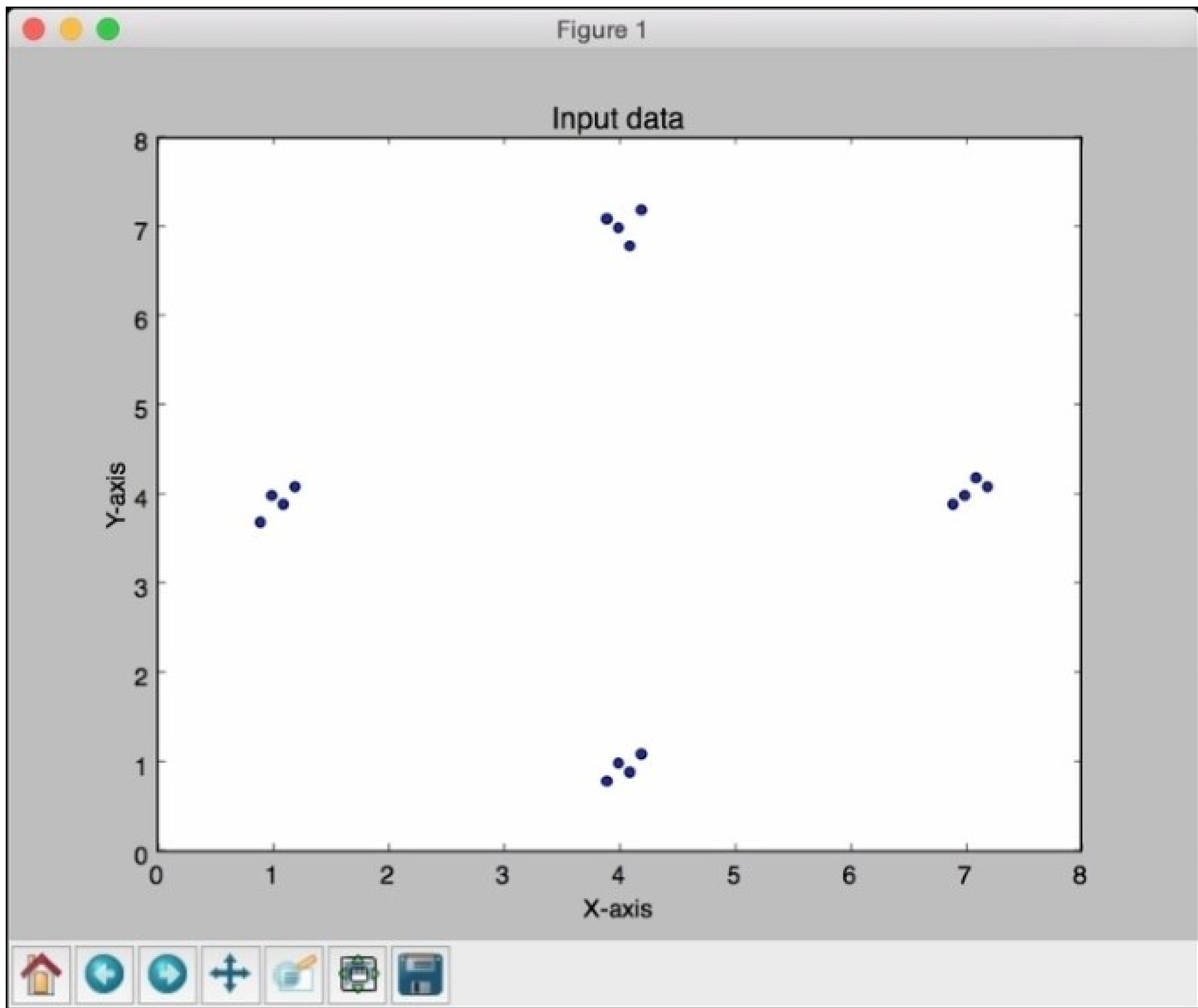
7. Plot the results, as follows:

```
# Plot results
plt.figure()
plt.plot(error)
plt.xlabel('Number of epochs')
plt.ylabel('Training error')
plt.title('Training error progress')
plt.grid()

plt.show()
```

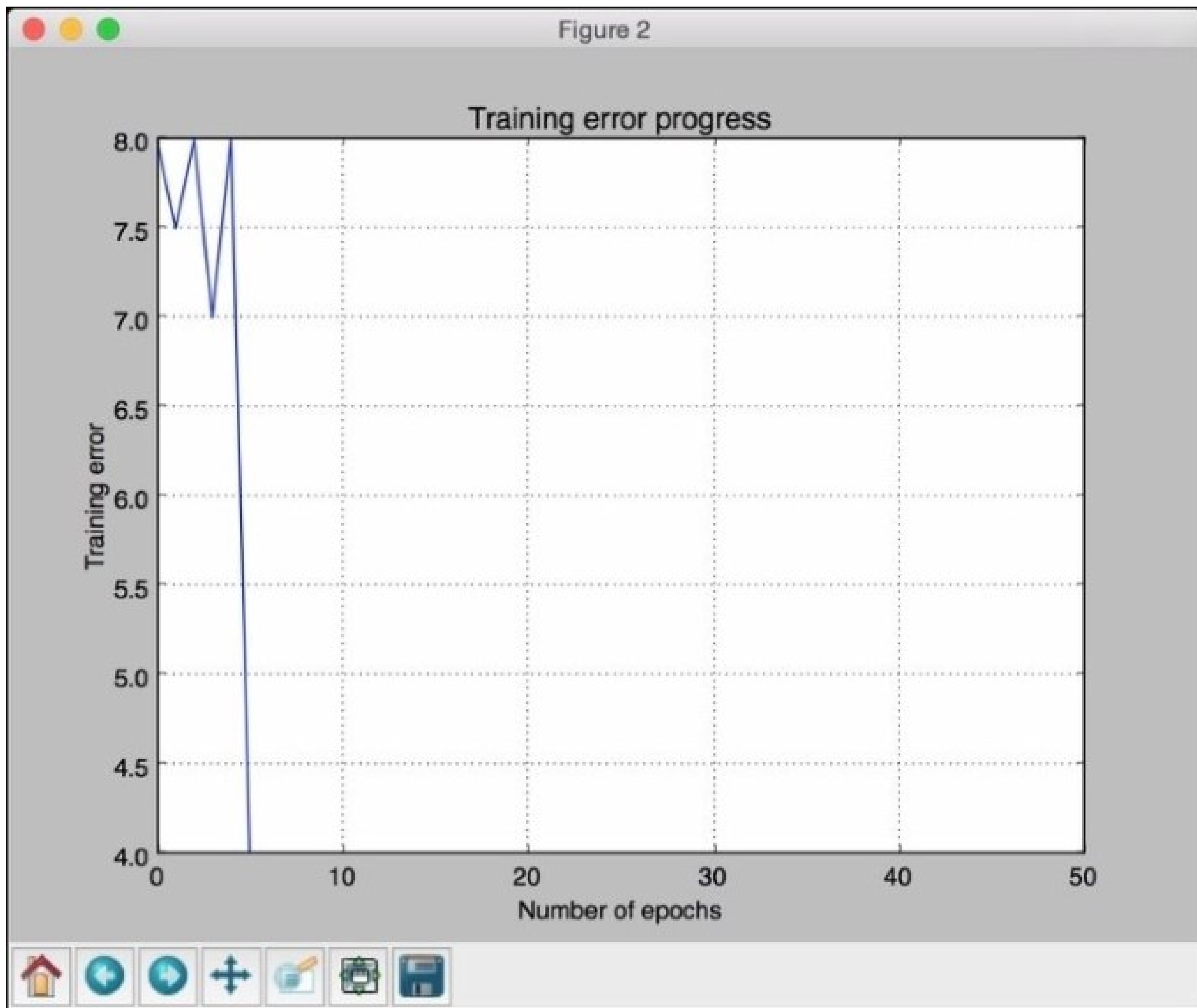
8. Let's test the neural network on new test data:

```
print single_layer_net.sim([[0.3, 4.5]])
print single_layer_net.sim([[4.5, 0.5]])
print single_layer_net.sim([[4.3, 8]])
```

9. The full code is in the `single_layer.py` file that's already provided to you. If you run this code, you will see two figures. The first figure displays the input data:



The second figure displays the training error progress:



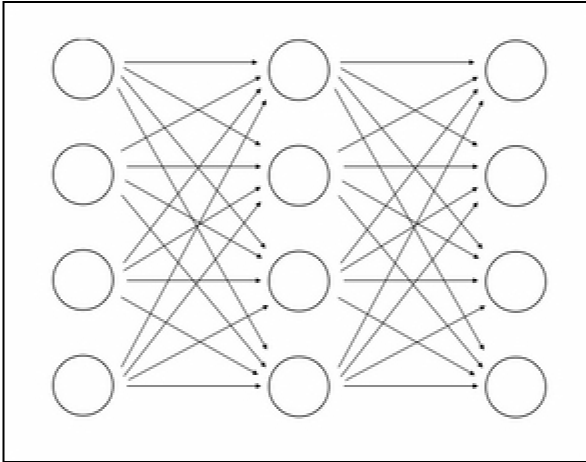
You will see the following printed on your Terminal, indicating where the input test points belong:

```
[[ 0.  0.]]
[[ 1.  0.]]
[[ 1.  1.]]
```

You can verify that the outputs are correct based on our labels.

Building a deep neural network

We are now ready to build a deep neural network. A deep neural network consists of an input layer, many hidden layers, and an output layer. This looks like the following:



The preceding figure depicts a multilayer neural network with one input layer, one hidden layer, and one output layer. In a deep neural network, there are many hidden layers between the input and the output layers.

How to do it…

1. Create a new Python file, and import the following packages:

```
import neurolab as nl
import numpy as np
import matplotlib.pyplot as plt
```

2. Let's define parameters to generate some training data:

```
# Generate training data
min_value = -12
max_value = 12
num_datapoints = 90
```

3. This training data will consist of a function that we define that will transform the values. We expect the neural network to learn this on its own, based on the input and output values that we provide:

```
x = np.linspace(min_value, max_value, num_datapoints)
y = 2 * np.square(x) + 7
y /= np.linalg.norm(y)
```

4. Reshape the arrays:

```
data = x.reshape(num_datapoints, 1)
labels = y.reshape(num_datapoints, 1)
```

5. Plot input data:

```
# Plot input data
plt.figure()
plt.scatter(data, labels)
plt.xlabel('X-axis')
plt.ylabel('Y-axis')
plt.title('Input data')
```

6. Define a deep neural network with two hidden layers, where each hidden layer consists of 10 neurons:

```
# Define a multilayer neural network with 2 hidden layers;
# Each hidden layer consists of 10 neurons and the output layer
# consists of 1 neuron
multilayer_net = nl.net.newff([[min_value, max_value]], [10,
10, 1])
```

7. Set the training algorithm to **gradient descent** (you can learn more about it at https://spin.atomicobject.com/2014/06/24/gradient-descent-linear-regression):

```
# Change the training algorithm to gradient descent
multilayer_net.trainf = nl.train.train_gd
```

8. Train the network:

```
# Train the network
error = multilayer_net.train(data, labels, epochs=800,
show=100, goal=0.01)
```

9. Run the network on training data to see the performance:

```
# Predict the output for the training inputs
predicted_output = multilayer_net.sim(data)
```

10. Plot the training error:

```
# Plot training error
plt.figure()
plt.plot(error)
plt.xlabel('Number of epochs')
plt.ylabel('Error')
plt.title('Training error progress')
```

11. Let's create a set of new inputs and run the neural network on them to see how it performs:

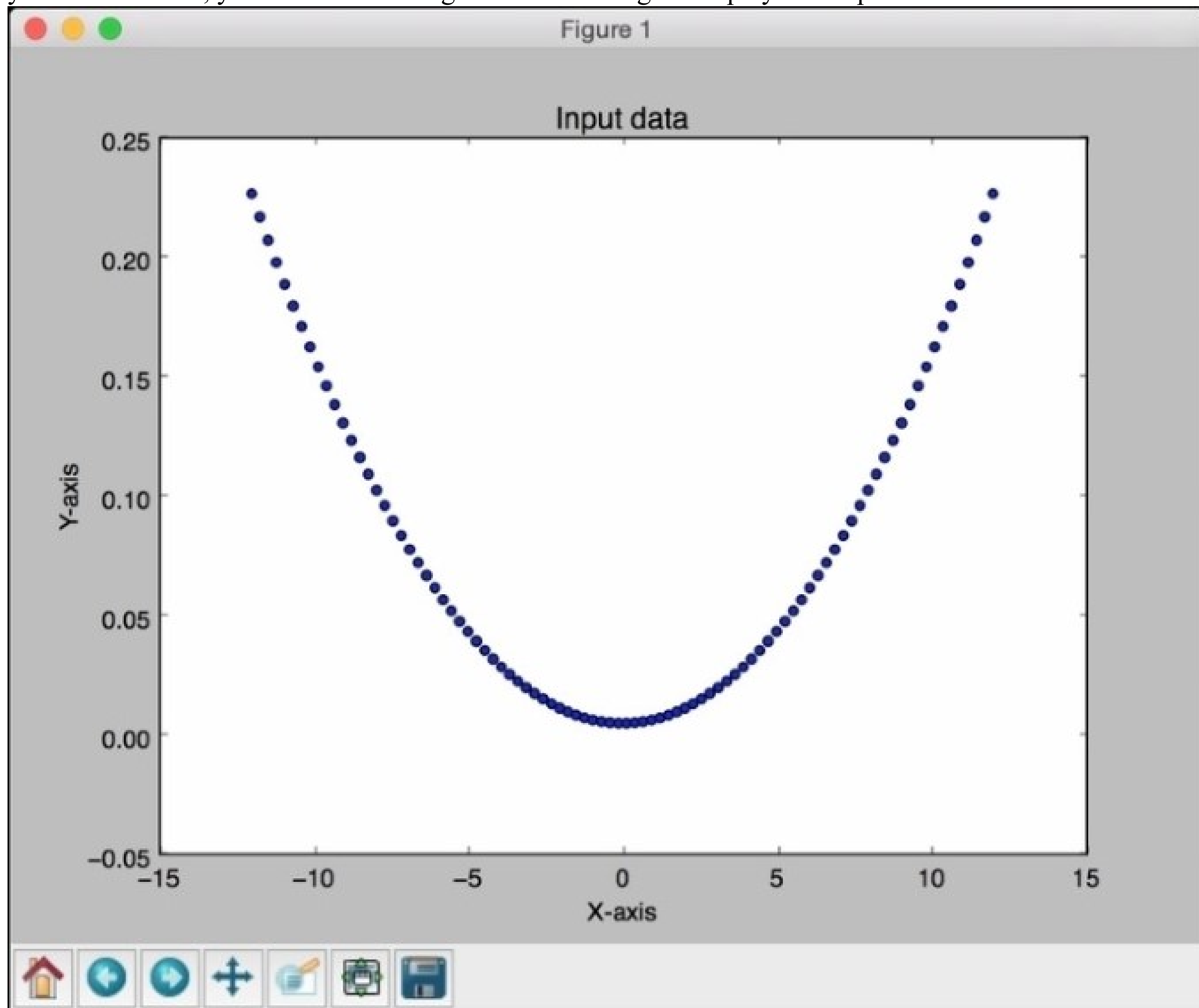
```
# Plot predictions
x2 = np.linspace(min_value, max_value, num_datapoints * 2)
```

```
y2 = multilayer_net.sim(x2.reshape(x2.size,1)).reshape(x2.size)
y3 = predicted_output.reshape(num_datapoints)
```

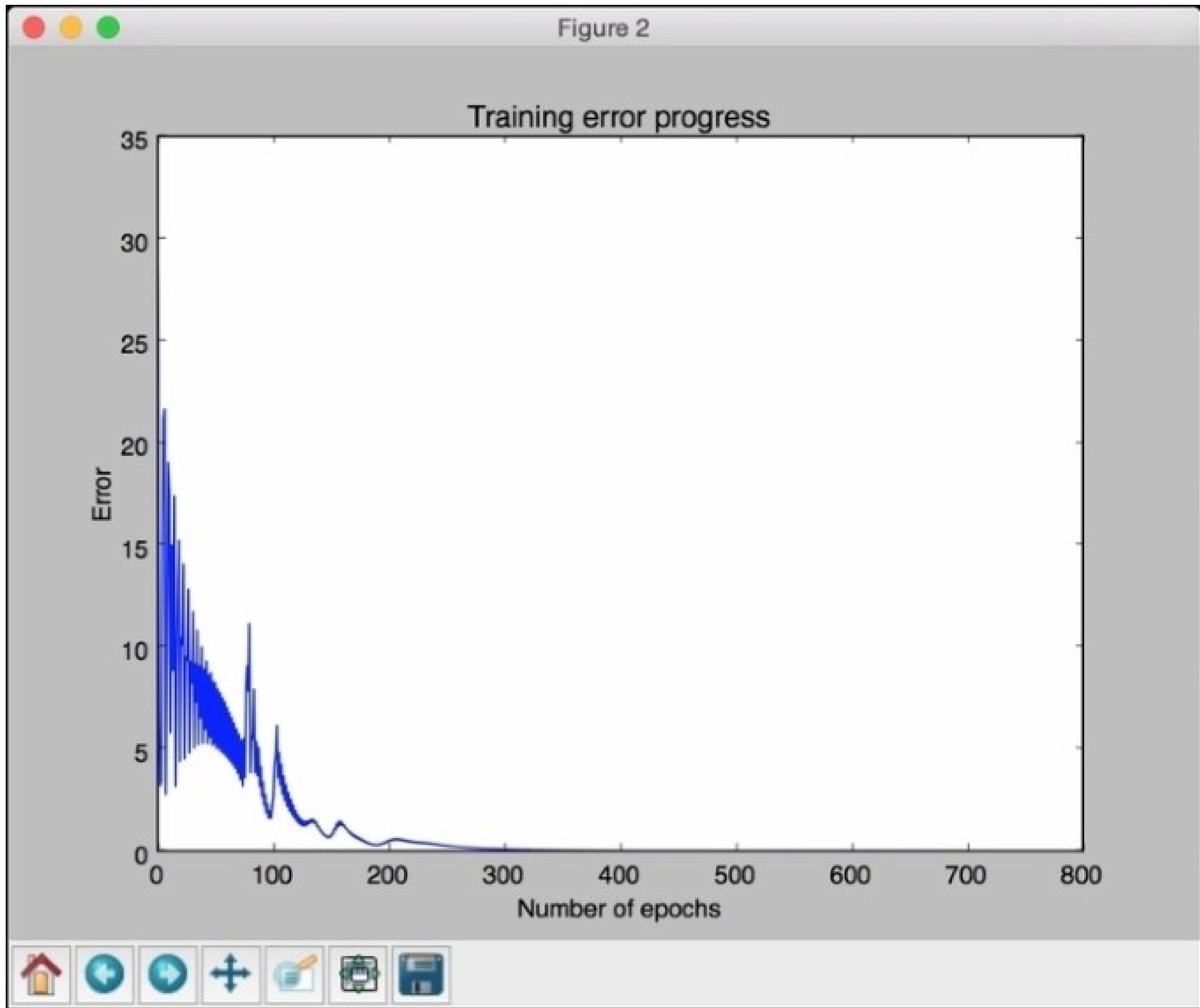
12. Plot the outputs:

```
plt.figure()
plt.plot(x2, y2, '-', x, y, '.', x, y3, 'p')
plt.title('Ground truth vs predicted output')

plt.show()
```

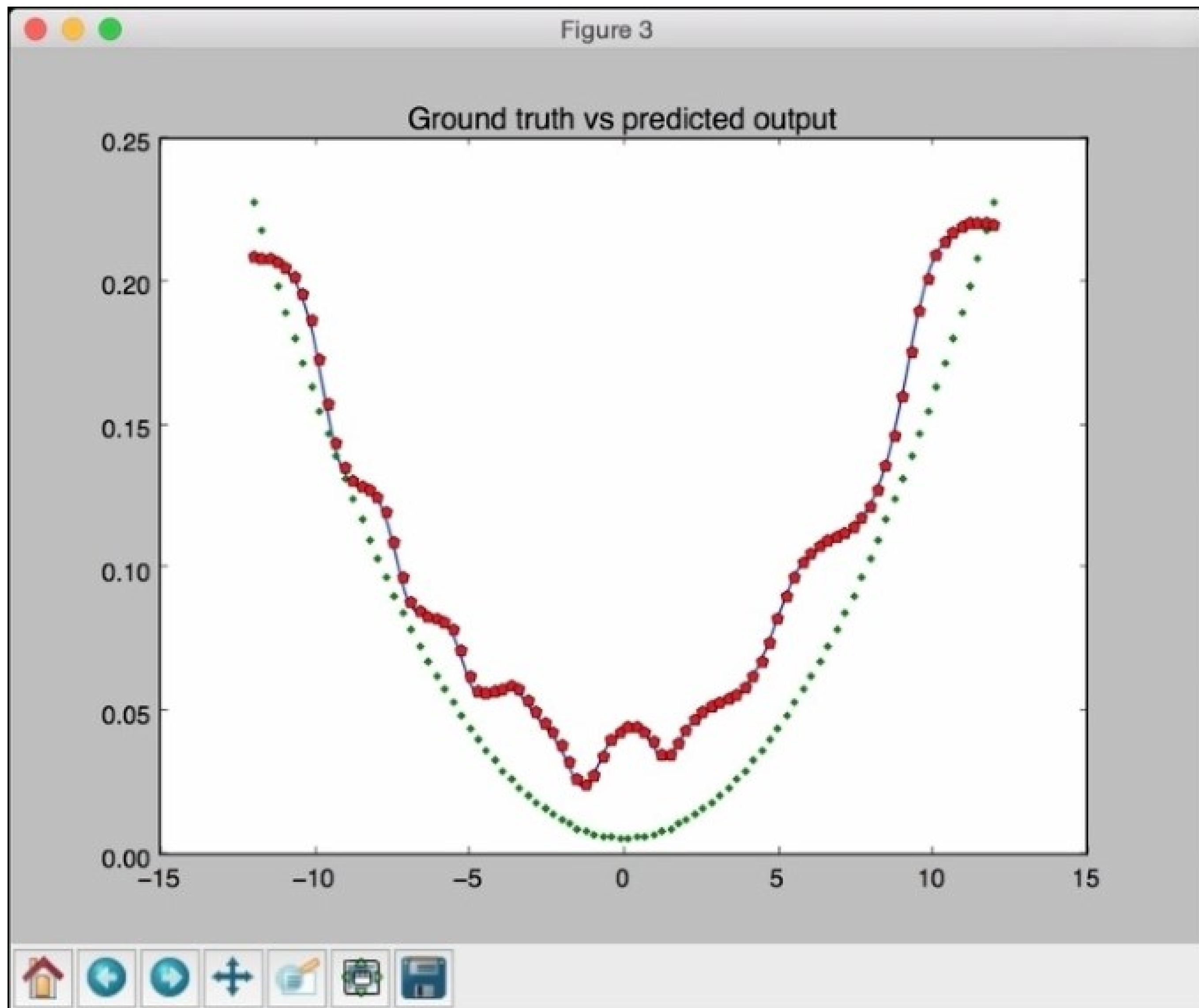
13. The full code is in the `deep_neural_network.py` file that's already provided to you. If you run this code, you will see three figures. The first figure displays the input data:



The second figure displays the training error progress:



The third figure displays the output of the neural network:



You will see the following on your Terminal:

```
Epoch: 100; Error: 1.64795788647;
Epoch: 200; Error: 0.517736068801;
Epoch: 300; Error: 0.13545620002;
Epoch: 400; Error: 0.0521272422892;
Epoch: 500; Error: 0.0465021594702;
Epoch: 600; Error: 0.0483261849312;
Epoch: 700; Error: 0.0431681554217;
Epoch: 800; Error: 0.0346446191022;
The maximum number of train epochs is reached
```

Creating a vector quantizer

You can use neural networks for vector quantization as well. **Vector quantization** is the *N*-dimensional version of "rounding off". This is very commonly used across multiple areas in computer vision, natural language processing, and machine learning in general.

How to do it…

1. Create a new Python file, and import the following packages:

```
import numpy as np
import matplotlib.pyplot as plt
import neurolab as nl
```

2. Let's load the input data from the `data_vq.txt` file:

```
# Define input data
input_file = 'data_vq.txt'
input_text = np.loadtxt(input_file)
data = input_text[:, 0:2]
labels = input_text[:, 2:]
```

3. Define a **learning vector quantization** (**LVQ**) neural network with two layers. The array in the last parameter specifies the percentage weightage to each output (they should sum up to 1):

```
# Define a neural network with 2 layers:
# 10 neurons in input layer and 4 neurons in output layer
net = nl.net.newlvq(nl.tool.minmax(data), 10, [0.25, 0.25,
0.25, 0.25])
```

4. Train the LVQ neural network:

```
# Train the neural network
error = net.train(data, labels, epochs=100, goal=-1)
```

5. Create a grid of values for testing and visualization:

```
# Create the input grid
xx, yy = np.meshgrid(np.arange(0, 8, 0.2), np.arange(0, 8, 0.2))
xx.shape = xx.size, 1
yy.shape = yy.size, 1
input_grid = np.concatenate((xx, yy), axis=1)
```

6. Evaluate the network on this grid:

```
# Evaluate the input grid of points
output_grid = net.sim(input_grid)
```

7. Define the four classes in our data:

```
# Define the 4 classes
class1 = data[labels[:,0] == 1]
class2 = data[labels[:,1] == 1]
class3 = data[labels[:,2] == 1]
class4 = data[labels[:,3] == 1]
```

8. Define the grids for all these classes:

```
# Define grids for all the 4 classes
grid1 = input_grid[output_grid[:,0] == 1]
grid2 = input_grid[output_grid[:,1] == 1]
grid3 = input_grid[output_grid[:,2] == 1]
grid4 = input_grid[output_grid[:,3] == 1]
```

9. Plot the outputs:

```
# Plot outputs
plt.plot(class1[:,0], class1[:,1], 'ko', class2[:,0],
class2[:,1], 'ko',
                class3[:,0], class3[:,1], 'ko', class4[:,0],
class4[:,1], 'ko')
plt.plot(grid1[:,0], grid1[:,1], 'b.', grid2[:,0], grid2[:,1],
'gx',
                grid3[:,0], grid3[:,1], 'cs', grid4[:,0],
grid4[:,1], 'ro')
plt.axis([0, 8, 0, 8])
plt.xlabel('X-axis')
plt.ylabel('Y-axis')
plt.title('Vector quantization using neural networks')

plt.show()
```

10. The full code is in the `vector_quantization.py` file that's already provided to you. If you run this code, you will see the following figure where the space is divided into regions. Each region corresponds to a bucket in the list of vector-quantized regions in the space:

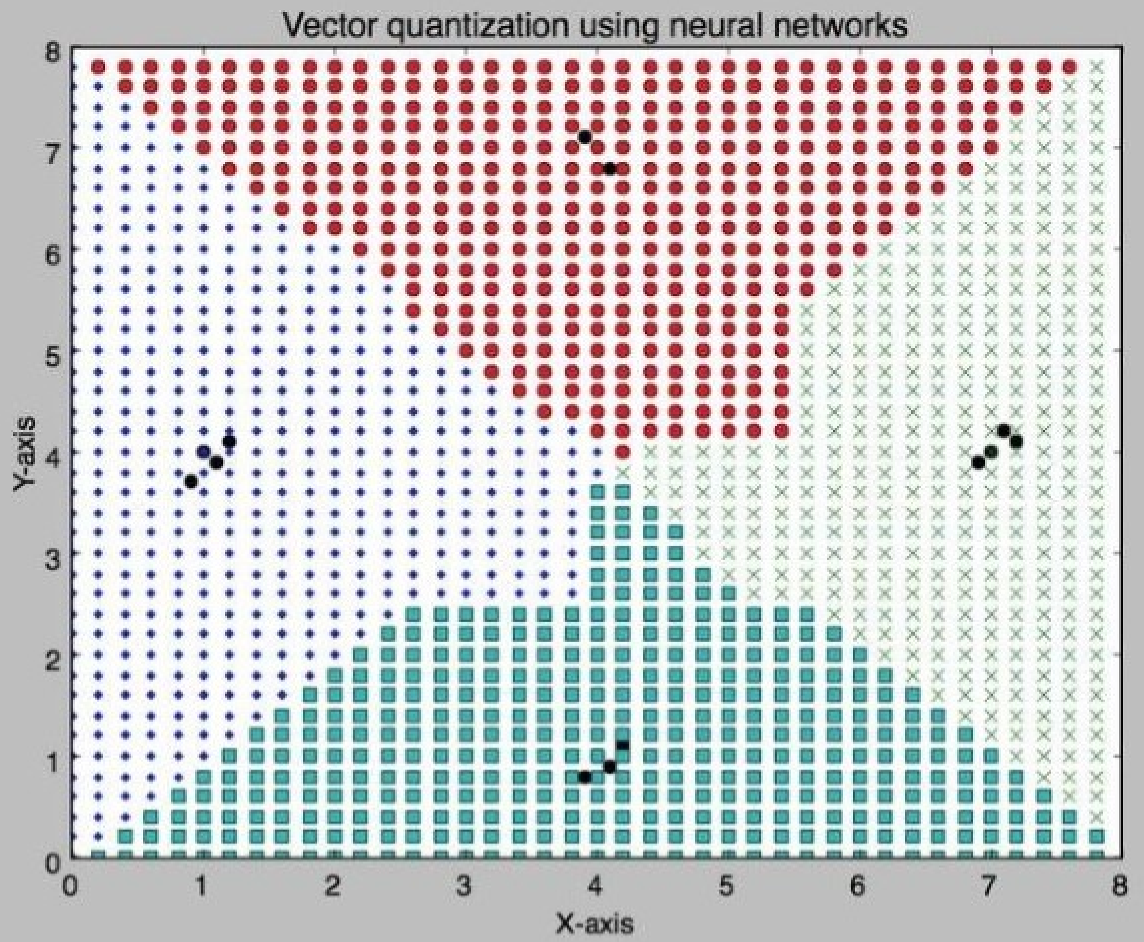


Figure 1
Vector quantization using neural networks
Y-axis
X-axis
0
1
2
3
4
5
6
7
8

Building a recurrent neural network for sequential data analysis

Recurrent neural networks are really good at analyzing sequential and time-series data. You can learn more about them at http://www.wildml.com/2015/09/recurrent-neural-networks-tutorial-part-1-introduction-to-rnns. When we deal with sequential and time-series data, we cannot just extend generic models. The temporal dependencies in the data are really important, and we need to account for this in our models. Let's look at how to build them.

How to do it…

1. Create a new Python file, and import the following packages:

```
import numpy as np
import matplotlib.pyplot as plt
import neurolab as nl
```

2. Define a function to create a waveform, based on input parameters:

```
def create_waveform(num_points):
    # Create train samples
    data1 = 1 * np.cos(np.arange(0, num_points))
    data2 = 2 * np.cos(np.arange(0, num_points))
    data3 = 3 * np.cos(np.arange(0, num_points))
    data4 = 4 * np.cos(np.arange(0, num_points))
```

3. Create different amplitudes for each interval to create a random waveform:

```
    # Create varying amplitudes
    amp1 = np.ones(num_points)
    amp2 = 4 + np.zeros(num_points)
    amp3 = 2 * np.ones(num_points)
    amp4 = 0.5 + np.zeros(num_points)
```

4. Combine the arrays to create the output arrays. This data corresponds to the input and the amplitude corresponds to the labels:

```
    data = np.array([data1, data2, data3,
data4]).reshape(num_points * 4, 1)
    amplitude = np.array([[amp1, amp2, amp3,
amp4]]).reshape(num_points * 4, 1)

    return data, amplitude
```

5. Define a function to draw the output after passing the data through the trained neural network:

```
# Draw the output using the network
def draw_output(net, num_points_test):
```

```
    data_test, amplitude_test = create_waveform(num_points_test)
    output_test = net.sim(data_test)
    plt.plot(amplitude_test.reshape(num_points_test * 4))
    plt.plot(output_test.reshape(num_points_test * 4))
```

6. Define the `main` function and start by creating sample data:

```
if __name__=='__main__':
    # Get data
    num_points = 30
    data, amplitude = create_waveform(num_points)
```

7. Create a recurrent neural network with two layers:

```
    # Create network with 2 layers
    net = nl.net.newelm([[-2, 2]], [10, 1], [nl.trans.TanSig(),
nl.trans.PureLin()])
```

8. Set the initialized functions for each layer:

```
    # Set initialized functions and init
    net.layers[0].initf = nl.init.InitRand([-0.1, 0.1], 'wb')
    net.layers[1].initf= nl.init.InitRand([-0.1, 0.1], 'wb')
    net.init()
```

9. Train the recurrent neural network:

```
    # Training the recurrent neural network
    error = net.train(data, amplitude, epochs=1000, show=100,
goal=0.01)
```

10. Compute the output from the network for the training data:

```
    # Compute output from network
    output = net.sim(data)
```

11. Plot training error:

```
    # Plot training results
    plt.subplot(211)
    plt.plot(error)
    plt.xlabel('Number of epochs')
    plt.ylabel('Error (MSE)')
```

12. Plot the results:

```
    plt.subplot(212)
    plt.plot(amplitude.reshape(num_points * 4))
    plt.plot(output.reshape(num_points * 4))
    plt.legend(['Ground truth', 'Predicted output'])
```

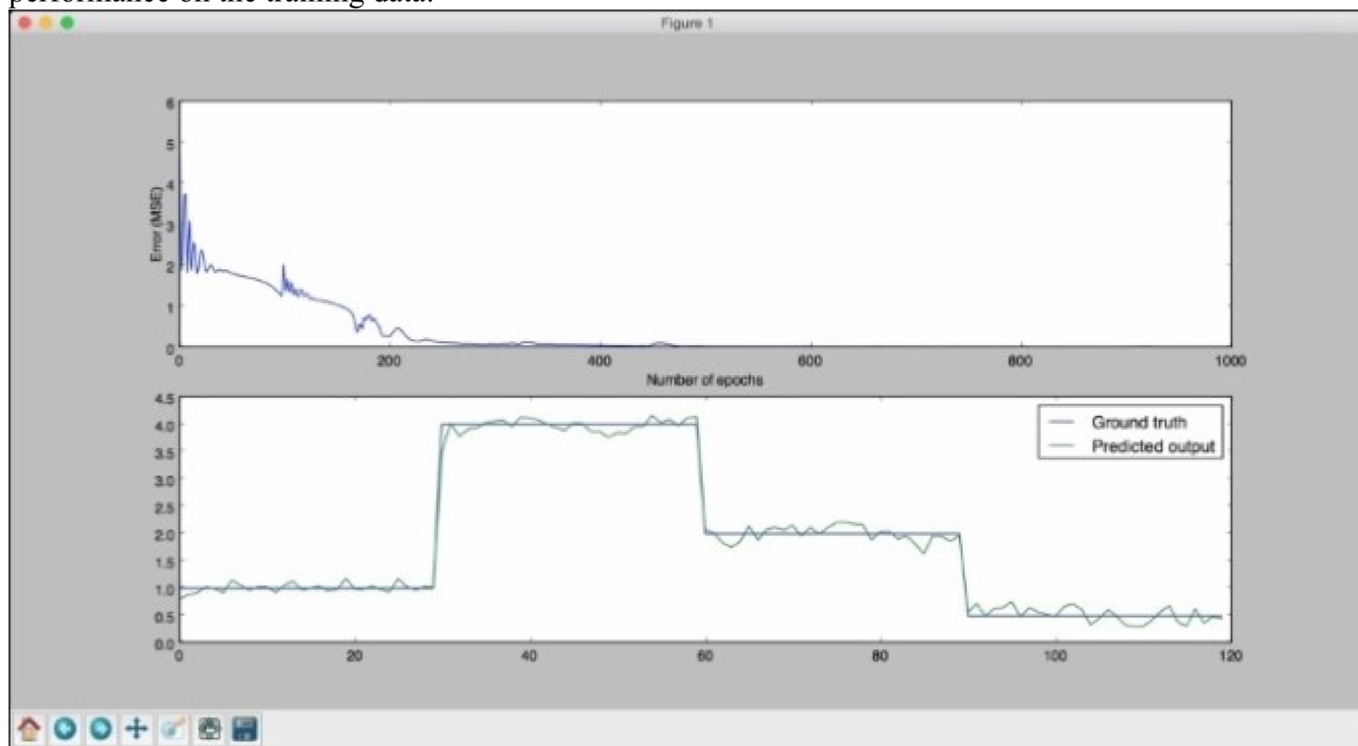
13. Create a waveform of random length and see whether the network can predict it:

```
# Testing on unknown data at multiple scales
plt.figure()

plt.subplot(211)
draw_output(net, 74)
plt.xlim([0, 300])
```

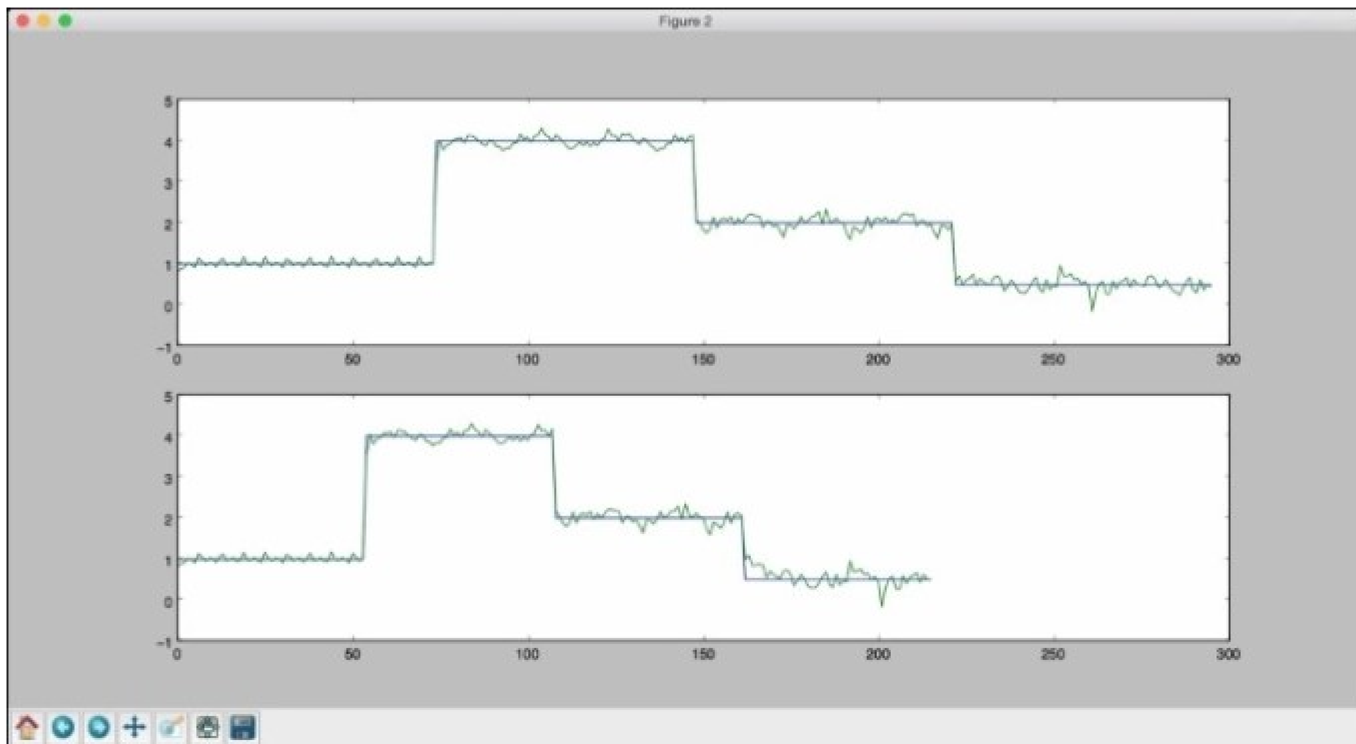
14. Create another waveform of a shorter length and see whether the network can predict it:

```
plt.subplot(212)
draw_output(net, 54)
plt.xlim([0, 300])

plt.show()
```

15. The full code is in the `recurrent_network.py` file that's already provided to you. If you run this code, you will see two figures. The first figure displays training errors and the performance on the training data:



The second figure displays how a trained recurrent neural net performs on sequences of arbitrary lengths:



You will see the following on your Terminal:

```
Epoch: 100; Error: 2.0202165367;
Epoch: 200; Error: 0.276370891;
Epoch: 300; Error: 0.0902055024828;
Epoch: 400; Error: 0.0662254210369;
Epoch: 500; Error: 0.0291456739963;
Epoch: 600; Error: 0.0274479103273;
Epoch: 700; Error: 0.0221256973779;
Epoch: 800; Error: 0.0227723305931;
Epoch: 900; Error: 0.0207200477057;
Epoch: 1000; Error: 0.0159299080472;
The maximum number of train epochs is reached
```

Visualizing the characters in an optical character recognition database

We will now look at how to use neural networks to perform optical character recognition. This refers to the process of identifying handwritten characters in images. We will use the dataset available at http://ai.stanford.edu/~btaskar/ocr. The default file name after downloading is `letter.data`. To start with, let's see how to interact with the data and visualize it.

How to do it…

1. Create a new Python file, and import the following packages:

```
import os
import sys

import cv2
import numpy as np
```

2. Define the input file name:

```
# Load input data
input_file = 'letter.data'
```

3. Define visualization parameters:

```
# Define visualization parameters
scaling_factor = 10
start_index = 6
end_index = -1
h, w = 16, 8
```

4. Keep looping through the file until the user presses the *Esc* key. Split the line into tab-separated characters:

```
# Loop until you encounter the Esc key
with open(input_file, 'r') as f:
    for line in f.readlines():
        data = np.array([255*float(x) for x in
line.split('\t')[start_index:end_index]])
```

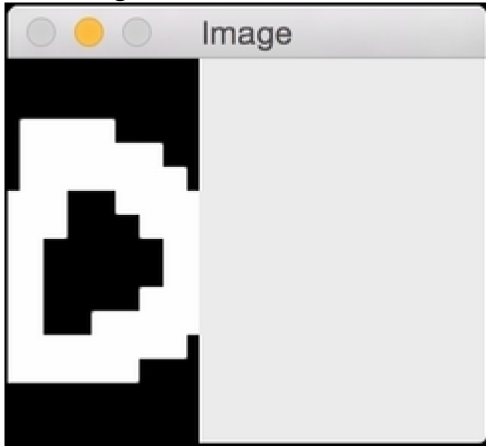
5. Reshape the array into the required shape, resize it, and display it:

```
        img = np.reshape(data, (h,w))
        img_scaled = cv2.resize(img, None, fx=scaling_factor,
fy=scaling_factor)
        cv2.imshow('Image', img_scaled)
```

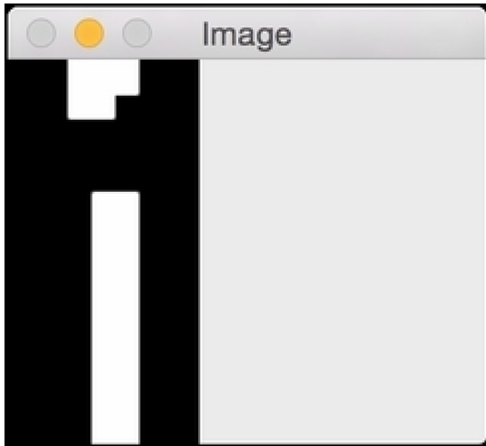
6. If the user presses *Esc*, break the loop:

```
        c = cv2.waitKey()
        if c == 27:
            break
```

7. The full code is in the `visualize_characters.py` file that's already provided to you. If you run this code, you will see a window displaying characters. For example, *o* looks like the following:



The character *i* looks like the following:



Building an optical character recognizer using neural networks

Now that we know how to interact with the data, let's build a neural network-based optical character-recognition system.

How to do it…

1. Create a new Python file, and import the following packages:

```
import numpy as np
import neurolab as nl
```

2. Define the input filename:

```
# Input file
input_file = 'letter.data'
```

3. When we work with neural networks that deal with large amounts of data, it takes a lot of time to train. To demonstrate how to build this system, we will take only `20` datapoints:

```
# Number of datapoints to load from the input file
num_datapoints = 20
```

4. If you look at the data, you will see that there are seven distinct characters in the first 20 lines. Let's define them:

```
# Distinct characters
orig_labels = 'omandig'

# Number of distinct characters
num_output = len(orig_labels)
```

5. We will use 90% of the data for training and remaining 10% for testing. Define the training and testing parameters:

```
# Training and testing parameters
num_train = int(0.9 * num_datapoints)
num_test = num_datapoints - num_train
```

6. The starting and ending indices in each line of the dataset file:

```
# Define dataset extraction parameters
start_index = 6
end_index = -1
```

7. Create the dataset:

```
# Creating the dataset
data = []
labels = []
with open(input_file, 'r') as f:
    for line in f.readlines():
        # Split the line tabwise
        list_vals = line.split('\t')
```

8. Add an error check to see whether the characters are in our list of labels:

```
        # If the label is not in our ground truth labels, skip
it
        if list_vals[1] not in orig_labels:
            continue
```

9. Extract the label, and append it the main list:

```
        # Extract the label and append it to the main list
        label = np.zeros((num_output, 1))
        label[orig_labels.index(list_vals[1])] = 1
        labels.append(label)
```

10. Extract the character, and append it to the main list:

```
        # Extract the character vector and append it to the
main list
        cur_char = np.array([float(x) for x in
list_vals[start_index:end_index]])
        data.append(cur_char)
```

11. Exit the loop once we have enough data:

```
        # Exit the loop once the required dataset has been
loaded
        if len(data) >= num_datapoints:
            break
```

12. Convert this data into NumPy arrays:

```
# Convert data and labels to numpy arrays
data = np.asfarray(data)
labels = np.array(labels).reshape(num_datapoints, num_output)
```

13. Extract the number of dimensions in our data:

```
# Extract number of dimensions
num_dims = len(data[0])
```

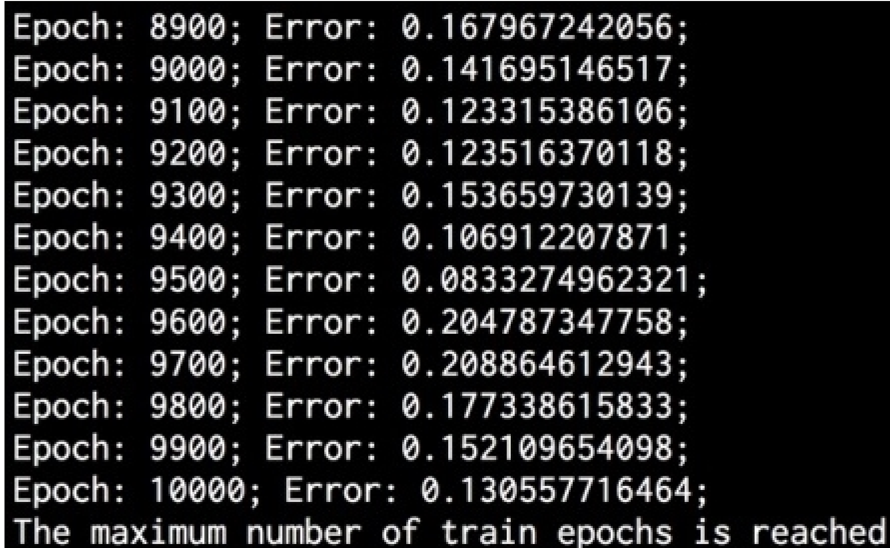
14. Train the neural network until `10,000` epochs:

```
# Create and train neural network
net = nl.net.newff([[0, 1] for _ in range(len(data[0]))], [128,
16, num_output])
net.trainf = nl.train.train_gd
error = net.train(data[:num_train,:], labels[:num_train,:],
epochs=10000,
        show=100, goal=0.01)
```

15. Predict the output for test inputs:

```
# Predict the output for test inputs
predicted_output = net.sim(data[num_train:, :])
print "\nTesting on unknown data:"
for i in range(num_test):
    print "\nOriginal:", orig_labels[np.argmax(labels[i])]
    print "Predicted:",
orig_labels[np.argmax(predicted_output[i])]
```

16. The full code is in the `ocr.py` file that's already provided to you. If you run this code, you will see the following on your Terminal at the end of training:



The output of the neural network is shown in the following screenshot:

```
Testing on unknown data:

Original: o
Predicted: o

Original: m
Predicted: m
```

Chapter 12. Visualizing Data

In this chapter, we will cover the following recipes:

- Plotting 3D scatter plots
- Plotting bubble plots
- Animating bubble plots
- Drawing pie charts
- Plotting date-formatted time series data
- Plotting histograms
- Visualizing heat maps
- Animating dynamic signals

Introduction

Data visualization is an important pillar of machine learning. It helps us formulate the right strategies to understand data. Visual representation of data assists us in choosing the right algorithms. One of the main goals of data visualization is to communicate clearly using graphs and charts. These graphs help us communicate information clearly and efficiently.

We encounter numerical data all the time in the real world. We want to encode this numerical data using graphs, lines, dots, bars, and so on to visually display the information contained in those numbers. This makes complex distributions of data more understandable and usable. This process is used in a variety of situations, including comparative analysis, tracking growth, market distribution, public opinion polls, and many others.

We use different charts to show patterns or relationships between variables. We use histograms to display the distribution of data. We use tables when we want to look up a specific measurement. In this chapter, we will look at various scenarios and discuss what visualizations we can use in these situations.

Plotting 3D scatter plots

In this recipe, we will learn how to plot 3D scatterplots and visualize them in three dimensions.

How to do it…

1. Create a new Python file, and import the following packages:

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
```

2. Create the empty figure:

```
# Create the figure
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
```

3. Define the number of values that we should generate:

```
# Define the number of values
n = 250
```

4. Create a `lambda` function to generate values in a given range:

```
# Create a lambda function to generate the random values in the
given range
f = lambda minval, maxval, n: minval + (maxval - minval) *
np.random.rand(n)
```

5. Generate X, Y, and Z values using this function:

```
# Generate the values
x_vals = f(15, 41, n)
y_vals = f(-10, 70, n)
z_vals = f(-52, -37, n)
```

6. Plot these values:

```
# Plot the values
ax.scatter(x_vals, y_vals, z_vals, c='k', marker='o')
ax.set_xlabel('X axis')
ax.set_ylabel('Y axis')
ax.set_zlabel('Z axis')

plt.show()
```

7. The full code is in the `scatter_3d.py` file that's already provided to you. If you run this code, you will see the following figure:

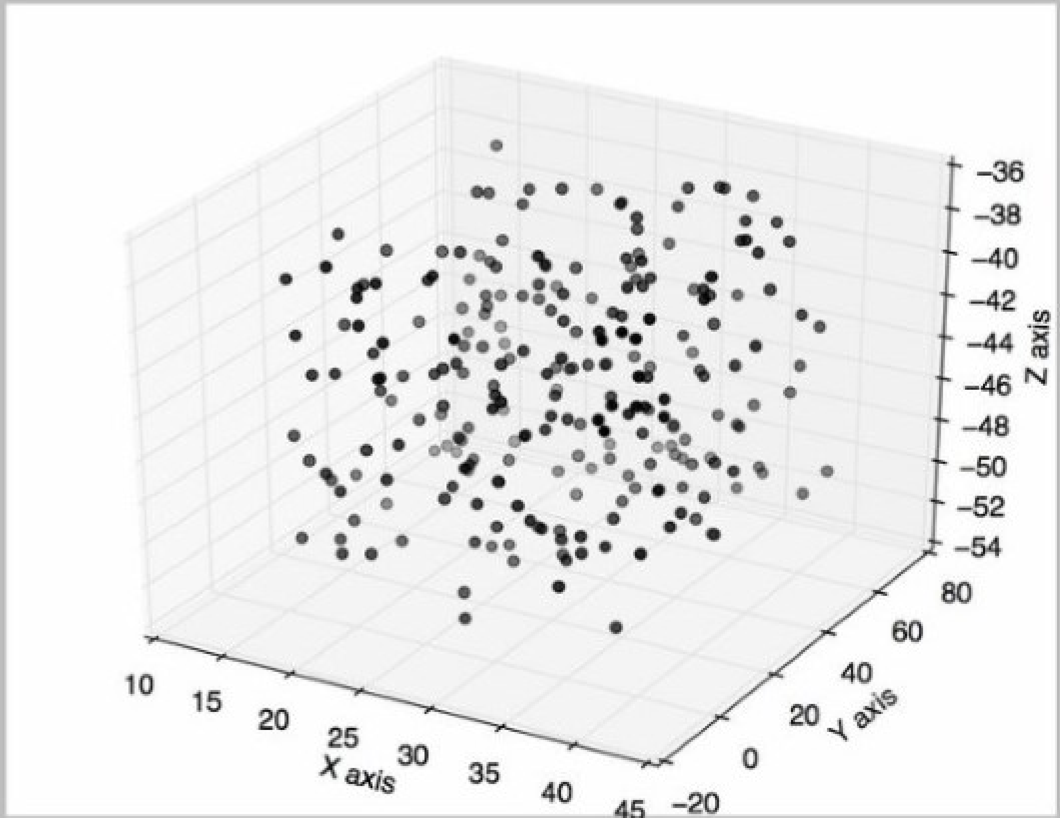


Figure 1
X axis
Y axis
Z axis
10
15
20
25
30
35
40
45
−20
0
20
40
60
80
−36
−38
−40
−42
−44
−46
−48
−50
−52
−54

Plotting bubble plots

Let's see how to plot bubble plots. The size of each circle in a 2D bubble plot represents the amplitude of that particular point.

How to do it…

1. Create a new Python file, and import the following packages:

```
import numpy as np
import matplotlib.pyplot as plt
```

2. Define the number of values that we should generate:

```
# Define the number of values
num_vals = 40
```

3. Generate random values for `x` and `y`:

```
# Generate random values
x = np.random.rand(num_vals)
y = np.random.rand(num_vals)
```

4. Define the area value for each point in the bubble plot:

```
# Define area for each bubble
# Max radius is set to a specified value
max_radius = 25
area = np.pi * (max_radius * np.random.rand(num_vals)) ** 2
```

5. Define the colors:

```
# Generate colors
colors = np.random.rand(num_vals)
```

6. Plot these values:

```
# Plot the points
plt.scatter(x, y, s=area, c=colors, alpha=1.0)

plt.show()
```

7. The full code is in the `bubble_plot.py` file that's already provided to you. If you run this code, you will see the following figure:

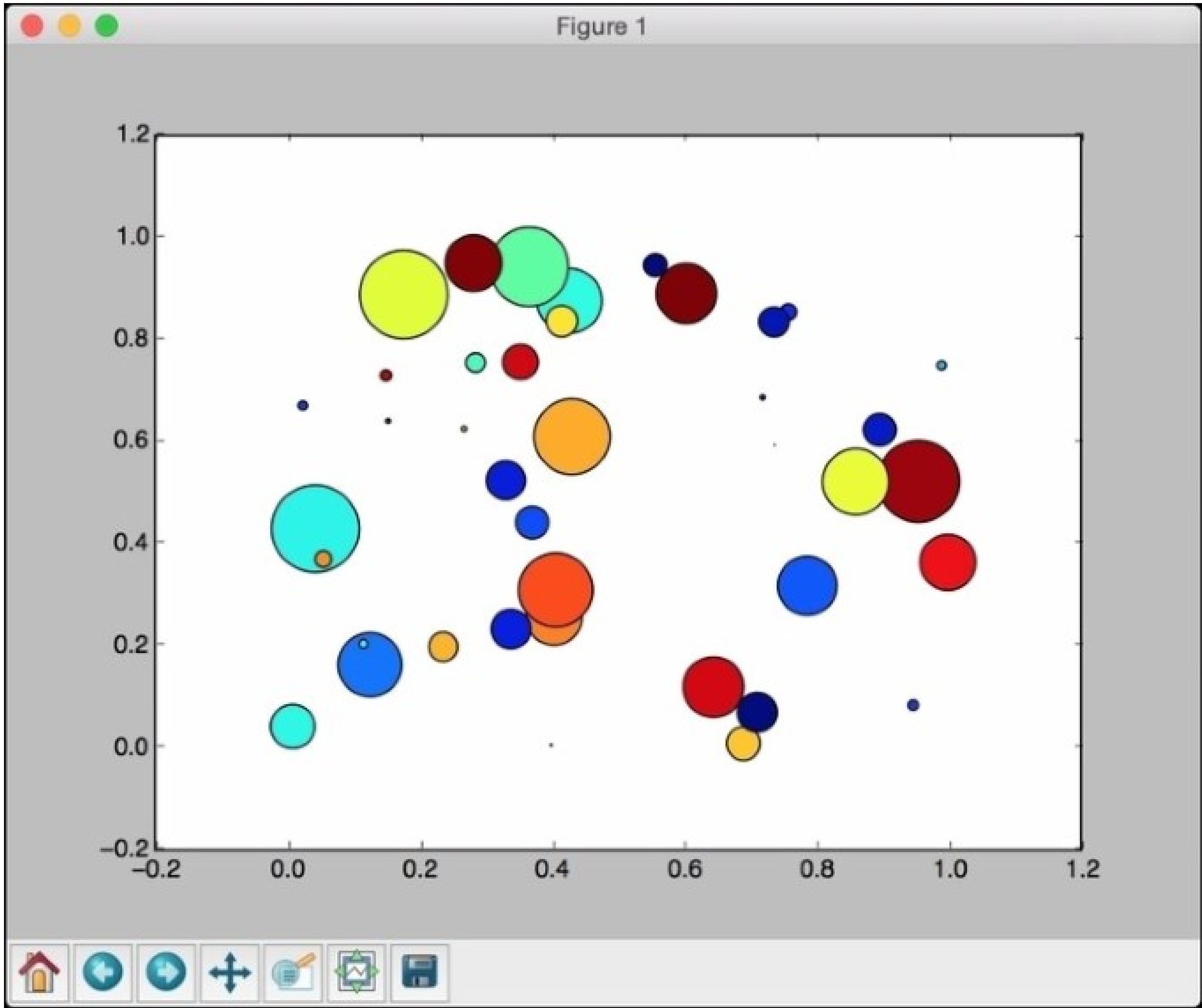


Figure 1
1.2
1.0
0.8
0.6
0.4
0.2
0.0
−0.2
−0.2
0.0
0.2
0.4
0.6
0.8
1.0
1.2

Animating bubble plots

Let's look at how to animate a bubble plot. This is useful when you want to visualize data that's transient and dynamic.

How to do it…

1. Create a new Python file, and import the following packages:

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.animation import FuncAnimation
```

2. Let's define a `tracker` function that will dynamically update the bubble plot:

```
def tracker(cur_num):
    # Get the current index
    cur_index = cur_num % num_points
```

3. Define the color:

```
    # Set the color of the datapoints
    datapoints['color'][:, 3] = 1.0
```

4. Update the size of the circles:

```
    # Update the size of the circles
    datapoints['size'] += datapoints['growth']
```

5. Update the position of the oldest datapoint in the set:

```
    # Update the position of the oldest datapoint
    datapoints['position'][cur_index] = np.random.uniform(0, 1,
2)
    datapoints['size'][cur_index] = 7
    datapoints['color'][cur_index] = (0, 0, 0, 1)
    datapoints['growth'][cur_index] = np.random.uniform(40, 150)
```

6. Update the parameters of the scatterplot:

```
    # Update the parameters of the scatter plot
    scatter_plot.set_edgecolors(datapoints['color'])
    scatter_plot.set_sizes(datapoints['size'])
    scatter_plot.set_offsets(datapoints['position'])
```

7. Define the `main` function and create an empty figure:

```
if __name__=='__main__':
    # Create a figure
    fig = plt.figure(figsize=(9, 7), facecolor=(0,0.9,0.9))
```

```
ax = fig.add_axes([0, 0, 1, 1], frameon=False)
ax.set_xlim(0, 1), ax.set_xticks([])
ax.set_ylim(0, 1), ax.set_yticks([])
```

8. Define the number of points that will be on the plot at any given point of time:

```
    # Create and initialize the datapoints in random positions
    # and with random growth rates.
    num_points = 20
```

9. Define the datapoints using random values:

```
    datapoints = np.zeros(num_points, dtype=[('position',
float, 2),
            ('size', float, 1), ('growth', float, 1), ('color',
float, 4)])
    datapoints['position'] = np.random.uniform(0, 1,
(num_points, 2))
    datapoints['growth'] = np.random.uniform(40, 150,
num_points)
```

10. Create the scatterplot that will be updated every frame:

```
    # Construct the scatter plot that will be updated every
frame
    scatter_plot = ax.scatter(datapoints['position'][:, 0],
datapoints['position'][:, 1],
                  s=datapoints['size'], lw=0.7,
edgecolors=datapoints['color'],
                  facecolors='none')
```

11. Start the animation using the `tracker` function:

```
    # Start the animation using the 'tracker' function
    animation = FuncAnimation(fig, tracker, interval=10)

    plt.show()
```

12. The full code is in the `dynamic_bubble_plot.py` file that's already provided to you. If you run this code, you will see the following figure:

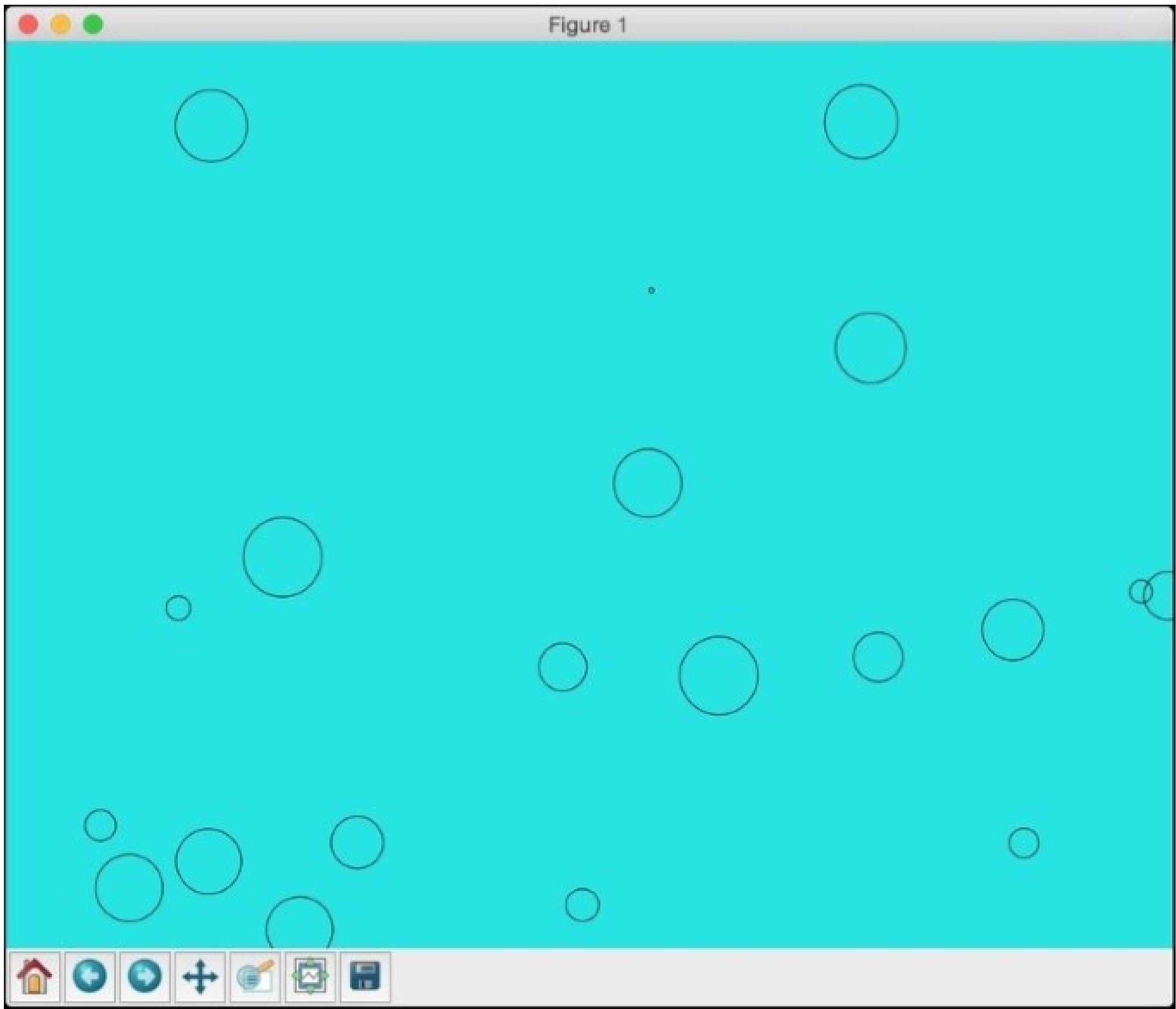


Figure 1

Drawing pie charts

Let's see how to plot pie charts. This is useful when you want to visualize the percentages of a set of labels in a group.

How to do it…

1. Create a new Python file, and import the following packages:

```
import numpy as np
import matplotlib.pyplot as plt
```

2. Define the labels and values:

```
# Labels and corresponding values in counter clockwise direction
data = {'Apple': 26,
        'Mango': 17,
        'Pineapple': 21,
        'Banana': 29,
        'Strawberry': 11}
```

3. Define the colors for visualization:

```
# List of corresponding colors
colors = ['orange', 'lightgreen', 'lightblue', 'gold', 'cyan']
```

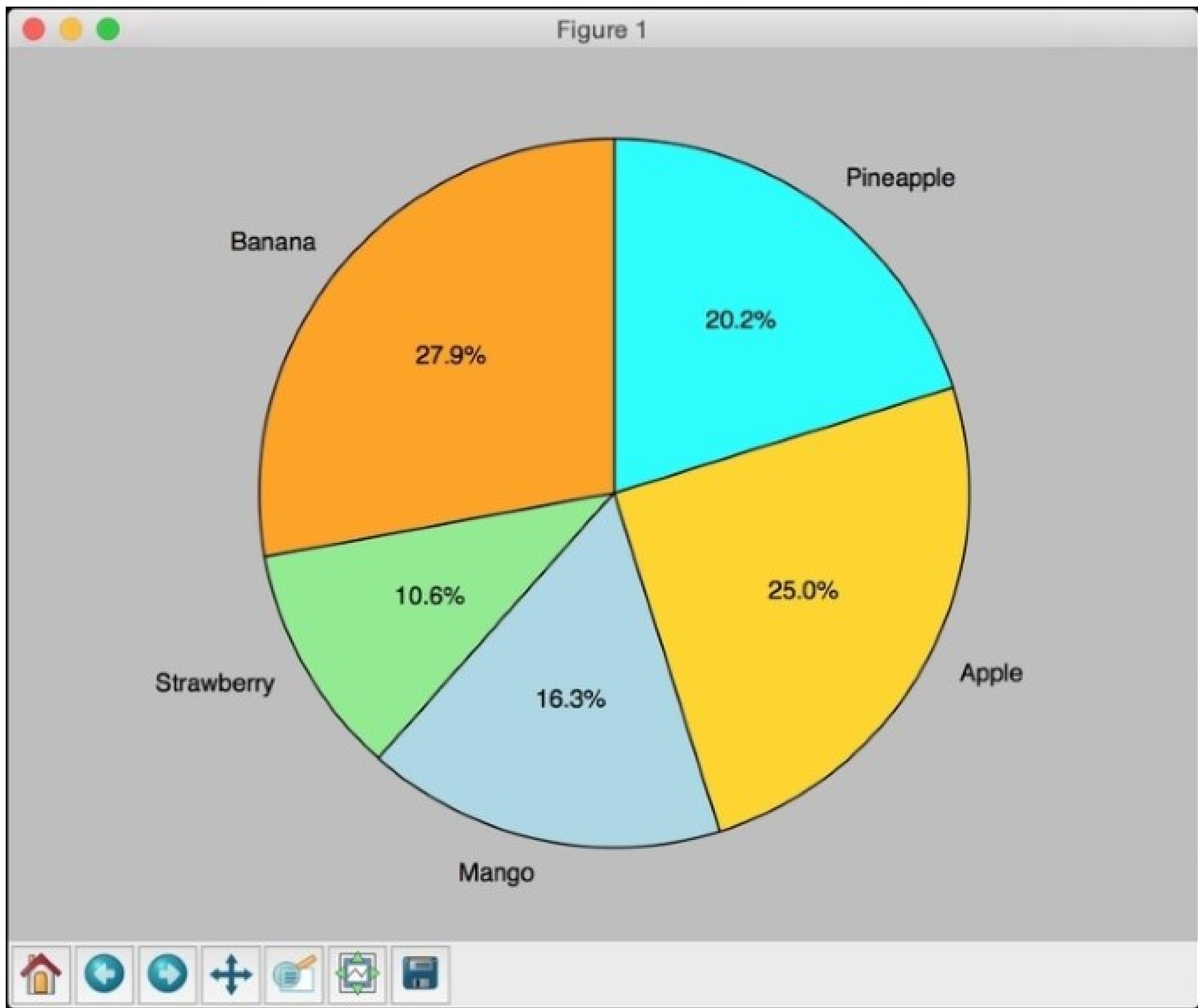
4. Define a variable to highlight a section of the pie chart by separating it from the rest. If you don't want to highlight any section, set all the values to `0`:

```
# Needed if we want to highlight a section
explode = (0, 0, 0, 0, 0)
```

5. Plot the pie chart. Note that if you use Python 3, you should use `list(data.values())` in the following function call:

```
# Plot the pie chart
plt.pie(data.values(), explode=explode, labels=data.keys(),
        colors=colors, autopct='%1.1f%%', shadow=False,
startangle=90)

# Aspect ratio of the pie chart, 'equal' indicates tht we
# want it to be a circle
plt.axis('equal')

plt.show()
```

6. The full code is in the `pie_chart.py` file that's already provided to you. If you run this code, you will see the following figure:



If you change the explode array to (`0, 0.2, 0, 0, 0`), then it will highlight the **Strawberry** section. You will see the following figure:

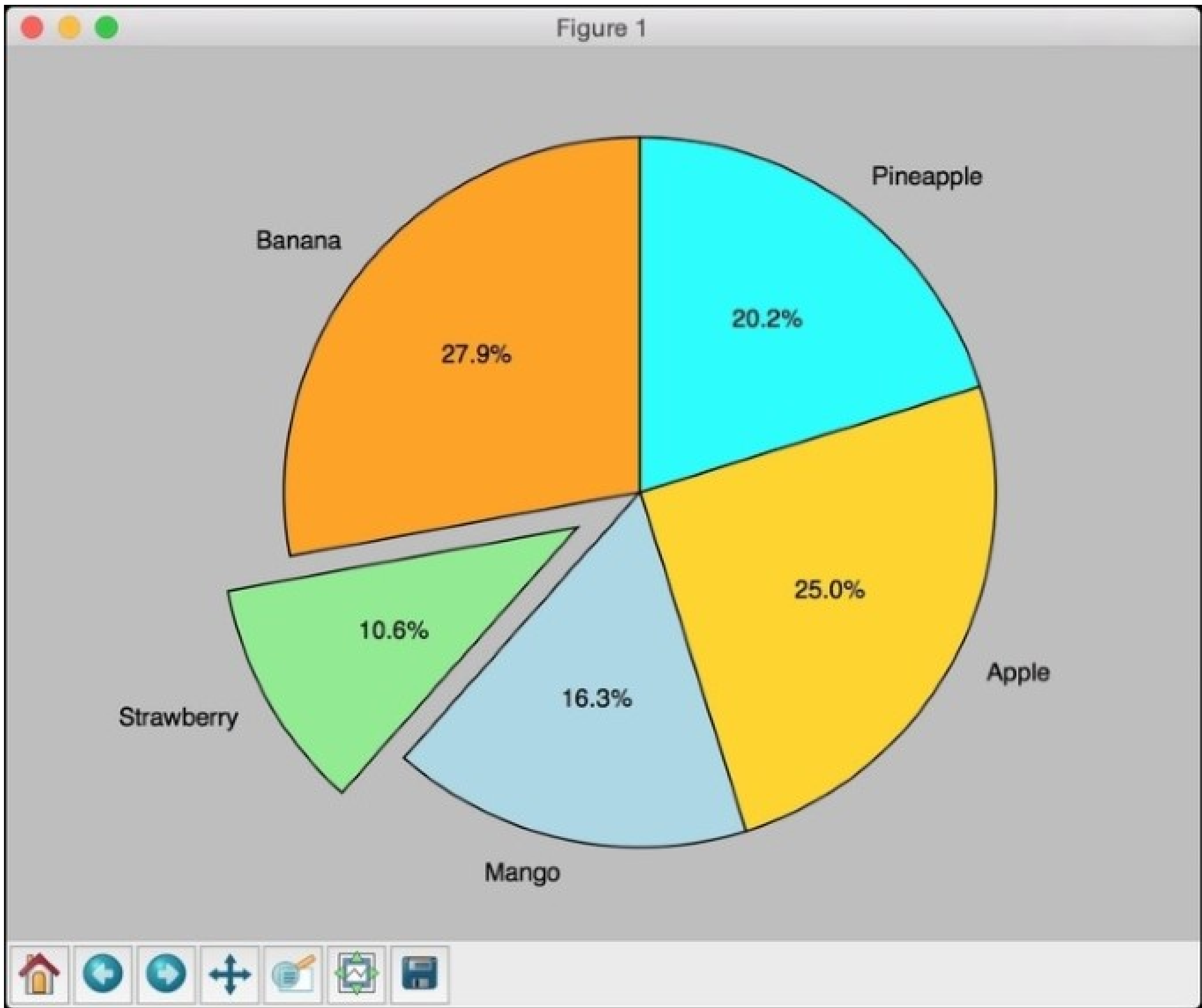


Figure 1
Pineapple
Banana
20.2%
27.9%
25.0%
10.6%
Apple
16.3%
Strawberry
Mango

Plotting date-formatted time series data

Let's look at how to plot time series data using date formatting. This is useful in visualizing stock data over time.

How to do it…

1. Create a new Python file, and import the following packages:

```
import numpy
import matplotlib.pyplot as plt
from matplotlib.mlab import csv2rec
import matplotlib.cbook as cbook
from matplotlib.ticker import Formatter
```

2. Define a function to format the dates. The `__init__` function sets the class variables:

```
# Define a class for formatting
class DataFormatter(Formatter):
    def __init__(self, dates, date_format='%Y-%m-%d'):
        self.dates = dates
        self.date_format = date_format
```

3. Extract the value at any given time and return it in the following format:

```
    # Extact the value at time t at position 'position'
    def __call__(self, t, position=0):
        index = int(round(t))
        if index >= len(self.dates) or index < 0:
            return ''

        return self.dates[index].strftime(self.date_format)
```

4. Define the `main` function. We'll use the Apple stock quotes CSV file that is available in matplotlib:

```
if __name__=='__main__':
    # CSV file containing the stock quotes
    input_file = cbook.get_sample_data('aapl.csv', 
asfileobj=False)
```

5. Load the CSV file:

```
    # Load csv file into numpy record array
    data = csv2rec(input_file)
```

6. Extract a subset of these values to plot them:

```
# Take a subset for plotting
data = data[-70:]
```

7. Create the formatter object and initialize it with the dates:

```
# Create the date formatter object
formatter = DataFormatter(data.date)
```

8. Define the X and Y axes:

```
# X axis
x_vals = numpy.arange(len(data))

# Y axis values are the closing stock quotes
y_vals = data.close
```

9. Plot the data:

```
# Plot data
fig, ax = plt.subplots()
ax.xaxis.set_major_formatter(formatter)
ax.plot(x_vals, y_vals, 'o-')
fig.autofmt_xdate()
plt.show()
```

10. The full code is in the `time_series.py` file that's already provided to you. If you run this code, you will see the following figure:

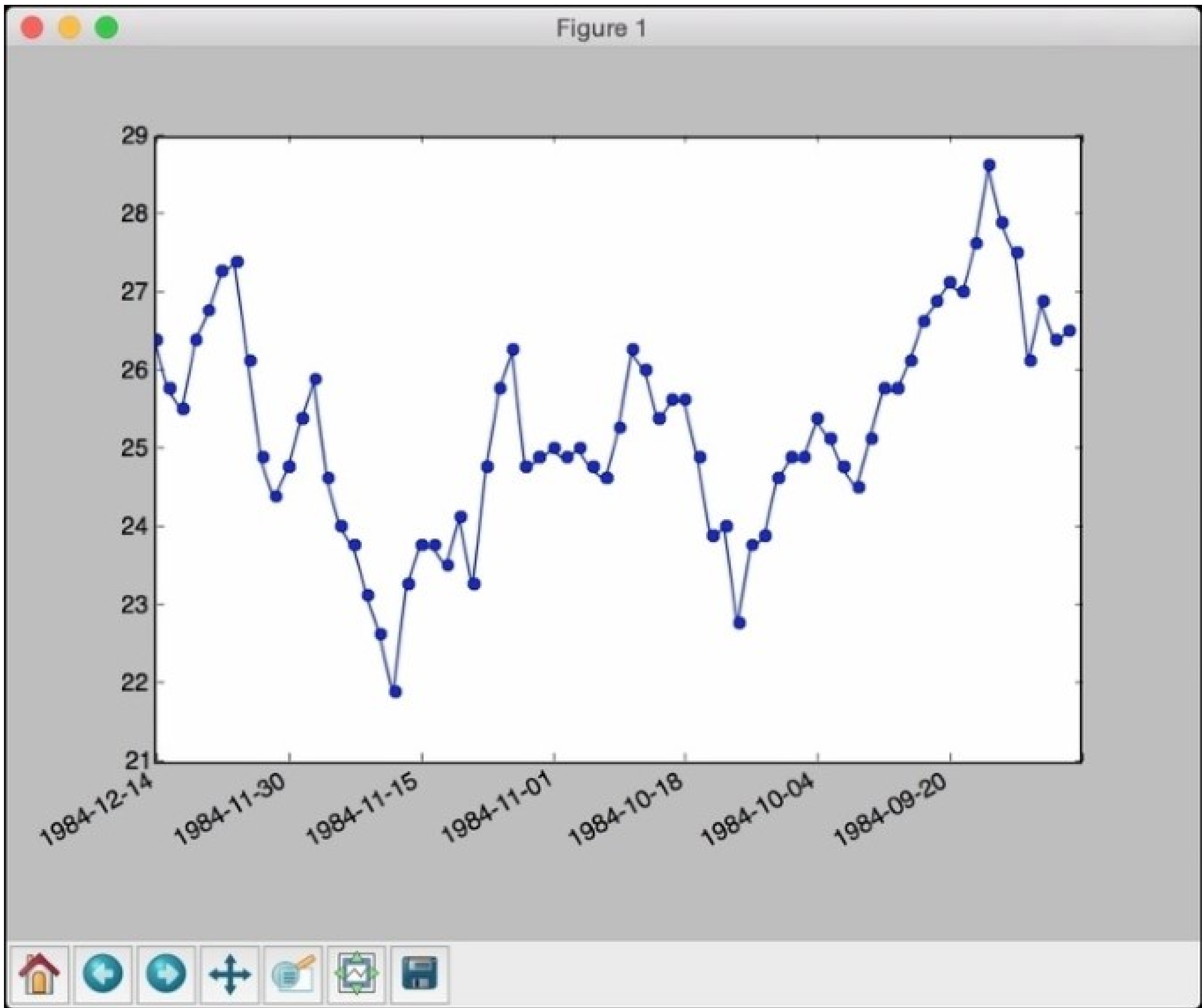


Figure 1
29
28
27
26
25
24
23
22
21
1984-12-14
1984-11-30
1984-11-15
1984-11-01
1984-10-18
1984-10-04
1984-09-20

Plotting histograms

Let's see how to plot histograms in this recipe. We'll compare two sets of data and build a comparative histogram.

How to do it…

1. Create a new Python file, and import the following packages:

```
import numpy as np
import matplotlib.pyplot as plt
```

2. We'll compare the production quantity of apples and oranges in this recipe. Let's define some values:

```
# Input data
apples = [30, 25, 22, 36, 21, 29]
oranges = [24, 33, 19, 27, 35, 20]

# Number of groups
num_groups = len(apples)
```

3. Create the figure and define its parameters:

```
# Create the figure
fig, ax = plt.subplots()

# Define the X axis
indices = np.arange(num_groups)

# Width and opacity of histogram bars
bar_width = 0.4
opacity = 0.6
```

4. Plot the histogram:

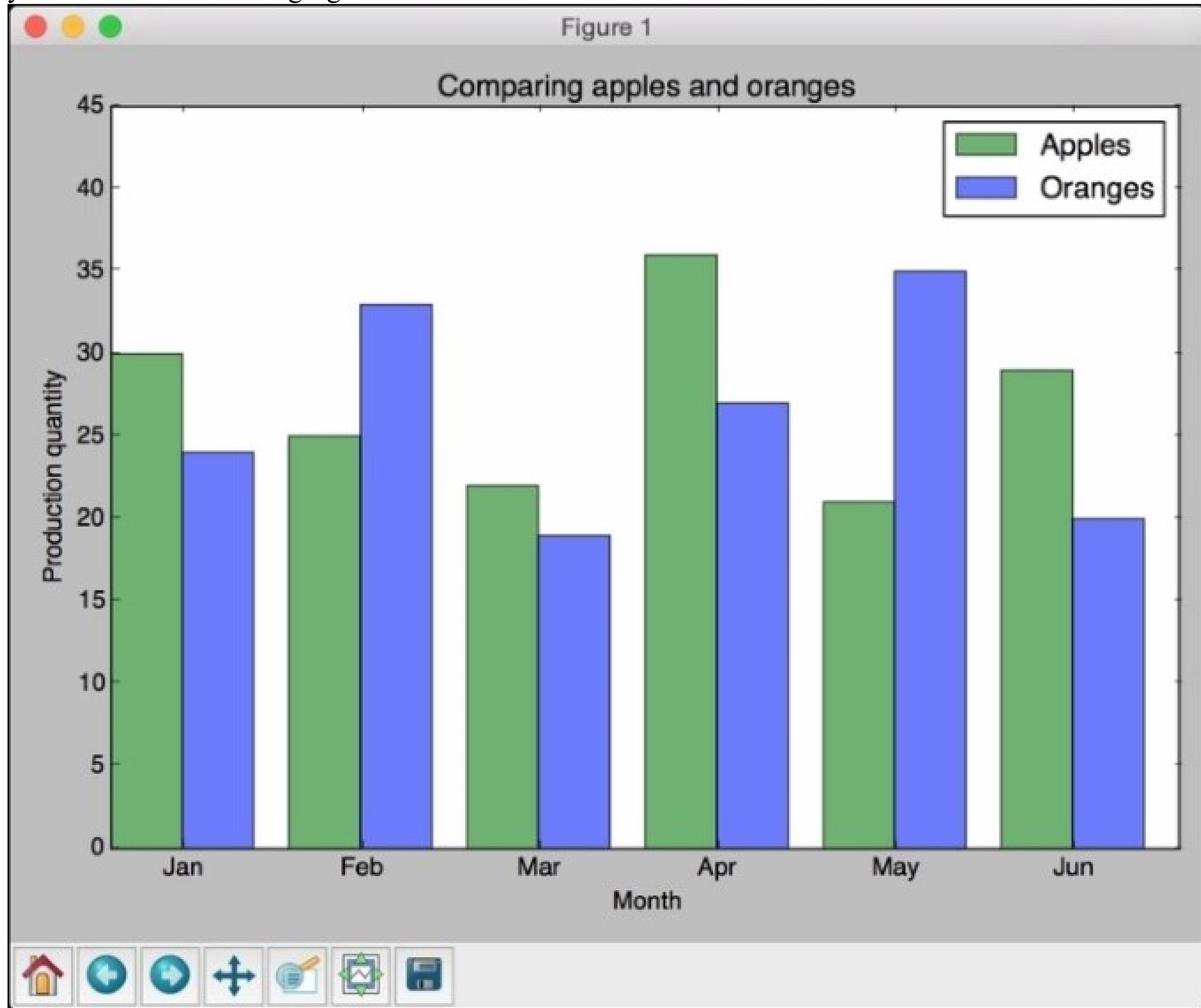
```
# Plot the values
hist_apples = plt.bar(indices, apples, bar_width,
        alpha=opacity, color='g', label='Apples')

hist_oranges = plt.bar(indices + bar_width, oranges, bar_width,
        alpha=opacity, color='b', label='Oranges')
```

5. Set the parameters of the plot:

```
plt.xlabel('Month')
plt.ylabel('Production quantity')
plt.title('Comparing apples and oranges')
plt.xticks(indices + bar_width, ('Jan', 'Feb', 'Mar', 'Apr',
```

```
'May', 'Jun'))
plt.ylim([0, 45])
plt.legend()
plt.tight_layout()

plt.show()
```

6. The full code is in the `histogram.py` file that's already provided to you. If you run this code, you will see the following figure:



Visualizing heat maps

Let's look at how to visualize heat maps in this recipe. This is a pictorial representation of data where two groups are associated point by point. The individual values that are contained in a matrix are represented as color values in the plot.

How to do it…

1. Create a new Python file, and import the following packages:

```
import numpy as np
import matplotlib.pyplot as plt
```

2. Define the two groups:

```
# Define the two groups
group1 = ['France', 'Italy', 'Spain', 'Portugal', 'Germany']
group2 = ['Japan', 'China', 'Brazil', 'Russia', 'Australia']
```

3. Generate a random 2D matrix:

```
# Generate some random values
data = np.random.rand(5, 5)
```

4. Create a figure:

```
# Create a figure
fig, ax = plt.subplots()
```

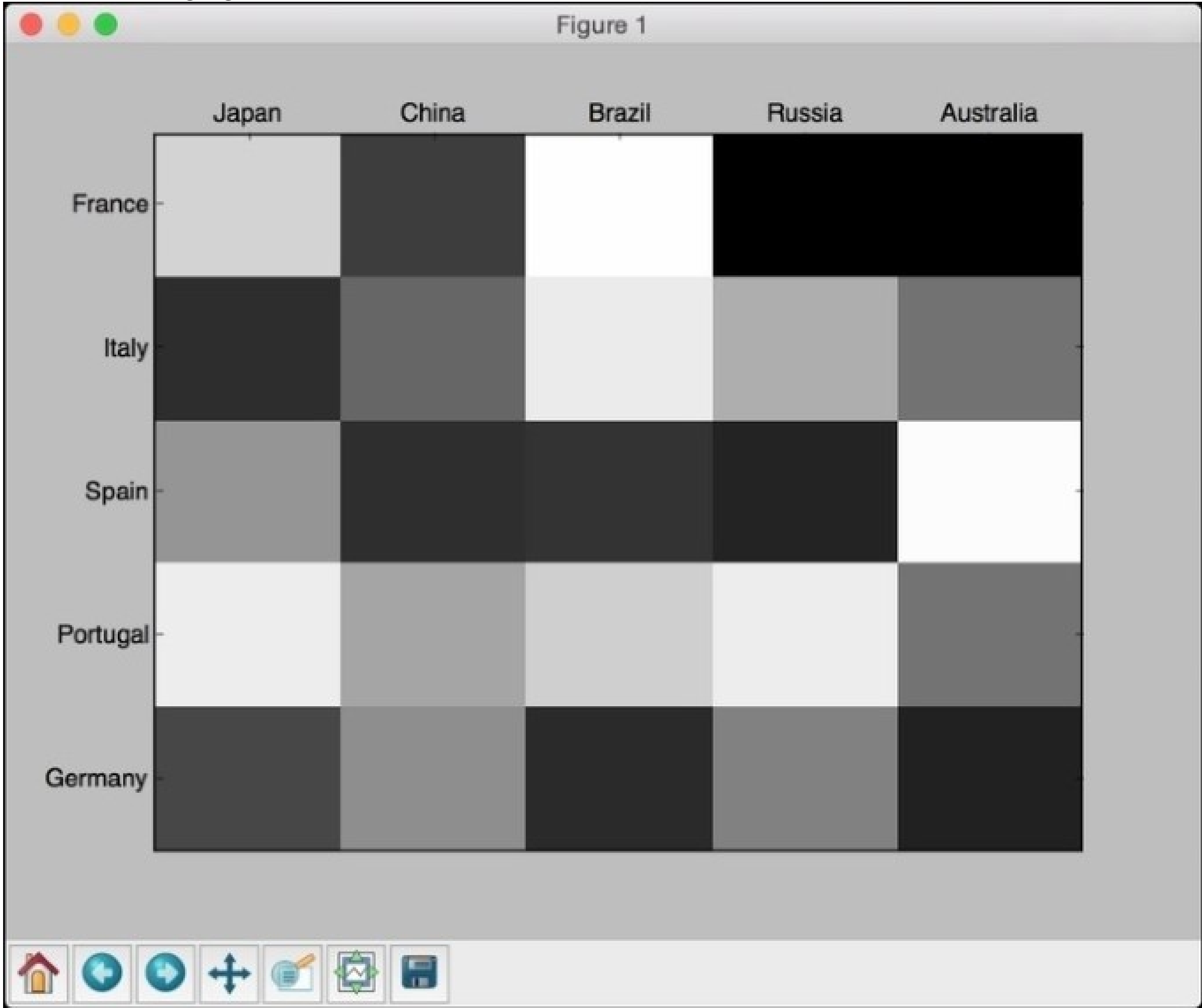
5. Create the heat map:

```
# Create the heat map
heatmap = ax.pcolor(data, cmap=plt.cm.gray)
```

6. Plot these values:

```
# Add major ticks at the middle of each cell
ax.set_xticks(np.arange(data.shape[0]) + 0.5, minor=False)
ax.set_yticks(np.arange(data.shape[1]) + 0.5, minor=False)

# Make it look like a table
ax.invert_yaxis()
ax.xaxis.tick_top()

# Add tick labels
ax.set_xticklabels(group2, minor=False)
ax.set_yticklabels(group1, minor=False)

plt.show()
```

7. The full code is in the `heatmap.py` file that's provided to you. If you run this code, you will see the following figure:



Animating dynamic signals

When we visualize real-time signals, it's nice to look at how the waveform builds up. In this recipe, we will see how to animate dynamic signals and visualize them as they are encountered in real time.

How to do it…

1. Create a new Python file, and import the following packages:

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation
```

2. Create a function to generate a damping sinusoid signal:

```
# Generate the signal
def generate_data(length=2500, t=0, step_size=0.05):
    for count in range(length):
        t += step_size
        signal = np.sin(2*np.pi*t)
        damper = np.exp(-t/8.0)
        yield t, signal * damper
```

3. Define an `initializer` function to initialize parameters of the plot:

```
# Initializer function
def initializer():
    peak_val = 1.0
    buffer_val = 0.1
```

4. Set these parameters:

```
    ax.set_ylim(-peak_val * (1 + buffer_val), peak_val * (1 +
buffer_val))
    ax.set_xlim(0, 10)
    del x_vals[:]
    del y_vals[:]
    line.set_data(x_vals, y_vals)
    return line
```

5. Define a function to draw the values:

```
def draw(data):
    # update the data
    t, signal = data
    x_vals.append(t)
    y_vals.append(signal)
    x_min, x_max = ax.get_xlim()
```

6. If the values go past the current X axis limits, then update and extend the graph:

```
    if t >= x_max:
        ax.set_xlim(x_min, 2 * x_max)
        ax.figure.canvas.draw()

    line.set_data(x_vals, y_vals)

    return line
```

7. Define the `main` function:

```
if __name__=='__main__':
    # Create the figure
    fig, ax = plt.subplots()
    ax.grid()
```

8. Extract the line:

```
    # Extract the line
    line, = ax.plot([], [], lw=1.5)
```

9. Create variables and initialize them to empty lists:

```
    # Create the variables
    x_vals, y_vals = [], []
```

10. Define and start the animation using the animator object:

```
    # Define the animator object
    animator = animation.FuncAnimation(fig, draw,
generate_data,
            blit=False, interval=10, repeat=False,
init_func=initializer)

    plt.show()
```

11. The full code is in the `moving_wave_variable.py` file that's already provided to you. If you run this code, you will see the following figure:

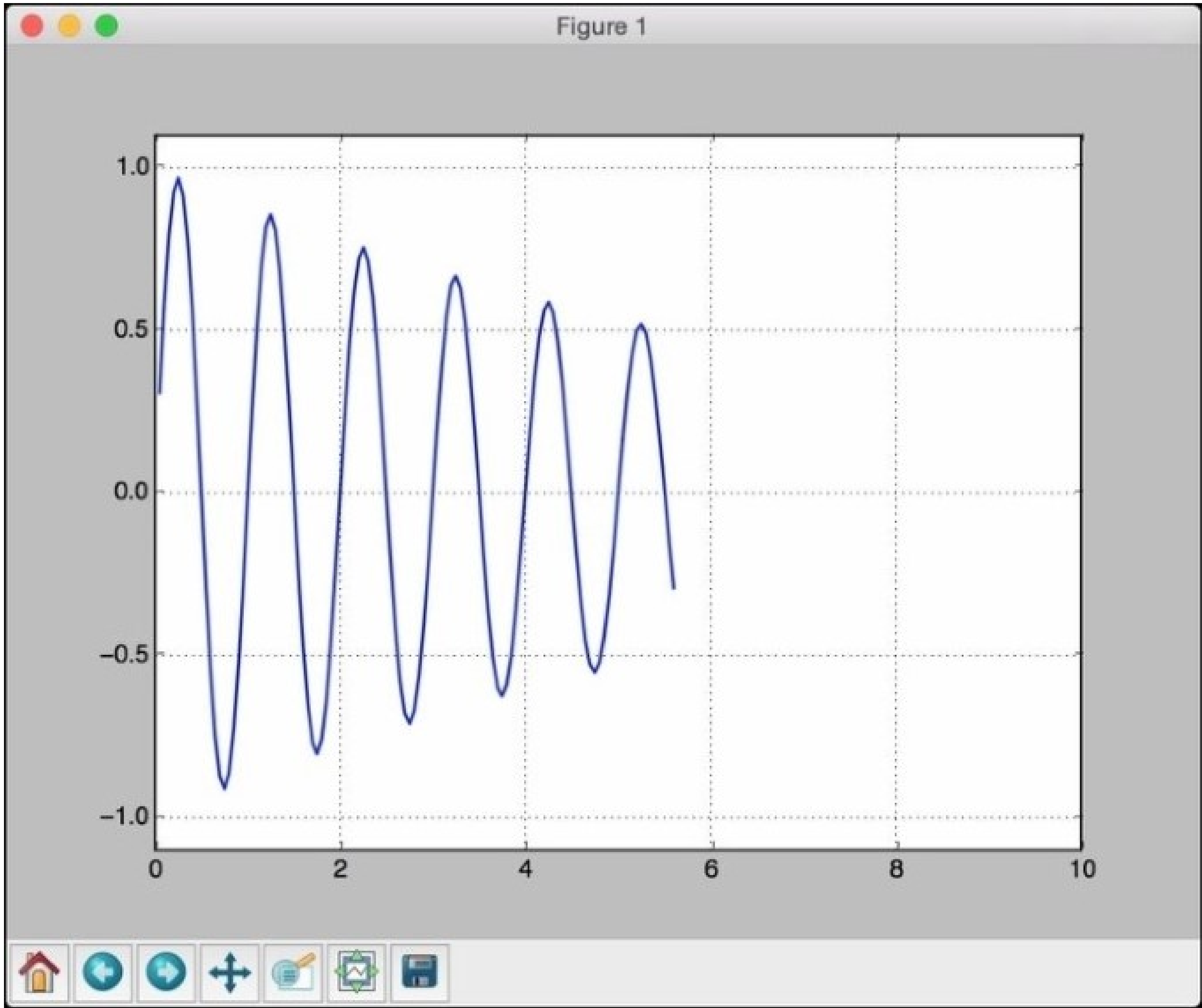


Figure 1
1.0
0.5
0.0
−0.5
−1.0
0
2
4
6
8
10

Part II. Module 2

Advanced Machine Learning with Python

Solve challenging data science problems by mastering cutting-edge machine learning techniques in Python

Chapter 1. Unsupervised Machine Learning

In this chapter, you will learn how to apply unsupervised learning techniques to identify patterns and structure within datasets.

Unsupervised learning techniques are a valuable set of tools for exploratory analysis. They bring out patterns and structure within datasets, which yield information that may be informative in itself or serve as a guide to further analysis. It's critical to have a solid set of unsupervised learning tools that you can apply to help break up unfamiliar or complex datasets into actionable information.

We'll begin by reviewing **Principal Component Analysis** (**PCA**), a fundamental data manipulation technique with a range of dimensionality reduction applications. Next, we will discuss **k-means clustering**, a widely-used and approachable unsupervised learning technique. Then, we will discuss Kohenen's **Self-Organizing Map** (**SOM**), a method of topological clustering that enables the projection of complex datasets into two dimensions.

Throughout the chapter, we will spend some time discussing how to effectively apply these techniques to make high-dimensional datasets readily accessible. We will use the **UCI Handwritten Digits** dataset to demonstrate technical applications of each algorithm. In the course of discussing and applying each technique, we will review practical applications and methodological questions, particularly regarding how to calibrate and validate each technique as well as which performance measures are valid. To recap, then, we will be covering the following topics in order:

- Principal component analysis
- k-means clustering
- Self-organizing maps

Principal component analysis

In order to work effectively with high-dimensional datasets, it is important to have a set of techniques that can reduce this dimensionality down to manageable levels. The advantages of this dimensionality reduction include the ability to plot multivariate data in two dimensions, capture the majority of a dataset's informational content within a minimal number of features, and, in some contexts, identify collinear model components.

Note

For those in need of a refresher, collinearity in a machine learning context refers to model features that share an approximately linear relationship. For reasons that will likely be obvious, these features tend to be unhelpful as the related features are unlikely to add information mutually that either one provides independently. Moreover, collinear features may emphasize local minima or other false leads.

Probably the most widely-used dimensionality reduction technique today is PCA. As we'll be applying PCA in multiple contexts throughout this book, it's appropriate for us to review the technique, understand the theory behind it, and write Python code to effectively apply it.

PCA – a primer

PCA is a powerful decomposition technique; it allows one to break down a highly multivariate dataset into a set of orthogonal components. When taken together in sufficient number, these components can explain almost all of the dataset's variance. In essence, these components deliver an abbreviated description of the dataset. PCA has a broad set of applications and its extensive utility makes it well worth our time to cover.

Note

Note the slightly cautious phrasing here—a given set of components of length less than the number of variables in the original dataset will almost always lose some amount of the information content within the source dataset. This lossiness is typically minimal, given enough components, but in cases where small numbers of principal components are composed from very high-dimensional datasets, there may be substantial lossiness. As such, when performing PCA, it is always appropriate to consider how many components will be necessary to effectively model the dataset in question.

PCA works by successively identifying the axis of greatest variance in a dataset (the principal components). It does this as follows:

1. Identifying the center point of the dataset.
2. Calculating the covariance matrix of the data.
3. Calculating the eigenvectors of the covariance matrix.
4. Orthonormalizing the eigenvectors.
5. Calculating the proportion of variance represented by each eigenvector.

Let's unpack these concepts briefly:

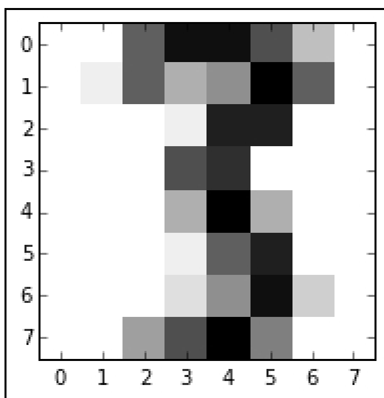
- **Covariance** is effectively variance applied to multiple dimensions; it is the variance between two or more variables. While a single value can capture the variance in one dimension or variable, it is necessary to use a *2 x 2* matrix to capture the covariance between two variables, a *3 x 3* matrix to capture the covariance between three variables, and so on. So the first step in PCA is to calculate this covariance matrix.
- An **Eigenvector** is a vector that is specific to a dataset and linear transformation. Specifically, it is the vector that does not change in direction before and after the transformation is performed. To get a better feeling for how this works, imagine that you're holding a rubber band, straight, between both hands. Let's say you stretch the band out until it is taut between your hands. The eigenvector is the vector that did not change direction between before the stretch and during it; in this case, it's the vector running directly through the center of the band from one hand to the other.
- **Orthogonalization** is the process of finding two vectors that are orthogonal (at right angles) to one another. In an n-dimensional data space, the process of orthogonalization takes a set of vectors and yields a set of orthogonal vectors.
- **Orthonormalization** is an orthogonalization process that also normalizes the product.
- **Eigenvalue** (roughly corresponding to the length of the eigenvector) is used to calculate the proportion of variance represented by each eigenvector. This is done by dividing the eigenvalue for each eigenvector by the sum of eigenvalues for all eigenvectors.

In summary, the covariance matrix is used to calculate Eigenvectors. An orthonormalization process is undertaken that produces orthogonal, normalized vectors from the Eigenvectors. The eigenvector with the greatest eigenvalue is the first principal component with successive components having smaller eigenvalues. In this way, the PCA algorithm has the effect of taking a dataset and transforming it into a new, lower-dimensional coordinate system.

Employing PCA

Now that we've reviewed the PCA algorithm at a high level, we're going to jump straight in and apply PCA to a key Python dataset—the UCI handwritten `digits` dataset, distributed as part of **scikit-learn**.

This dataset is composed of *1,797* instances of handwritten digits gathered from 44 different writers. The input (pressure and location) from these authors' writing is resampled twice across an *8 x 8* grid so as to yield maps of the kind shown in the following image:



These maps can be transformed into feature vectors of length 64, which are then readily usable as analysis input. With an input dataset of 64 features, there is an immediate appeal to using a technique like PCA to reduce the set of variables to a manageable amount. As it currently stands, we cannot effectively explore the dataset with exploratory visualization!

We will begin applying PCA to the handwritten `digits` dataset with the following code:

```
import numpy as np
from sklearn.datasets import load_digits
import matplotlib.pyplot as plt
from sklearn.decomposition import PCA
from sklearn.preprocessing import scale
from sklearn.lda import LDA
import matplotlib.cm as cm

digits = load_digits()
data = digits.data
```

```
n_samples, n_features = data.shape
n_digits = len(np.unique(digits.target))
labels = digits.target
```

This code does several things for us:

1. First, it loads up a set of necessary libraries, including `numpy`, a set of components from scikit-learn, including the `digits` dataset itself, PCA and data scaling functions, and the plotting capability of matplotlib.
2. The code then begins preparing the `digits` dataset. It does several things in order:
    - First, it loads the dataset before creating helpful variables
    - The `data` variable is created for subsequent use, and the number of distinct `digits` in the `target` vector (*0* through to *9*, so `n_digits = 10`) is saved as a variable that we can easily access for subsequent analysis
    - The `target` vector is also saved as labels for later use
    - All of this variable creation is intended to simplify subsequent analysis
3. With the dataset ready, we can initialize our PCA algorithm and apply it to the dataset:

    ```
    pca = PCA(n_components=10)
    data_r = pca.fit(data).transform(data)

    print('explained variance ratio (first two components): %s' %
    str(pca.explained_variance_ratio_))
    print('sum of explained variance (first two components): %s' %
    str(sum(pca.explained_variance_ratio_)))
    ```

4. This code outputs the variance explained by each of the first ten principal components ordered by explanatory power.

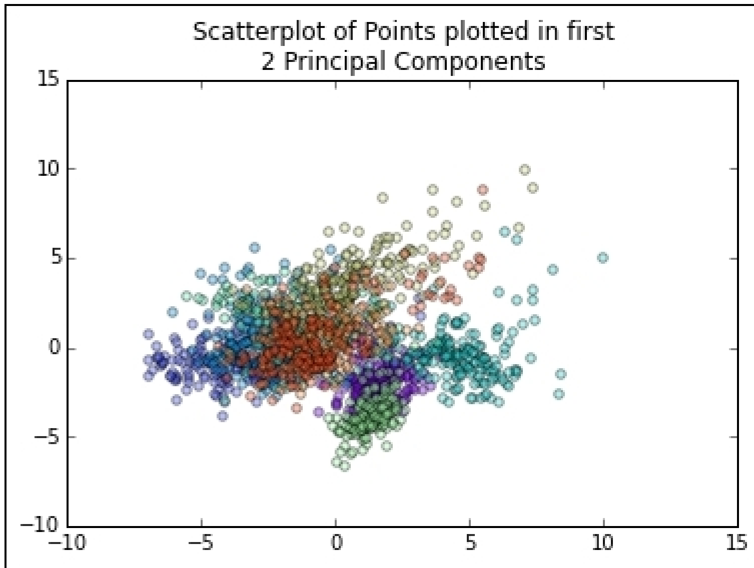
In the case of this set of `10` principal components, they collectively explain *0.589* of the overall dataset variance. This isn't actually too bad, considering that it's a reduction from *64* variables to `10` components. It does, however, illustrate the potential lossiness of PCA. The key question, though, is whether this reduced set of components makes subsequent analysis or classification easier to achieve; that is, whether many of the remaining components contained variance that disrupts classification attempts.

Having created a `data_r` object containing the output of `pca` performed over the `digits` dataset, let's visualize the output. To do so, we'll first create a vector of `colors` for class coloration. We then simply create a scatterplot with colorized classes:

```
X = np.arange(10)
ys = [i+x+(i*x)**2 for i in range(10)]

plt.figure()
colors = cm.rainbow(np.linspace(0, 1, len(ys)))
for c, i target_name in zip(colors, [1,2,3,4,5,6,7,8,9,10], labels):
   plt.scatter(data_r[labels == I, 0], data_r[labels == I, 1],
```

```
c=c, alpha = 0.4)
plt.legend()
plt.title('Scatterplot of Points plotted in first \n'
'10 Principal Components')
plt.show()
```

The resulting scatterplot looks as follows:



This plot shows us that, while there is some separation between classes in the first two principal components, it may be tricky to classify highly accurately with this dataset. However, classes do appear to be clustered and we may be able to get reasonably good results by employing a clustering analysis. In this way, PCA has given us some insight into how the dataset is structured and has informed our subsequent analysis.

At this point, let's take this insight and move on to examine clustering by the application of the k-means clustering algorithm.

Introducing k-means clustering

In the previous section, you learned that unsupervised machine learning algorithms are used to extract key structural or information content from large, possibly complex datasets. These algorithms do so with little or no manual input and function without the need for training data (sets of labeled explanatory and response variables needed to train an algorithm in order to recognize the desired classification boundaries). This means that unsupervised algorithms are effective tools to generate information about the structure and content of new or unfamiliar datasets. They allow the analyst to build a strong understanding in a fraction of the time.

Clustering – a primer

Clustering is probably the archetypal unsupervised learning technique for several reasons.

A lot of development time has been sunk into optimizing clustering algorithms, with efficient implementations available in most data science languages including Python.

Clustering algorithms tend to be very fast, with smoothed implementations running in polynomial time. This makes it uncomplicated to run multiple clustering configurations, even over large datasets. Scalable clustering implementations also exist that parallelize the algorithm to run over **TB-scale** datasets.

Clustering algorithms are frequently easily understood and their operation is thus easy to explain if necessary.

The most popular clustering algorithm is k-means; this algorithm forms k-many clusters by first randomly initiating the clusters as k-many points in the data space. Each of these points is the mean of a cluster. An iterative process then occurs, running as follows:

- Each point is assigned to a cluster based on the least (within cluster) sum of squares, which is intuitively the nearest mean.
- The center (centroid) of each cluster becomes the new mean. This causes each of the means to shift.

Over enough iterations, the centroids move into positions that minimize a performance metric (the performance metric most commonly used is the "within cluster least sum of squares" measure). Once this measure is minimized, observations are no longer reassigned during iteration; at this point the algorithm has converged on a solution.

Kick-starting clustering analysis

Now that we've reviewed the clustering algorithm, let's run through the code and see what clustering can do for us:

```
from time import time
import numpy as np
import matplotlib.pyplot as plt
```

```
np.random.seed()



digits = load_digits()
data = scale(digits.data)

n_samples, n_features = data.shape
n_digits = len(np.unique(digits.target))
labels = digits.target

sample_size = 300

print("n_digits: %d, \t n_samples %d, \t n_features %d"
   % (n_digits, n_samples, n_features))


print(79 * '_')
print('% 9s' % 'init''         time   inertia   homo   compl
v-meas   ARI     AMI  silhouette')

def bench_k_means(estimator, name, data):
   t0 = time()
   estimator.fit(data)
   print('% 9s %.2fs %i %.3f %.3f %.3f %.3f %.3f %.3f'
      % (name, (time() - t0), estimator.inertia_,
         metrics.homogeneity_score(labels, estimator.labels_),
         metrics.completeness_score(labels, estimator.labels_),
         metrics.v_measure_score(labels, estimator.labels_),
         metrics.adjusted_rand_score(labels, estimator.labels_),
         metrics.silhouette_score(data, estimator.labels_,
            metric='euclidean',
            sample_size=sample_size)))
```

Note

One critical difference between this code and the PCA code we saw previously is that this code begins by applying a scale function to the `digits` dataset. This function scales values in the dataset between *0* and *1*. It's critically important to scale data wherever needed, either on a log scale or bound scale, so as to prevent the magnitude of different feature values to have disproportionately powerful effects on the dataset. The key to determining whether the data needs scaling at all (and what kind of scaling is needed, within which range, and so on) is very much tied to the shape and nature of the data. If the distribution of the data shows outliers or variation within a large range, it may be appropriate to apply log-scaling. Whether this is done manually through visualization and exploratory analysis techniques or through the use of summary statistics, decisions around scaling are tied to the data under inspection and the analysis techniques to be used. A further discussion of scaling decisions and considerations may be found in Chapter 7, *Feature Engineering Part II*.

Helpfully, scikit-learn uses the k-means++ algorithm by default, which improves over the original k-means algorithm in terms of both running time and success rate in avoiding poor clusterings.

The algorithm achieves this by running an initialization procedure to find cluster centroids that approximate minimal variance within classes.

You may have spotted from the preceding code that we're using a set of performance estimators to track how well our k-means application is performing. It isn't practical to measure the performance of a clustering algorithm based on a single correctness percentage or using the same performance measures that are commonly used with other algorithms. The definition of success for clustering algorithms is that they provide an interpretation of how input data is grouped that trades off between several factors, including class separation, in-group similarity, and cross-group difference.

The **homogeneity score** is a simple, zero-to-one-bounded measure of the degree to which clusters contain only assignments of a given class. A score of one indicates that all clusters contain measurements from a single class. This measure is complimented by the **completeness score**, which is a similarly bounded measure of the extent to which all members of a given class are assigned to the same cluster. As such, a completeness score and homogeneity score of one indicates a perfect clustering solution.

The **validity measure (v-measure)** is a harmonic mean of the homogeneity and completeness scores, which is exactly analogous to the F-measure for binary classification. In essence, it provides a single, 0-1-scaled value to monitor both homogeneity and completeness.

The **Adjusted Rand Index** (**ARI**) is a similarity measure that tracks the consensus between sets of assignments. As applied to clustering, it measures the consensus between the true, pre-existing observation labels and the labels predicted as an output of the clustering algorithm. The Rand index measures labeling similarity on a *0-1* bound scale, with one equaling perfect prediction labels.

The main challenge with all of the preceding performance measures as well as other similar measures (for example, Akaike's mutual information criterion) is that they require an understanding of the ground truth, that is, they require some or all of the data under inspection to be labeled. If labels do not exist and cannot be generated, these measures won't work. In practice, this is a pretty substantial drawback as very few datasets come prelabeled and the creation of labels can be time-consuming.

One option to measure the performance of a k-means clustering solution without labeled data is the **Silhouette Coefficient**. This is a measure of how well-defined the clusters within a model are. The Silhouette Coefficient for a given dataset is the mean of the coefficient for each sample, where this coefficient is calculated as follows:

$$s = \frac{b-a}{\max(a,b)}$$

The definitions of each term are as follows:

- *a*: The mean distance between a sample and all other points in the same cluster
- *b*: The mean distance between a sample and all other points in the next nearest cluster

This score is bounded between *-1* and *1*, with *-1* indicating incorrect clustering, *1* indicating very dense clustering, and scores around *0* indicating overlapping clusters. This tends to fit our expectations of how a good clustering solution is composed.

In the case of the `digits` dataset, we can employ all of the performance measures described here. As such, we'll complete the preceding example by initializing our `bench_k_means` function over the `digits` dataset:

```
bench_k_means(KMeans(init='k-means++', n_clusters=n_digits,
n_init=10), name="k-means++", data=data)
print(79 * '_')
```

This yields the following output (note that the random seed means your results will vary from mine!):

```
n_digits: 10,    n_samples 1797,        n_features 64
_______________________________________________________________________________
init         time   inertia    homo   compl  v-meas     ARI     AMI  silhouette
k-means++   0.25s    69517   0.596   0.643   0.619   0.465   0.592    0.123
_______________________________________________________________________________
```

Lets take a look at these results in more detail.

The Silhouette score at `0.123` is fairly low, but not surprisingly so, given that the handwritten digits data is inherently noisy and does tend to overlap. However, some of the other scores are not that impressive. The V-measure at `0.619` is reasonable, but in this case is held back by a poor homogeneity measure, suggesting that the cluster centroids did not resolve perfectly. Moreover, the ARI at `0.465` is not great.

Note

Let's put this in context. The worst case classification attempt, random assignment, would give at best 10% classification accuracy. All of our performance measures would be accordingly very low. While we're definitely doing a lot better than that, we're still trailing far behind the best computational classification attempts. As we'll see in Chapter 4, *Convolutional Neural Networks*, convolutional nets achieve results with extremely low classification errors on handwritten digit datasets. We're unlikely to achieve this level of accuracy with traditional k-means clustering!

All in all, it's reasonable to think that we could do better.

To give this another try, we'll apply an additional stage of processing. To learn how to do this, we'll apply PCA—the technique we previously walked through—to reduce the dimensionality of our input dataset. The code to achieve this is very simple, as follows:

```
pca = PCA(n_components=n_digits).fit(data)
bench_k_means(KMeans(init=pca.components_, n_clusters=10),
name="PCA-based",
data=data)
```

This code simply applies `PCA` to the `digits` dataset, yielding as many principal components as there are classes (in this case, digits). It can be sensible to review the output of `PCA` before proceeding as the presence of any small principal components may suggest a dataset that contains collinearity or otherwise merits further inspection.

This instance of clustering shows noticeable improvement:

n_digits: 10, n_samples 1797, n_features 64

init	time	inertia	homo	compl	v-meas	ARI	silhouette
PCA-based	0.02s	71820	0.673	0.715	0.693	0.567	0.121

The V-measure and ARI have increased by approximately *0.08* points, with the V-measure reading a fairly respectable `0.693`. The Silhouette Coefficient did not change significantly. Given the complexity and interclass overlap within the `digits` dataset, these are good results, particularly stemming from such a simple code addition!

Inspection of the `digits` dataset with clusters superimposed shows that some meaningful clusters appear to have been formed. It is also apparent from the following plot that actually detecting the character from the input feature vectors may be a challenging task:

Tuning your clustering configurations

The previous examples described how to apply k-means, walked through relevant code, showed how to plot the results of a clustering analysis, and identified appropriate performance metrics. However, when applying k-means to real-world datasets, there are some extra precautions that need to be taken, which we will discuss.

Another critical practical point is how to select an appropriate value for *k*. Initializing k-means clustering with a specific *k* value may not be harmful, but in many cases it is not clear initially how many clusters you might find or what values of *k* may be helpful.

We can rerun the preceding code for multiple values of *k* in a batch and look at the performance metrics, but this won't tell us which instance of *k* is most effectively capturing structure within the data. The risk is that as *k* increases, the Silhouette Coefficient or unexplained variance may decrease dramatically, without meaningful clusters being formed. The extreme case of this would be if $k = o$, where o is the number of observations in the sample; every point would have its own cluster, the Silhouette Coefficient would be low, but the results wouldn't be meaningful. There are, however, many less extreme cases in which overfitting may occur due to an overly high *k* value.

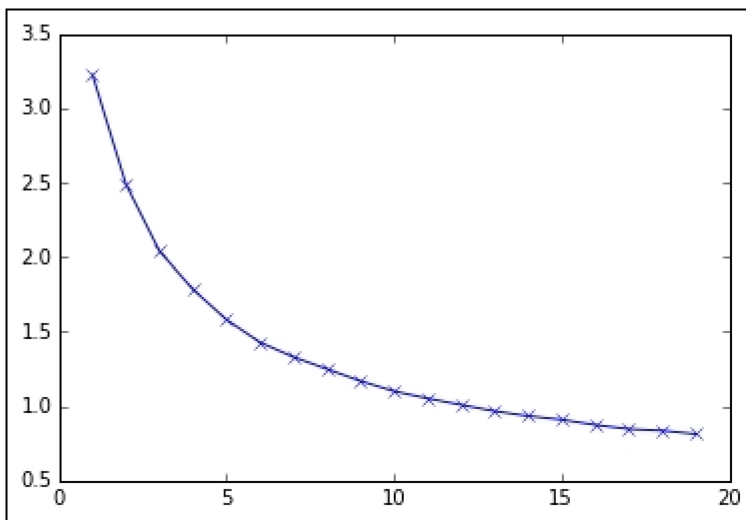
To mitigate this risk, it's advisable to use supporting techniques to motivate a selection of *k*. One useful technique in this context is the **elbow method**. The elbow method is a very simple technique; for each instance of *k*, plot the percentage of explained variance against *k*. This typically leads to a plot that frequently looks like a bent arm.

For the PCA-reduced dataset, this code looks like the following snippet:

```
import numpy as np
from sklearn.cluster import KMeans
from sklearn.datasets import load_digits
from scipy.spatial.distance import cdist
import matplotlib.pyplot as plt
from sklearn.decomposition import PCA
from sklearn.preprocessing import scale

digits = load_digits()
data = scale(digits.data)

n_samples, n_features = data.shape
n_digits = len(np.unique(digits.target))
labels = digits.target

K = range(1,20)
explainedvariance= []
for k in K:
   reduced_data = PCA(n_components=2).fit_transform(data)
   kmeans = KMeans(init = 'k-means++', n_clusters = k, n_init = k)
   kmeans.fit(reduced_data)
```

```
explainedvariance.append(sum(np.min(cdist(reduced_data,
kmeans.cluster_centers_, 'euclidean'), axis =
1))/data.shape[0])
plt.plot(K, meandistortions, 'bx-')
plt.show()
```

This application of the elbow method takes the `PCA` reduction from the previous code sample and applies a test of the explained variance (specifically, a test of the variance within clusters). The result is output as a measure of unexplained variance for each value of *k* in the range specified. In this case, as we're using the `digits` dataset (which we know to have ten classes), the `range` specified was `1` to `20`:



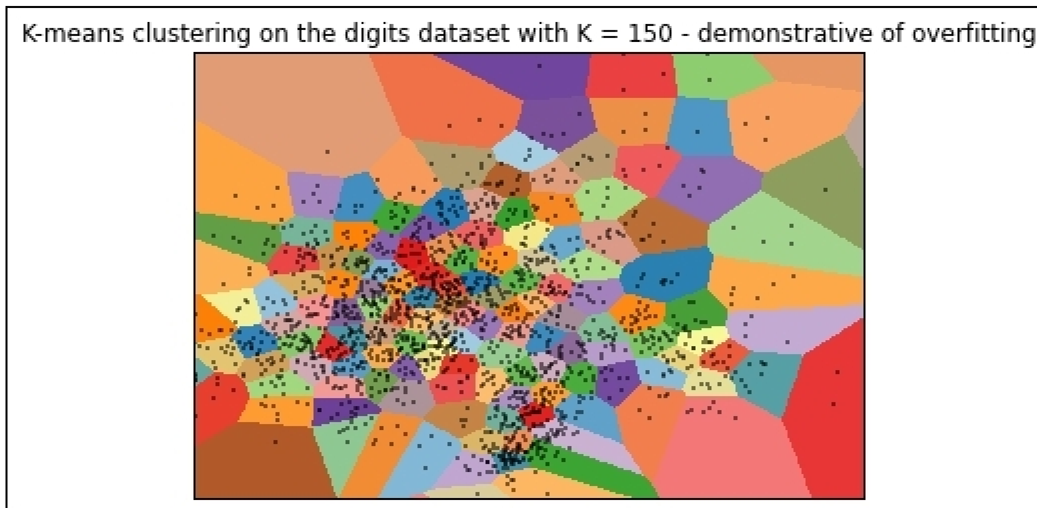
The elbow method involves selecting the value of *k* that maximizes explained variance while minimizing `K`; that is, the value of *k* at the crook of the elbow. The technical sense underlying this is that a minimal gain in explained variance at greater values of *k* is offset by the increasing risk of overfitting.

Elbow plots may be more or less pronounced and the elbow may not always be clearly identifiable. This example shows a more gradual progression than may be observable in other cases with other datasets. It's worth noting that, while we know the number of classes within the dataset to be ten, the elbow method starts to show diminishing returns on *k* increases almost immediately and the elbow is located at around five classes. This has a lot to do with the substantial overlap between classes, which we saw in previous plots. While there are ten classes, it becomes increasingly difficult to clearly identify more than five or so.

With this in mind, it's worth noting that the elbow method is intended for use as a heuristic rather than as some kind of objective principle. The use of PCA as a preprocess to improve clustering performance also tends to smooth the graph, delivering a more gradual curve than otherwise.

In addition to making use of the elbow method, it can be valuable to look at the clusters themselves, as we did earlier in the chapter, using PCA to reduce the dimensionality of the data. By plotting the dataset

and projecting cluster assignation onto the data, it is sometimes very obvious when a k-means implementation has fitted to a local minima or has overfit the data. The following plot demonstrates extreme overfitting of our previous k-means clustering algorithm to the `digits` dataset, artificially prompted by using **K = 150**. In this example, some clusters contain a single observation; there's really no way that this output would generalize to other samples well:



Plotting the elbow function or cluster assignments is quick to achieve and straightforward to interpret. However, we've spoken of these techniques in terms of being heuristics. If a dataset contains a deterministic number of classes, we may not be sure that a heuristic method will deliver generalizable results.

Another drawback is that visual plot checking is a very manual technique, which makes it poorly-suited for production environments or automation. In such circumstances, it's ideal to find a code-based, automatable method. One solid option in this case is **v-fold cross-validation**, a widely-used validation technique.

Cross-validation is simple to undertake. To make it work, one splits the dataset into *v* parts. One of the parts is set aside individually as a test set. The model is trained against the training data, which is all parts except the test set. Let's try this now, again using the `digits` dataset:

```
import numpy as np
from sklearn import cross_validation
from sklearn.cluster import KMeans
from sklearn.datasets import load_digits
from sklearn.preprocessing import scale


digits = load_digits()
data = scale(digits.data)
```

```
n_samples, n_features = data.shape
n_digits = len(np.unique(digits.target))
labels = digits.target



kmeans = KMeans(init='k-means++', n_clusters=n_digits,
n_init=n_digits)
cv = cross_validation.ShuffleSplit(n_samples, n_iter = 10, test_size
= 0.4, random_state = 0)
scores = cross_validation.cross_val_score(kmeans, data, labels, cv =
cv, scoring = 'adjusted_rand_score')
print(scores)
print(sum(scores)/cv.n_iter)
```

This code performs some now familiar data loading and preparation and initializes the k-means clustering algorithm. It then defines `cv`, the cross-validation parameters. This includes specification of the number of iterations, `n_iter`, and the amount of data that should be used in each fold. In this case, we're using 60% of the data samples as training data and 40% as test data.

We then apply the k-means model and `cv` parameters that we've specified within the cross-validation scoring function and print the results as `scores`. Let's take a look at these scores now:

```
[ 0.39276606  0.49571292  0.43933243  0.53573558
0.42459285
  0.55686854  0.4573401   0.49876358  0.50281585  0.4689295 ]

0.4772857426
```

This output gives us, in order, the adjusted Rand score for cross-validated, k-means++ clustering performed across each of the `10` folds in order. We can see that results do fluctuate between around `0.4` and `0.55`; the earlier ARI score for k-means++ without PCA fell within this range (at `0.465`). What we've created, then, is code that we can incorporate into our analysis in order to check the quality of our clustering automatically on an ongoing basis.

As noted earlier in this chapter, your choice of success measure is contingent on what information you already have. In most cases, you won't have access to ground truth labels from a dataset and will be obliged to use a measure such as the Silhouette Coefficient that we discussed previously.

Note

Sometimes, even using both cross-validation and visualizations won't provide a conclusive result. Especially with unfamiliar datasets, it's not unheard of to run into issues where some noise or secondary signal resolves better at a different *k* value than the signal you're attempting to analyze.

As with every other algorithm discussed in this book, it is imperative to understand the dataset one wishes to work with. Without this insight, it's entirely possible for even a technically correct and

rigorous analysis to deliver inappropriate conclusions. Chapter 6, *Text Feature Engineering* will discuss principles and techniques for the inspection and preparation of unfamiliar datasets more thoroughly.

Self-organizing maps

A SOM is a technique to generate topological representations of data in reduced dimensions. It is one of a number of techniques with such applications, with a better-known alternative being PCA. However, SOMs present unique opportunities, both as dimensionality reduction techniques and as a visualization format.

SOM – a primer

The SOM algorithm involves iteration over many simple operations. When applied at a smaller scale, it behaves similarly to k-means clustering (as we'll see shortly). At a larger scale, SOMs reveal the topology of complex datasets in a powerful way.

An SOM is made up of a grid (commonly rectangular or hexagonal) of nodes, where each node contains a weight vector that is of the same dimensionality as the input dataset. The nodes may be initialized randomly, but an initialization that roughly approximates the distribution of the dataset will tend to train faster.

The algorithm iterates as observations are presented as input. Iteration takes the following form:

- Identifying the winning node in the current configuration—the **Best Matching Unit** (**BMU**). The BMU is identified by measuring the Euclidean distance in the data space of all the weight vectors.
- The BMU is adjusted (moved) towards the input vector.
- Neighboring nodes are also adjusted, usually by lesser amounts, with the magnitude of neighboring movement being dictated by a neighborhood function. (Neighborhood functions vary. In this chapter, we'll use a Gaussian neighborhood function.)

This process repeats over potentially many iterations, using sampling if appropriate, until the network converges (reaching a position where presenting a new input does not provide an opportunity to minimize loss).

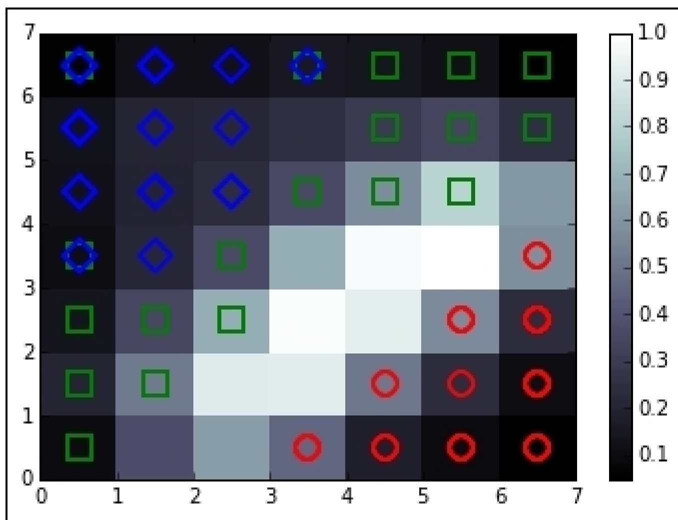
A node in an SOM is not unlike that of a neural network. It typically possesses a weight vector of length equal to the dimensionality of the input dataset. This means that the topology of the input dataset can be preserved and visualized through a lower-dimensional mapping.

The code for this SOM class implementation is available in the book repository in the `som.py` script. For now, let's start working with the SOM algorithm in a familiar context.

Employing SOM

As discussed previously, the SOM algorithm is iterative, being based around Euclidean distance comparisons of vectors.

This mapping tends to form a fairly readable 2D grid. In the case of the commonly-used Iris tutorial dataset, an SOM will map it out pretty cleanly:



In this diagram, the classes have been separated and also ordered spatially. The background coloring in this case is a clustering density measure. There is some minimal overlap between the blue and green classes, where the SOM performed an imperfect separation. On the Iris dataset, an SOM will tend to approach a converged solution on the order of 100 iterations, with little visible improvement after 1,000. For more complex datasets containing less clearly divisible cases, this process can take tens of thousands of iterations.

Awkwardly, there aren't implementations of the SOM algorithm within pre-existing Python packages like scikit-learn. This makes it necessary for us to use our own implementation.

The SOM code we'll be working with for this purpose is located in the associated GitHub repository. For now, let's take a look at the relevant script and get an understanding of how the code works:

```
import numpy as np
from sklearn.datasets import load_digits
from som import Som
from pylab import plot,axis,show,pcolor,colorbar,bone


digits = load_digits()
data = digits.data
labels = digits.target
```

At this point, we've loaded the `digits` dataset and identified `labels` as a separate set of data. Doing this will enable us to observe how the SOM algorithm separates classes when assigning them to `map`:

```
som = Som(16,16,64,sigma=1.0,learning_rate=0.5)
som.random_weights_init(data)
print("Initiating SOM.")
som.train_random(data,10000)
```

```
print("\n. SOM Processing Complete")

bone()
pcolor(som.distance_map().T)
colorbar()
```

At this point, we have utilized a `Som` class that is provided in a separate file, `Som.py`, in the repository. This class contains the methods required to deliver the SOM algorithm we discussed earlier in the chapter. As arguments to this function, we provide the dimensions of the map (After trialing a range of options, we'll start out with 16 x 16 in this case—this grid size gave the feature map enough space to spread out while retaining some overlap between groups.) and the dimensionality of the input data. (This argument determines the length of the weight vector within the SOM's nodes.) We also provide values for sigma and learning rate.

Sigma, in this case, defines the spread of the neighborhood function. As noted previously, we're using a Gaussian neighborhood function. The appropriate value for sigma varies by grid size. For an *8 x 8* grid, we would typically want to use a value of *1.0* for Sigma, while in this case we're using *1.3* for a *16 x 16* grid. It is fairly obvious when one's value for sigma is off; if the value is too small, values tend to cluster near the center of the grid. If the values are too large, the grid typically ends up with several large, empty spaces towards the center.

The *learning rate* self-explanatorily defines the initial learning rate for the SOM. As the map continues to iterate, the learning rate adjusts according to the following function:

$$learning\ rate(t) = learning\ rate / (1 + t / (0.5 * t))$$

Here, t is the iteration index.

We follow up by first initializing our SOM with random weights.

Note

As with k-means clustering, this initialization method is slower than initializing based on an approximation of the data distribution. A preprocessing step similar to that employed by the k-means++ algorithm would accelerate the SOM's runtime. Our SOM runs sufficiently quickly over the `digits` dataset to make this optimization unnecessary for now.

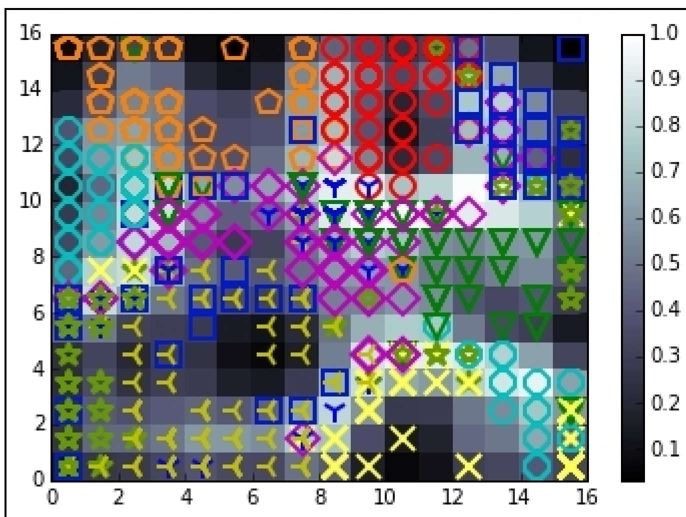
Next, we set up label and color assignations for each class, so that we can distinguish classes on the plotted SOM. Following this, we iterate through each data point.

On each iteration, we plot a class-specific marker for the BMU as calculated by our SOM algorithm.

When the SOM finishes iteration, we add a **U-Matrix** (a colorized matrix of relative observation density) as a monochrome-scaled plot layer:

```
labels[labels == '0'] = 0
labels[labels == '1'] = 1
labels[labels == '2'] = 2
labels[labels == '3'] = 3
labels[labels == '4'] = 4
labels[labels == '5'] = 5
labels[labels == '6'] = 6
labels[labels == '7'] = 7
labels[labels == '8'] = 8
labels[labels == '9'] = 9

markers = ['o', 'v', '1', '3', '8', 's', 'p', 'x', 'D', '*']
colors = ["r", "g", "b", "y", "c", (0,0.1,0.8), (1,0.5,0),
(1,1,0.3), "m", (0.4,0.6,0)]
for cnt,xx in enumerate(data):
   w = som.winner(xx)
   plot(w[0]+.5,w[1]+.5,markers[labels[cnt]],
   markerfacecolor='None', markeredgecolor=colors[labels[cnt]],
   markersize=12, markeredgewidth=2)
   axis([0,som.weights.shape[0],0,som.weights.shape[1]])
   show()
```

This code generates a plot similar to the following:



This code delivers a 16 x 16 node SOM plot. As we can see, the map has done a reasonably good job of separating each cluster into topologically distinct areas of the map. Certain classes (particularly the digits five in cyan circles and nine in green stars) have been located over multiple parts of the SOM space. For the most part, though, each class occupies a distinct region and it's fair to say that the SOM has been reasonably effective. The U-Matrix shows that regions with a high density of points are co-

habited by data from multiple classes. This isn't really a surprise as we saw similar results with k-means and PCA plotting.

Further reading

Victor Powell and Lewis Lehe provide a fantastic interactive, visual explanation of PCA at http://setosa.io/ev/principal-component-analysis/, this is ideal for readers who are new to the core concepts of PCA or who are not quite getting it.

For a lengthier and more mathematically-involved treatment of PCA, touching on underlying matrix transformations, Jonathon Shlens from Google research provides a clear and thorough explanation at http://arxiv.org/abs/1404.1100.

For a thorough worked example that translates Jonathon's description into clear Python code, consider Sebastian Raschka's demonstration using the Iris dataset at http://sebastianraschka.com/Articles/2015_pca_in_3_steps.html.

Finally, consider the sklearn documentation for more details on arguments to the PCA class at http://scikit-learn.org/stable/modules/generated/sklearn.decomposition.PCA.html.

For a lively and expert treatment of k-means, including detailed investigations of the conditions that cause it to fail, and potential alternatives in such cases, consider David Robinson's fantastic blog, variance explained at http://varianceexplained.org/r/kmeans-free-lunch/.

A specific discussion of the Elbow method is provided by Rick Gove at https://bl.ocks.org/rpgove/0060ff3b656618e9136b.

Finally, consider sklearn's documentation for another view on unsupervised learning algorithms, including k-means at http://scikit-learn.org/stable/tutorial/statistical_inference/unsupervised_learning.html.

Much of the existing material on Kohonen's SOM is either rather old, very high-level, or formally expressed. A decent alternative to the description in this book is provided by John Bullinaria at http://www.cs.bham.ac.uk/~jxb/NN/l16.pdf.

For readers interested in a deeper understanding of the underlying mathematics, I'd recommend reading the work of Tuevo Kohonen directly. The 2012 edition of self-organising maps is a great place to start.

The concept of multicollinearity, referenced in the chapter, is given a clear explanation for the unfamiliar at https://onlinecourses.science.psu.edu/stat501/node/344.

Summary

In this chapter, we've reviewed three techniques with a broad range of applications for preprocessing and dimensionality reduction. In doing so, you learned a lot about an unfamiliar dataset.

We started out by applying PCA, a widely-utilized dimensionality reduction technique, to help us understand and visualize a high-dimensional dataset. We then followed up by clustering the data using k-means clustering, identifying means of improving and measuring our k-means analysis through performance metrics, the elbow method, and cross-validation. We found that k-means on the `digits` dataset, taken as is, didn't deliver exceptional results. This was due to class overlap that we spotted through PCA. We overcame this weakness by applying PCA as a preprocess to improve our subsequent clustering results.

Finally, we developed an SOM algorithm that delivered a cleaner separation of the `digit` classes than PCA.

Having learned some key basics around unsupervised learning techniques and analytical methodology, let's dive into the use of some more powerful unsupervised learning algorithms.

Chapter 2. Deep Belief Networks

In the preceding chapter, we looked at some widely-used dimensionality reduction techniques, which enable a data scientist to get greater insight into the nature of datasets.

The next few chapters will focus on some more sophisticated techniques, drawing from the area of deep learning. This chapter is dedicated to building an understanding of how to apply the **Restricted Boltzmann Machine** (**RBM**) and manage the deep learning architecture one can create by chaining RBMs—the **deep belief network** (**DBN**). DBNs are trainable to effectively solve complex problems in text, image, and sound recognition. They are used by leading companies for object recognition, intelligent image search, and robotic spatial recognition.

The first thing that we're going to do is get a solid grounding in the algorithm underlying DBN; unlike clustering or PCA, this code isn't widely-known by data scientists and we're going to review it in some depth to build a strong working knowledge. Once we've worked through the theory, we'll build upon it by stepping through code that brings the theory into focus and allows us to apply the technique to real-world data. The diagnosis of these techniques is not trivial and needs to be rigorous, so we'll emphasize the thought processes and diagnostic techniques that enable us to effectively watch and control the success of your implementation.

By the end of this chapter, you'll understand how the RBM and DBN algorithms work, know how to use them, and feel confident in your ability to improve the quality of the results you get out of them. To summarize, the contents of this chapter are as follows:

- Neural networks – a primer
- Restricted Boltzmann Machines
- Deep belief networks

Neural networks – a primer

The RBM is a form of recurrent neural network. In order to understand how the RBM works, it is necessary to have a more general understanding of neural networks. Readers with an understanding of artificial neural network (hereafter neural network, for the sake of simplicity) algorithms will find familiar elements in the following description.

There are many accounts that cover neural networks in great theoretical detail; we won't go into great detail retreading this ground. For the purposes of this chapter, we will first describe the components of a neural network, common architectures, and prevalent learning processes.

The composition of a neural network

For unfamiliar readers, neural networks are a class of mathematical models that train to produce and optimize a definition for a function (or distribution) over a set of input features. The specific objective of a given neural network application can be defined by the operator using a performance measure (typically a cost function); in this way, neural networks may be used to classify, predict, or transform their inputs.

The use of the word neural in neural networks is the product of a long tradition of drawing from heavy-handed biological metaphors to inspire machine learning research. Hence, artificial neural networks algorithms originally drew (and frequently still draw) from biological neuronal structures.

A neural network is composed of the following elements:

- A learning process: A neural network learns by adjusting parameters within the weight function of its nodes. This occurs by feeding the output of a performance measure (as described previously, in supervised learning contexts this is frequently a cost function, some measure of inaccuracy relative to the target output of the network) into the learning function of the network. This learning function outputs the required weight adjustments (Technically, it typically calculates the partial derivatives—terms required by gradient descent.) to minimize the cost function.
- A set of neurons or weights: Each contains a weight function (the activation function) that manipulates input data. The activation function may vary substantially between networks (with one well-known example being the hyperbolic tangent). The key requirement is that the weights must be adaptive, that is,, adjustable based on updates from the learning process. In order to model non-parametrically (that is, to model effectively without defining details of the probability distribution), it is necessary to use both visible and hidden units. Hidden units are never observed.
- Connectivity functions: They control which nodes can relay data to which other nodes. Nodes may be able to freely relay input to one another in an unrestricted or restricted fashion, or they may be more structured in layers through which input data must flow in a directed fashion. There is a broad range of interconnection patterns, with different patterns producing very different network properties and possibilities.

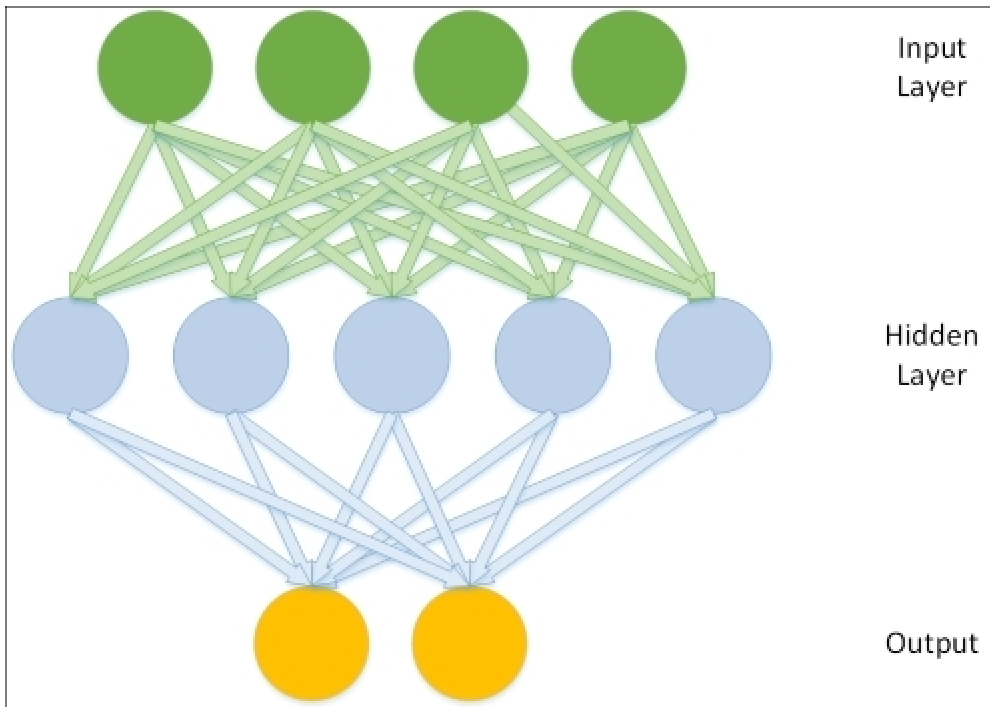
Utilizing this set of elements enables us to build a broad range of neural networks, ranging from the familiar directed acyclic graph (with perhaps the best-known example being the **Multi-Layer Perceptron** (**MLP**)) to creative alternatives. The **Self-Organizing Map** (**SOM**) that we employed in the preceding chapter was a type of neural network, with a unique learning process. The algorithm that we'll examine later in this chapter, that of the RBM, is another neural network algorithm with some unique properties.

Network topologies

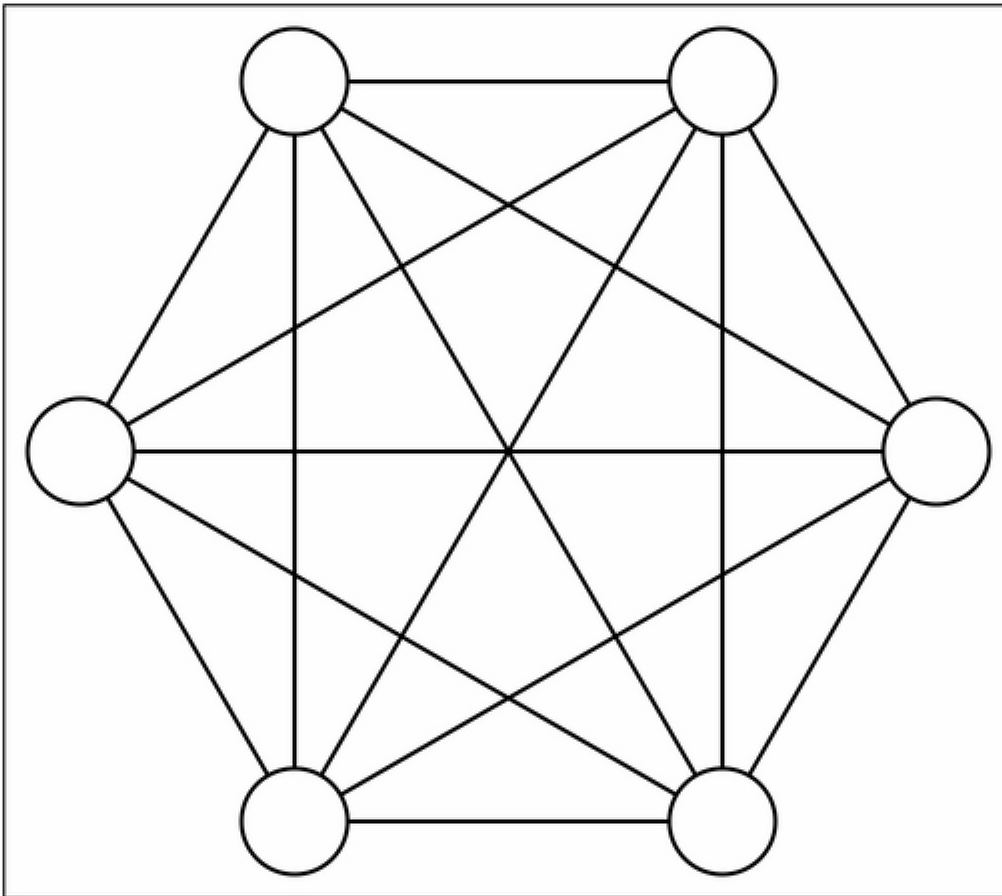
There are many variations on how the neurons in a neural network are connected, with structural decisions being an important factor in determining the network's learning capabilities. Common topologies in unsupervised learning tend to differ from those common to supervised learning. One common and now familiar unsupervised learning topology is that of the SOM that we discussed in the last chapter.

The SOM, as we saw, directly projects individual input cases onto a weight vector contained by each node. It then proceeds to reorder these nodes until an appropriate mapping of the dataset is converged on. The actual structure of the SOM was a variant based on the details of training, specific outcome of a given instance of training, and design decisions taken in structuring the network, but square or hexagonal grid structures are becoming increasingly common.

A very common topology type in supervised learning is that of a three-layer, feedforward network, with the classical case being the MLP. In this network topology model, the neurons in the network are split into layers, with each layer communicating to the layer "beyond" it. The first layer contains inputs that are fed to a hidden layer. The hidden layer develops a representation of the data using weight activations (with the right activation function, for example, sigmoid or gauss, an MLP can act as a universal function approximator) and activation values are communicated to the output layer. The output layer typically delivers network results. This topology, therefore, looks as follows:



Other network topologies deliver different capabilities. The topology of a Boltzmann Machine, for instance, differs from those described previously. The Boltzmann machine contains hidden and visible neurons, like those of a three-layer network, but all of these neurons are connected to one another in a directed, cyclic graph:



This topology makes Boltzmann machines stochastic—probabilistic rather than deterministic—and able to develop in one of several ways given a sufficiently complex problem. The Boltzmann machine is also generative, which means that it is able to fully (probabilistically) model all of the input variables, rather than using the observed variables to specifically model the target variables.

Which network topology is appropriate depends to a large extent on your specific challenge and the desired output. Each tends to be strong in certain areas. Furthermore, each of the topologies described here will be accompanied by a learning process that enables the network to iteratively converge on an (ideally optimal) solution.

There are a broad range of learning processes, with specific processes and topologies being more or less compatible with one another. The purpose of a learning process is to enable the network to adjust its weights, iteratively, in such a way as to create an increasingly accurate representation of the input data.

As with network topologies, there are a great many learning processes to consider. Some familiarity is assumed and a great many excellent resources on learning processes exist (some good examples are given at the end of this chapter). This section will focus on delivering a common characterization of learning processes, while later in the chapter, we'll look in greater detail at a specific example.

As noted, the objective of learning in a neural network is to iteratively improve the distribution of weights across the model so that it approximates the function underlying input data with increasing accuracy. This process requires a performance measure. This may be a classification error measure, as is commonly used in supervised, classification contexts (that is, with the backpropagation learning algorithm in MLP networks). In stochastic networks, it may be a probability maximization term (such as energy in energy-based networks).

In either case, once there is a measure to increase probability, the network is effectively attempting to reduce that measure using an optimization method. In many cases, the optimization of the network is achieved using **gradient descent**. As far as the gradient descent algorithm method is concerned, the size of your performance measure value on a given training iteration is analogous to the slope of your gradient. Minimizing the performance measure is therefore a question of descending that gradient to the point at which the error measure is at its lowest for that set of weights.

The size of the network's updates for the next iteration (the learning rate of your algorithm) may be influenced by the magnitude of your performance measure, or it may be hard-coded.

The weight updates by which your network adjusts may be derived from the error surface itself; if so, your network will typically have a means of calculating the gradient, that is, deriving the values to which updates need to adjust the parameters on your network's activated weight functions so as to continue to reduce the performance measure.

Having reviewed the general concepts underlying network topologies and learning methods, let's move into the discussion of a specific neural network, the RBM. As we'll see, the RBM is a key part of a powerful deep learning algorithm.

Restricted Boltzmann Machine

The RBM is a fundamental part of this chapter's subject deep learning architecture—the DBN. The following sections will begin by introducing the theory behind an RBM, including the architectural structure and learning processes.

Following that, we'll dive straight into the code for an RBM class, making links between the theoretical elements and functions in code. We'll finish by touching on the applications of RBMs and the practical factors associated with implementing an RBM.

Introducing the RBM

A Boltzmann machine is a particular type of stochastic, recurrent neural network. It is an energy-based model, which means that it uses an energy function to associate an energy value with each configuration of the network.

We briefly discussed the structure of a Boltzmann machine in the previous section. As mentioned, a Boltzmann machine is a directed cyclic graph, where every node is connected to all other nodes. This property enables it to model in a recurrent fashion, such that the model's outputs evolve and can be viewed over time.

The learning loop in a Boltzmann machine involves maximizing the probability of the training dataset, X. As noted, the specific performance measure used is energy, which is characterized as the negative log of the probability for a dataset X, given a vector of model parameters, Θ. This measure is calculated and used to update the network's weights in such a way as to minimize the free energy in the network.

The Boltzmann machine has seen particular success in processing image data, including photographs, facial features, and handwriting classification contexts.

Unfortunately, the Boltzmann machine is not practical for more challenging ML problems. This is due to the fact that there are challenges with the machine's ability to scale; as the number of nodes increases, the compute time grows exponentially, eventually leaving us in a position where we're unable to compute the free energy of the network.

Note

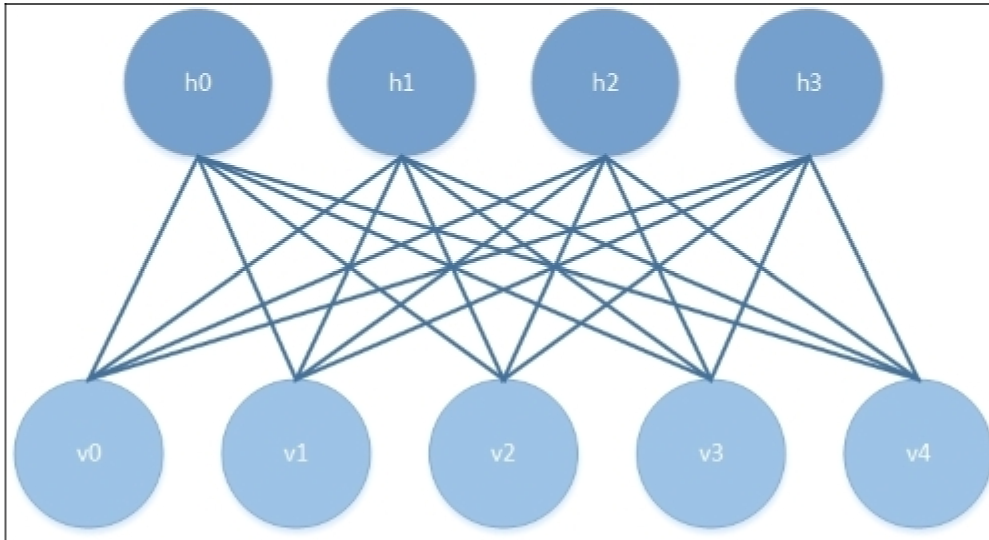
For those with an interest in the underlying formal reasoning, this happens because the probability of a data point, x, $p(x; \Theta)$, must integrate to 1 over all x. Achieving this requires that we use a partition function, Z, used as a normalizing constant. (Z is a constant such that multiplying a non-negative function by Z will make the non-negative function integrate to 1 over all inputs; in this case, over all x.)

The probability model function is a function of a set of normal distributions. In order to get the energy for our model, we need to differentiate for each of the model's parameters; however, this becomes complicated because of the partition function. Each model parameter produces equations dependent on other model parameters and we ultimately find ourselves unable to calculate the energy without (potentially) hugely expensive calculations, whose cost increases as the network scales.

In order to overcome the weaknesses of the Boltzmann machine, it is necessary to make adjustments to both the network topology and training process.

Topology

The main topological change that delivers efficiency improvements is the restriction of connectivity between nodes. First, one must prevent connection between nodes within the same layer. Additionally, all skip-layer connections (that is, direct connections between non-consecutive layers) must be prevented. A Boltzmann machine with this architecture is referred to as an RBM and appears as shown in the following diagram:



One advantage of this topology is that the hidden and visible layers are conditionally independent given one another. As such, it is possible to sample from one layer using the activations of the other.

Training

We observed previously that, for Boltzmann machines, the training time of the machine scales extremely poorly as the machine is scaled up to additional nodes, putting us in a position where we cannot evaluate the energy function that we're attempting to use in training.

The RBM is typically trained using a procedure with a different learning algorithm at its heart, the **Permanent Contrastive Divergence** (**PCD**) algorithm, which provides an approximation of maximum likelihood. PCD doesn't evaluate the energy function itself, but instead allows us to estimate the gradient of the energy function. With this information, we can proceed by making very small adjustments in the direction of the steepest gradient via which we may progress, as desired, toward the local minimum.

The PCD algorithm is made up of two phases. These are referred to as the positive and negative phases, and each phase has a corresponding effect on the energy of the model. The positive phase increases the probability of the training dataset, *X*, thus reducing the energy of the model. Following this, the negative

phase uses a sampling approach from the model to estimate the negative phase gradient. The overall effect of the negative phase is to decrease the probability of samples generated by the model.

Sampling in the negative phase and throughout the update process is achieved using a form of sampling called **Gibbs sampling**.

Note

Gibbs sampling is a variant of the **Markov Chain Monte Carlo** (**MCMC**) family of algorithms, and samples from an approximated multivariate probability distribution. What this means is, rather than using a summed calculation in building our probabilistic model (just as we might do, for instance, when we flip a coin a certain number of times; in such cases, we may sum the number of heads attempts as a proportion of the sum of all attempts), we approximate the value of an integral instead. The subject of how to create a probabilistic model by approximating an integral deserves more time than this book can give it. As such the *Further reading* section of this chapter provides an excellent paper reference. The key points to bear in mind for now (and stripping out a lot of important detail!) are that, instead of summing each case exactly once, we sample based on the (often non-uniform) distribution of the data in question. Gibbs sampling is a probabilistic sampling method for each parameter in a model, based on all of the other parameter values in that model. As soon as a new parameter value is obtained, it is immediately used in sampling calculations for other parameters.

Some of you may be asking at this point why PCD is necessary. Why not use a more familiar method, such as gradient descent with line search? To put it simply, we cannot easily calculate the free energy of our network as this calculation involves an integration across all the network's nodes. We recognized this limitation when we called out the big weakness of the Boltzmann machine—that the compute time grows exponentially as the number of nodes increases, leaving us in a situation where we're trying to minimize a function whose value we cannot calculate!

What PCD provides is a way to estimate the gradient of the energy function. This enables an approximation of the network's free energy, which is fast enough to be viable for application and has shown to be generally accurate. (Refer to the *Further reading* section for a performance comparison.)

As we saw previously, the RBM's probability model function is the joint distribution of our model parameters, making Gibbs sampling appropriate!

The training loop in an initialized RBM involves several steps:

1. We obtain the current iteration's activated hidden layer weight values.
2. We perform the positive phase of PCD, using the state of the Gibbs chain from the previous iteration as input.
3. We perform the negative phase of PCD using the pre-existing state of the Gibbs chain. This gives us the free energy value.
4. We update the activated weights on the hidden layer using the energy value we've calculated.

This algorithm allows the RBM to iteratively step toward a decreased free energy value. The RBM continues to train until both the probability of the training dataset integrates to one and free energy is equal to zero, at which point the RBM has converged.

Now that we've had a chance to review the RBM's topology and training process, let's apply the algorithm to classify a substantial real dataset.

Applications of the RBM

Now that we have a general working knowledge of the RBM algorithm, let's walk through code to create an RBM. We'll be working with an RBM class that will allow us to classify the MNIST handwritten digits dataset. The code we're about to review does the following:

- It sets up the initial parameters of an RBM, including layer size, shareable bias vectors, and shareable weight matrix for connectivity with external network structures (this enables deep belief networks)
- It defines functions for communication and inference between hidden and visible layers
- It defines functions that allow us to update the parameters of network nodes
- It defines functions that handle efficient sampling for the learning process, using PCD-k to accelerate sampling (making it possible to compute in a reasonable frame of time)
- It defines functions that compute the free energy of the model (used to calculate the gradient required for PCD-k updates)
- It identifies the **Psuedo-Likelihood** (**PL**), usable as a log-likelihood proxy to guide the selection of appropriate hyperparameters

Let's begin examining our `RBM` class:

```
class RBM(object):
    def __init__(
        self,
        input=None,
        n_visible=784,
        n_hidden=500,
        w=None,
        hbias=None,
        vbias=None,
        numpy_rng=None,
        theano_rng=None
    ):
```

The first element that we need to build is an `RBM` constructor, which we can use to define the parameters of the model, such as the number of visible and hidden nodes (`n_visible` and `n_hidden`) as well as additional parameters that can be used to adjust how the RBM's inference functions and CD updates are performed.

The `w` parameter can be used as a pointer to a shared weight matrix. This becomes more relevant when implementing a DBN, as we'll see later in the chapter; in such architectures, the weight matrix needs to be shared between different parts of the network.

The `hbias` and `vbias` parameters are used similarly as optional references to shared hidden and visible (respectively) units' bias vectors. Again, these are used in DBNs.

The `input` parameter enables the RBM to be connected, top-to-tail, to other graph elements. This allows one to, for instance, chain RBMs.

Having set up this constructor, we next need to flesh out each of the preceding parameters:

```
        self.n_visible = n_visible
        self.n_hidden = n_hidden

        if numpy_rng is None:
            numpy_rng = numpy.random.RandomState(1234)

        if theano_rng is None:
            theano_rng = RandomStreams(numpy_rng.randint(2 ** 30))
```

This is fairly straightforward stuff; we set the visible and hidden nodes for our RBM and set up two random number generators. The `theano_rng` parameter will be used later in our code to sample from the RBM's hidden units:

```
        if W is None:
            initial_W = numpy.asarray(
                numpy_rng.uniform(
                    low=-4 * numpy.sqrt(6. / (n_hidden + n_visible)),
                    high=4 * numpy.sqrt(6. / (n_hidden + n_visible)),
                    size=(n_visible, n_hidden)
                ),
                dtype=theano.config.floatX
            )
```

This code switches up the data type for `W` so that it can be run over the GPU. Next, we set up shared variables using `theano.shared`, which allows a variable's storage to be shared between functions that it appears in. Within the current example, the shared variables that we create will be the weight vector (`W`) and bias variables for hidden and visible units (`hbias` and `vbias`, respectively). When we move on to creating deep networks with multiple components, the following code will allow us to share components between parts of our networks:

```
            W = theano.shared(value=initial_W, name='W', borrow=True)

        if hbias is None:
            hbias = theano.shared(
                value=numpy.zeros(
                    n_hidden,
                    dtype=theano.config.floatX
                ),
                name='hbias',
                borrow=True
            )
```

```
        if vbias is None:
            vbias = theano.shared(
                value=numpy.zeros(
                    n_visible,
                    dtype=theano.config.floatX
                ),
                name='vbias',
                borrow=True
            )
```

At this point, we're ready to initialize the input layer as follows:

```
        self.input = input
        if not input:
            self.input = T.matrix('input')

        self.W = W
        self.hbias = hbias
        self.vbias = vbias
        self.theano_rng = theano_rng
        self.params = [self.W, self.hbias, self.vbias]
```

As we now have an initialized `input` layer, our next task is to create the symbolic graph that we described earlier in the chapter. Achieving this is a matter of creating functions to manage the interlayer propagation and activation computation operations of the network:

```
def propup(self, vis):
        pre_sigmoid_activation = T.dot(vis, self.W) + self.hbias
        return [pre_sigmoid_activation,
T.nnet.sigmoid(pre_sigmoid_activation)]


    def propdown(self, hid):
        pre_sigmoid_activation = T.dot(hid, self.W.T) + self.vbias
        return [pre_sigmoid_activation,
T.nnet.sigmoid(pre_sigmoid_activation)]
```

These two functions pass the activation of one layer's units to the other layer. The first function passes the visible units' activation upward to the hidden units so that the hidden units can compute their activation conditional on a sample of the visible units. The second function does the reverse—propagating the hidden layer's activation downward to the visible units.

It's probably worth asking why we're creating both `propup` and `propdown`. As we reviewed it, PCD only requires that we perform sampling from the hidden units. So what's the value of `propup`?

In a nutshell, sampling from the visible layer becomes useful when we want to sample from the RBM to review its progress. In most applications where our RBM is processing visual data, it is immediately

valuable to periodically take the output of sampling from the visible layer and plot it, as shown in the following example:

As we can see here, over the course of iteration, our network begins to change its labeling; in the first case, 7 morphs into 9, while elsewhere 9 becomes 6 and the network gradually reaches a definition of 3-ness.

As we discussed earlier, it's helpful to have as many views on the operation of your RBM as possible to ensure that it's delivering meaningful results. Sampling from the outputs it generates is one way to improve this visibility.

Armed with information about the visible layer's activation, we can deliver a sample of the unit activations from the hidden layer, given the activation of the hidden nodes:

```
    def sample_h_given_v(self, v0_sample):

 pre_sigmoid_h1, h1_mean = self.propup(v0_sample)
    h1_sample = self.theano_rng.binomial(size=h1_mean.shape,
    n=1, p=h1_mean, dtype=theano.config.floatX)

    return [pre_sigmoid_h1, h1_mean, h1_sample]
```

Likewise, we can now sample from the visible layer given hidden unit activation information:

```
    def sample_v_given_h(self, h0_sample):
    pre_sigmoid_v1, v1_mean = self.propdown(h0_sample)
      v1_sample = self.theano_rng.binomial(size=v1_mean.shape,
      n=1, p=v1_mean, dtype=theano.config.floatX)

      return [pre_sigmoid_v1, v1_mean, v1_sample]
```

We've now achieved the connectivity and update loop required to perform a Gibbs sampling step, as described earlier in this chapter. Next, we should define this sampling step!

```
def gibbs_hvh(self, h0_sample):

    pre_sigmoid_v1, v1_mean, v1_sample =
    self.sample_v_given_h(h0_sample)
    pre_sigmoid_h1, h1_mean, h1_sample =
    self.sample_h_given_v(v1_sample)
    return [pre_sigmoid_v1, v1_mean, v1_sample,
            pre_sigmoid_h1, h1_mean, h1_sample]
```

As discussed, we need a similar function to sample from the visible layer:

```
def gibbs_vhv(self, v0_sample):

    pre_sigmoid_h1, h1_mean, h1_sample =
    self.sample_h_given_v(v0_sample)
    pre_sigmoid_v1, v1_mean, v1_sample =
    self.sample_v_given_h(h1_sample)
    return [pre_sigmoid_h1, h1_mean, h1_sample,
            pre_sigmoid_v1, v1_mean, v1_sample]
```

The code that we've written so far gives us some of our model. It set up the nodes and layers and connections between layers. We've written the code that we need in order to update the network based on Gibbs sampling from the hidden layer.

What we're still missing is code that allows us to perform the following:

- Compute the free energy of the model. As we discussed, the model uses energy as the term to do the following:
    - Implement PCD using our Gibbs sampling step code, and setting the Gibbs step count parameter, $k = 1$, to compute the parameter gradient for gradient descent
    - Create a means to feed the output of PCD (the computed gradient) to our previously defined network update code
- Develop the means to track the progress and success of our RBM throughout the training.

First off, we'll create the means to calculate the free energy of our RBM. Note that this is the inverse log of the probability distribution for the hidden layer, which we discussed earlier:

```
def free_energy(self, v_sample):

      wx_b = T.dot(v_sample, self.W) + self.hbias
      vbias_term = T.dot(v_sample, self.vbias)
      hidden_term = T.sum(T.log(1 + T.exp(wx_b)), axis=1)
      return -hidden_term - vbias_term
```

Next, we'll implement PCD. At this point, we'll be setting a couple of interesting parameters. The `lr`, short for learning rate, is an adjustable parameter used to adjust learning speed. The `k` parameter points to the number of steps to be performed by PCD (remember the PCD-k notation from earlier in the chapter?).

We discussed the PCD as containing two phases, positive and negative. The following code computes the positive phase of PCD:

```
def get_cost_updates(self, lr=0.1, persistent = , k=1):

        pre_sigmoid_ph, ph_mean, ph_sample =
        self.sample_h_given_v(self.input)



            chain_start = persistent
```

Meanwhile, the following code implements the negative phase of PCD. To do so, we scan the `gibbs_hvh` function `k` times, using Theano's scan operation, performing one Gibbs sampling step with each scan. After completing the negative phase, we acquire the free energy value:

```
        (
            [
                pre_sigmoid_nvs,
                nv_means,
                nv_samples,
                pre_sigmoid_nhs,
                nh_means,
                nh_samples
            ],
            updates
        ) = theano.scan(
            self.gibbs_hvh,
            outputs_info=[None, None, None, None, None, chain_start],
            n_steps=k
        )


        chain_end = nv_samples[-1]

        cost = T.mean(self.free_energy(self.input)) - T.mean(
            self.free_energy(chain_end))

        gparams = T.grad(cost, self.params,
        consider_constant=[chain_end])
```

Having written code that performs the full PCD process, we need a way to feed the outputs to our network. At this point, we're able to connect our PCD learning process to the code to update the network that we reviewed earlier. The preceding updates dictionary points to `theano.scan` of the `gibbs_hvh` function. As you may recall, `gibbs_hvh` currently contains rules for random states of `theano_rng`. What we need to do now is add the new parameter values and variable containing the state of the Gibbs chain to the dictionary (the `updates` variable):

```
        for gparam, param in zip(gparams, self.params):
            updates[param] = param - gparam * T.cast(
                lr,
                dtype=theano.config.floatX
            )

            updates = nh_samples[-1]
            monitoring_cost =
            self.get_pseudo_likelihood_cost(updates)


        return monitoring_cost, updates
```

We now have almost all the parts that we need to make our RBM work. What's clearly missing is a means to inspect training, either during or after completion, to ensure that our RBM is learning an appropriate representation of the data.

We talked previously about how to train an RBM, specifically about challenges posed by the partition function. Furthermore, earlier in the code, we implemented one means by which we can inspect an RBM during training; we created the `gibbs_vhv` function to perform Gibbs sampling from the model.

In our previous discussion around how to validate an RBM, we discussed visually plotting the filters that the RBM has created. We'll review how this can be achieved shortly.

The final possibility is to use the inverse log of the PL as a more tractable proxy to the likelihood itself. Technically, the log-PL is the sum of the log-probabilities of each data point (each `x`) conditioned on all other data points. As discussed, this becomes too expensive with larger-dimensional datasets, so a stochastic approximation to log-PL is used.

We referenced a function that will enable us to get PL cost during the `get_cost_updates` function, specifically the `get_pseudo_likelihood_cost` function. Now it's time to flesh out this function and obtain the pseudo-likelihood:

```
def get_pseudo_likelihood_cost(self, updates):

        bit_i_idx = theano.shared(value=0, name='bit_i_idx')
        xi = T.round(self.input)

        fe_xi = self.free_energy(xi)

        xi_flip = T.set_subtensor(xi[:, bit_i_idx], 1 - xi[:,
        bit_i_idx])

        fe_xi_flip = self.free_energy(xi_flip)

        cost = T.mean(self.n_visible *
        T.log(T.nnet.sigmoid(fe_xi_flip - fe_xi)))
```

```
        updates[bit_i_idx] = (bit_i_idx + 1) % self.n_visible

        return cost
```

We've now filled out each element on the list of missing components and have completely reviewed the `RBM` class. We've explored how each element ties into the theory behind the RBM and should now have a thorough understanding of how the RBM algorithm works. We understand what the outputs of our RBM will be and will soon be able to review and assess them. In short, we're ready to train our RBM. Beginning the training of the RBM is a matter of running the following code, which triggers the `train_set_x` function. We'll discuss this function in greater depth later in this chapter:

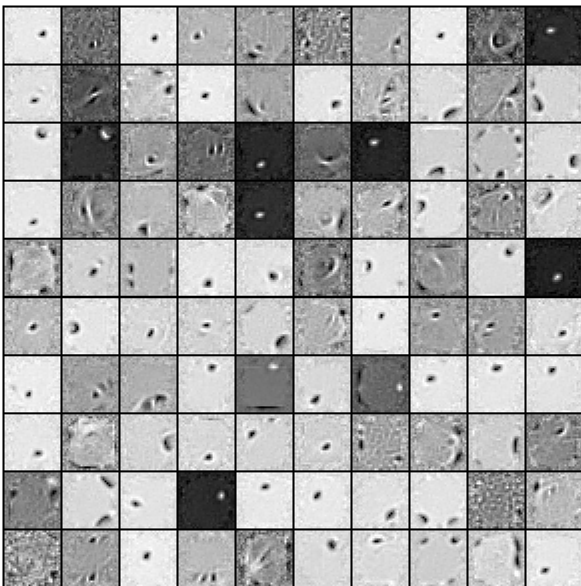
```
    train_rbm = theano.function(
        [index],
        cost,
        updates=updates,
        givens={
            x: train_set_x[index * batch_size: (index + 1) *
            batch_size]
        },
        name='train_rbm'
    )

    plotting_time = 0.
    start_time = time.clock()
```

Having updated the RBM's updates and training set, we run through training epochs. Within each epoch, we train over the training data before plotting the weights as a matrix (as described earlier in the chapter):

```
    for epoch in xrange(training_epochs):

        mean_cost = []
        for batch_index in xrange(n_train_batches):
            mean_cost += [train_rbm(batch_index)]

        print 'Training epoch %d, cost is ' % epoch,
        numpy.mean(mean_cost)

        plotting_start = time.clock()
        image = Image.fromarray(
            tile_raster_images(
                X=rbm.W.get_value(borrow=True).T,
                img_shape=(28, 28),
                tile_shape=(10, 10),
                tile_spacing=(1, 1)
```

```
            )
        )
        image.save('filters_at_epoch_%i.png' % epoch)
        plotting_stop = time.clock()
        plotting_time += (plotting_stop - plotting_start)

    end_time = time.clock()

    pretraining_time = (end_time - start_time) - plotting_time

    print ('Training took %f minutes' % (pretraining_time / 60.))
```

The weights tend to plot fairly recognizably and resemble Gabor filters (linear filters commonly used for edge detection in images). If your dataset is handwritten characters on a fairly low-noise background, you tend to find that the weights trace the strokes used. For photographs, the filters will approximately trace edges in the image. The following image shows an example output:



Finally, we create the persistent Gibbs chains that we need to derive our samples. The following function performs a single Gibbs step, as discussed previously, then updates the chain:

```
plot_every = 1000

    (
        [
            presig_hids,
            hid_mfs,
            hid_samples,
            presig_vis,
```

```
            vis_mfs,
            vis_samples
        ],
        updates
    ) = theano.scan(
        rbm.gibbs_vhv,
        outputs_info=[None, None, None, None, None,
persistent_vis_chain],
        n_steps=plot_every
    )
```

This code runs the `gibbs_vhv` function we described previously, plotting network output samples for our inspection:

```
    updates.update({persistent_vis_chain: vis_samples[-1]})
    sample_fn = theano.function(
        [],
        [
            vis_mfs[-1],
            vis_samples[-1]
        ],
        updates=updates,
        name='sample_fn'
    )

    image_data = numpy.zeros(
        (29 * n_samples + 1, 29 * n_chains - 1),
        dtype='uint8'
    )
    for idx in xrange(n_samples):

        vis_mf, vis_sample = sample_fn()
        print ' ... plotting sample ', idx
        image_data[29 * idx:29 * idx + 28, :] = tile_raster_images(
            X=vis_mf,
            img_shape=(28, 28),
            tile_shape=(1, n_chains),
            tile_spacing=(1, 1)
        )

    image = Image.fromarray(image_data)
    image.save('samples.png')
```

At this point, we have an entire RBM. We have the PCD algorithm and the ability to update the network using this algorithm and Gibbs sampling. We have several visible output methods so that we can assess how well our RBM has trained.

However, we're not done yet! Next, we'll begin to see what the most frequent and powerful application of the RBM is.

Further applications of the RBM

We can use the RBM as an ML algorithm in and of itself. It functions comparably well with other algorithms. Advantageously, it can be scaled up to a point where it can learn high-dimensional datasets. However, this isn't where the real strength of the RBM lies.

The RBM is most commonly used as a pretraining mechanism for a highly effective deep network architecture called a DBN. DBNs are extremely powerful tools to learn and classify a range of image datasets. They possess a very good ability to generalize to unknown cases and are among the best image-learning tools available. For this reason, DBNs are in use at many of the world's top tech and data science companies, primarily in image search and recognition contexts.

Deep belief networks

A DBN is a graphical model, constructed using multiple stacked RBMs. While the first RBM trains a layer of features based on input from the pixels of the training data, subsequent layers treat the activations of preceding layers as if they were pixels and attempt to learn the features in subsequent hidden layers. This is frequently described as learning the representation of data and is a common theme in deep learning.

How many multiple RBMs there should be depends on what is needed for the problem at hand. From a practical perspective, it's a trade-off between increasing accuracy and increasing computational cost. It is the case that each layer of RBMs will improve the lower bound of the log probability of the training data. In other words; the DBN almost inevitably becomes less bad with each additional layer of features.

As far as layer size is concerned, it is generally advantageous to reduce the number of nodes in the hidden layers of successive RBMs. One should avoid contexts in which an RBM has at least as many visible units as the RBM preceding it has hidden units (which raises the risk of simply learning the identity function of the network).

It can be advantageous (but is by no means necessary) when successive RBMs decrease in layer size until the final RBM has a layer size approximating the dimensionality of variance in the data. Affixing an MLP to the end of a DBN whose layers have too many nodes will harm classification performance; it's like trying to affix a drinking straw to the end of a hosepipe! Even an MLP with many neurons may not successfully train in such contexts. On a related note, it has been noted that even if the layers don't contain very many nodes, with enough layers, more or less any function can be modeled.

Determining what the dimensionality of variance in the data is, is not a simple task. One tool that can support this task is PCA; as we saw in the preceding chapter, PCA can enable us to get a reasonable idea as to how many components of meaningful size exist in the input data.

Training a DBN

Training a DBN is typically done greedily, which is to say that it trains to optimize locally at each layer, rather than attempting to reach a global optimum. The learning process is as follows:

- The first layer of the DBN is trained using the method that we saw in our earlier discussion of RBM learning. As such, the first layer converts its data distribution to a posterior distribution using Gibbs sampling over the hidden units.
- This distribution is far more conducive for RBM training than the input data itself so the next RBM layer learns that distribution!
- Successive RBM layers continue to train on the samples output by preceding layers.
- All of the parameters within this architecture are tuned using a performance measure.

This performance measure may vary. It may be a log-likelihood proxy used in gradient descent, as discussed earlier in the chapter. In supervised contexts, a classifier (for example, an MLP) can be added as the final layer of the architecture and prediction accuracy can be used as the performance measure to fine-tune the deep architecture.

Let's move on to using the DBN in practice.

Applying the DBN

Having discussed the DBN and theory surrounding it, it's time to set up our own. We'll be working in a similar way to the RBM, by walking through a `DBN` class and connecting the code to the theory, discussing what to expect and how to review the network's performance, before initializing and training our network to see it in action.

Let's take a look at our `DBN` class:

```
class DBN(object):

    def __init__(self, numpy_rng, theano_rng=None, n_ins=784,
                 hidden_layers_sizes=[500, 500], n_outs=10):

        self.sigmoid_layers = []
        self.rbm_layers = []
        self.params = []
        self.n_layers = len(hidden_layers_sizes)

        assert self.n_layers > 0

        if not theano_rng:
            theano_rng = RandomStreams(numpy_rng.randint(2 ** 30))


        self.x = T.matrix('x')
        self.y = T.ivector('y')
```

The `DBN` class contains a number of parameters that bear further explanation. The `numpy_rng` and `theano_rng` parameters, used to determine initial weights, are already familiar from our examination of the `RBM` class. The `n_ins` parameter is a pointer to the dimension (in features) of the DBN's input. The `hidden_layers_sizes` parameter is a list of hidden layer sizes. Each value in this list will guide the `DBN` constructor in creating an RBM layer of the relevant size; as you'll note, the `n_layers` parameter refers to the number of layers in the network and is set by `hidden_layers_sizes`. Adjustment of values in this list enables us to make DBNs whose layer sizes taper down from the input layer size, to increasingly succinct representations, as discussed earlier in the chapter.

It's also worth noting that `self.sigmoid_layers` will store the MLP component (the final layer of the DBN), while `self.rbm_layers` stores the RBM layers used to pretrain the MLP.

With this done, we do the following to complete our DBN architecture:

- We create `n_layers` sigmoid layers
- We connect the sigmoid layers to form an MLP

- We construct an RBM for each sigmoid layer with a shared weight matrix and hidden bias between each sigmoid layer and RBM

The following code creates `n_layers` many layers with sigmoid activations; first creating the input layer, then creating hidden layers whose size corresponds to the values in our `hidden_layers_sizes` list:

```
for i in xrange(self.n_layers):

    if i == 0:
        input_size = n_ins
    else:
               input_size = hidden_layers_sizes[i - 1]
    if i == 0:
        layer_input = self.x
    else:
        layer_input = self.sigmoid_layers[-1].output

    sigmoid_layer = HiddenLayer(rng=numpy_rng,
                                input=layer_input,
                                n_in=input_size,
                                n_out=hidden_layers_sizes[i],
                                activation=T.nnet.sigmoid)
    self.sigmoid_layers.append(sigmoid_layer)

    self.params.extend(sigmoid_layer.params)
```

Next up, we create an RBM that shares weights with the sigmoid layers. This directly leverages the `RBM` class that we described previously:

```
    rbm_layer = RBM(numpy_rng=numpy_rng,
                    theano_rng=theano_rng,
                    input=layer_input,
                    n_visible=input_size,
                    n_hidden=hidden_layers_sizes[i],
                    W=sigmoid_layer.W,
                    hbias=sigmoid_layer.b)
    self.rbm_layers.append(rbm_layer)
```

Finally, we add a logistic regression layer to the end of the DBN so as to form an MLP:

```
self.logLayer = LogisticRegression(
    input=self.sigmoid_layers[-1].output,
    n_in=hidden_layers_sizes[-1],
    n_out=n_outs)
self.params.extend(self.logLayer.params)
```

```
        self.finetune_cost =
self.logLayer.negative_log_likelihood(self.y)

        self.errors = self.logLayer.errors(self.y)
```

Now that we've put together our `MLP` class, let's construct `DBN`. The following code constructs the network with `28 * 28` inputs (that is, `28*28` pixels in the MNIST image data), three hidden layers of decreasing size, and `10` output values (for each of the `10` handwritten number classes in the MNIST dataset):

```
    numpy_rng = numpy.random.RandomState(123)
    print '... building the model'
    dbn = DBN(numpy_rng=numpy_rng, n_ins=28 * 28,
              hidden_layers_sizes=[1000, 800, 720],
              n_outs=10)
```

As discussed earlier in this section, a DBN trains in two stages—a layer-wise pretraining in which each layer takes the output of the preceding layer to train on, which is followed by a fine-tuning step (backpropagation) that allows for weight adjustment across the whole network. The first stage, pretraining, is achieved by performing one step of PCD within each layer's RBM. The following code will perform this pretraining step:

```
    print '... getting the pretraining functions'
    pretraining_fns =
    dbn.pretraining_functions(train_set_x=train_set_x,
    batch_size=batch_size, k=k)

    print '... pre-training the model'
    start_time = time.clock()

    for i in xrange(dbn.n_layers):
        for epoch in xrange(pretraining_epochs):
            c = []
            for batch_index in xrange(n_train_batches):
                c.append(pretraining_fns[i](index=batch_index,
                                            lr=pretrain_lr))
            print 'Pre-training layer %i, epoch %d, cost ' % (i,
epoch),
            print numpy.mean(c)

    end_time = time.clock()
```

Running the pretrained DBN is then achieved by the following command:

```
python code/DBN.py
```

Note

Note that even with GPU acceleration, this code will spend quite a lot of time pretraining, and it is therefore suggested that you run it overnight.

Validating the DBN

Validation of a DBN as a whole is done in a very familiar way. We can use the minimal validation error from cross-validation as one error measure. However, the minimal cross-validation error can underestimate the error expected on cross-validation data as the meta-parameters may overfit to the new data.

As such, we should use our cross-validation error to adjust our metaparameters until the cross-validation error is minimized. Then we should expose our DBN to the held-out test set, using test error as our validation measure. Our `DBN` class performs exactly this training process.

However, this doesn't tell us exactly what to do if the network fails to train adequately. What do we do if our DBN is underperforming?

The first thing to do is recognize the potential causes and, in this area, there are some usual culprits. We know that the training of underlying RBMs is also quite tricky and any individual layer may fail to train. Thankfully, our `RBM` class gives us the ability to tap into and view the weights (filters) being generated by each layer, and we can plot these to get a view on what our network is attempting to represent.

Additionally, we want to ask whether our network is overfitting, or else, underfitting. Either is entirely possible and it's useful to recognize how and why this might be happening. In the case of underfitting, the training process may simply be unable to find good parameters for the model. This is particularly common when you are using a larger network to resolve a large problem space, but can be seen even with some smaller models. If you think that underfitting might be happening with your DBN, you have a couple of options. The first is to simply reduce the size of your hidden layers. This may, or may not, work well. A better alternative is to gradually taper your hidden layers such that each layer learns a refined version of the preceding layer's representation. How to do this, how sharply to taper, and when to stop is a matter of trial and error in the first case and of experience-based learning over the long term.

Overfitting is a well-known phenomenon where your algorithm trains overly specifically on the training data provided. This class of problem is typically identified at the point of cross-validation (where your error rate will increase dramatically), but can be quite pernicious. Means of resolving an overfitting issue do exist; one can increase the training dataset size. A more heavy-handed Bayesian approach would be to attach an additional criterion (for example, a prior) that is used to reduce the value of fitting the training data. Some of the most effective methods to improve classification performance are preprocessing methods, which we'll discuss in Chapters 6, *Text Feature Engineering* and Chapter 7, *Feature Engineering Part II*.

Though this code will initialize from a predefined position (given a seed value), the stochastic nature of the model means that it will quickly diverge and results may vary. When running on my system, this DBN achieved a minimal cross-validation error of 1.19%. More importantly, it achieved a test error of

1.30% after 46 supervised epochs. These are good results; indeed, they are comparable with field-leading examples!

Further reading

For a primer on neural networks, it makes sense to read from a range of sources. There are many concerns to be aware of and different authors emphasize on different material. A solid introduction is provided by Kevin Gurney in An Introduction to Neural Networks.

An excellent piece on the intuitions underlying Markov Chain Monte Carlo is available at http://twiecki.github.io/blog/2015/11/10/mcmc-sampling/.

For readers with a specific interest in the intuitions supporting Gibbs Sampling, Philip Resnik, and Eric Hardisty's paper, *Gibbs Sampling for the Uninitiated*, provides a technical, but clear description of how Gibbs works. It's particularly notable to have some really first-rate analogies! Find them at https://www.umiacs.umd.edu/~resnik/pubs/LAMP-TR-153.pdf.

There aren't many good explanations of Contrastive Divergence, one I like is provided by Oliver Woodford at http://www.robots.ox.ac.uk/~ojw/files/NotesOnCD.pdf. If you're a little daunted by the heavy use of formal expressions, I would still recommend that you read it for its articulate description of theory and practical concerns involved.

This chapter used the Theano documentation available at http://deeplearning.net/tutorial/contents.html as a base for discussion and implementation of RBM and DBN classes.

Summary

We've covered a lot of ground in this chapter! We began with an overview of Neural Networks, focusing on the general properties of topology and learning method before taking a deep dive into the RBM algorithm and RBM code itself. We took this solid understanding forward to create a DBN. In doing so, we linked the DBN theory and code together, before firing up our DBN to work over the MNIST dataset. We performed image classification in a 10-class problem and achieved an extremely competitive result, with classification error below 2%!

In the next chapter, we'll continue to build on your mastery of deep learning by introducing you to another deep learning architecture—**Stacked Denoising Autoencoders** (**SDA**).

Chapter 3. Stacked Denoising Autoencoders

In this chapter, we'll continue building our skill with deep architectures by applying **Stacked Denoising Autoencoders** (**SdA**) to learn feature representations for high-dimensional input data.

We'll start, as before, by gaining a solid understanding of the theory and concepts that underpin autoencoders. We'll identify related techniques and call out the strengths of autoencoders as part of your data science toolkit. We'll discuss the use of **Denoising Autoencoders** (**dA**), a variation of the algorithm that introduces stochastic corruption to the input data, obliging the autoencoder to decorrupt the input and, in so doing, build a more effective feature representation.

We'll follow up on theory, as before, by walking through the code for a dA class, linking theory and implementation details to build a strong understanding of the technique.

At this point, we'll take a journey very similar to that taken in the preceding chapter—by stacking dA, we'll create a deep architecture that can be used to pretrain an MLP network, which offers substantial performance improvements in a range of unsupervised learning applications including speech data processing.

Autoencoders

The autoencoder (also called the **Diabolo network**) is another crucial component of deep architectures. The autoencoder is related to the RBM, with autoencoder training resembling RBM training; however, autoencoders can be easier to train than RBMs with contrastive divergence and are thus preferred in contexts where RBMs train less effectively.

Introducing the autoencoder

An autoencoder is a simple three-layer neural network whose output units are directly connected back to the input units. The objective of the autoencoder is to encode the **i-dimensional** input into an **h-dimensional** representation, where $h < i$, before reconstructing (decoding) the input at the output layer. The training process involves iteration over this process until the reconstruction error is minimized—at which point one should have arrived at the most efficient representation of input data (should, barring the possibility of arriving at local minima!).

In a preceding chapter, we discussed PCA as being a powerful dimensionality reduction technique. This description of autoencoders as finding the most efficient reduced-dimensional representation of input data will no doubt be familiar and you may be asking why we're exploring another technique that fulfils the same role.

The simple answer is that like the SOM, autoencoders can provide nonlinear reductions, which enables them to process high-dimensional input data more effectively than PCA. This revives a form of our earlier question—why discuss autoencoders if they deliver what an SOM does, without even providing the illuminating visual presentation?

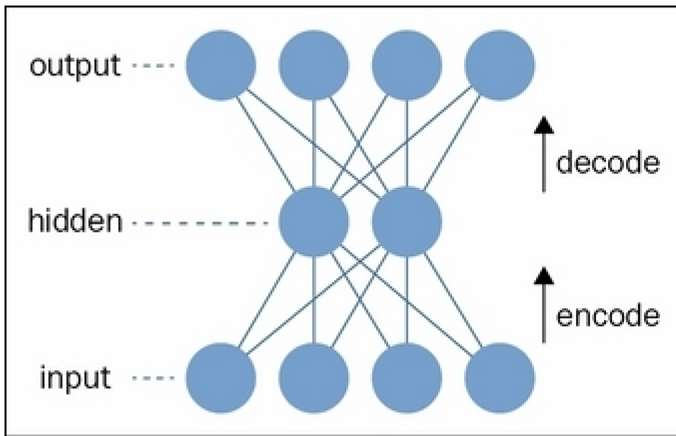
Simply put, autoencoders are a more developed and sophisticated set of techniques; the use of denoising and stacking techniques enable reductions of high-dimensional, multimodal data that can be trained with relative ease to greater accuracy, at greater scale, than the techniques that we discussed in Chapter 1, *Unsupervised Machine Learning*.

Having discussed the capabilities of autoencoders at a high level, let's dig in a little further to understand the topology of autoencoders as well as what their training involves.

Topology

As described earlier in this chapter, an autoencoder has a relatively simple structure. It is a three-layer neural network, with **input**, **hidden**, and **output** layers. The **input** feeds forward into the **hidden** layer, then the **output** layer, as with most neural network architectures. One topological feature worth mentioning is that the **hidden** layer is typically of fewer nodes than the **input** or **output** layers. (However, as intimated previously, the required number of **hidden** nodes is really a function of the complexity of the **input** data; the goal of the **hidden** layer is to bottleneck the information content from the **input** and force the network to identify a representation that captures underlying statistical properties. Representing very complex input accurately might require a large quantity of hidden nodes.)

The key feature of an autoencoder is that the **output** is typically set to be the **input**; the performance measure for an autoencoder is its accuracy in reconstructing the **input** after encoding it within the **hidden** layer. Autoencoder topology tends to take the following form:



The encoding function that occurs between the **input** and **hidden** layers is a mapping of an input (x) to a new form (y). A simple example mapping function might be a nonlinear (in this case sigmoid, s) function of the input as follows:

$$y = s(Wx + b)$$

However, more sophisticated encodings may exist or be developed to accommodate specific subject domains. In this case, of course, W represents the weight values assigned to x and b is an adjustable variable that can be tuned to enable the minimization of reconstruction error.

The autoencoder then decodes to deliver its output. This reconstruction is intended to take the same shape as x and will occur through a similar transformation as follows:

$$z = s\left(W'y + b'\right)$$

Here, b' and W' are typically also configurable to allow network optimization.

Training

The network trains, as discussed, by minimizing the reconstruction error. One popular method to measure this error is a simple squared error measure, as shown in the following formula:

$$E = \frac{1}{2}\left\|z - x\right\|^2$$

However, different and more appropriate error measures exist for cases where the input is in a less generic format (such as a set of bit probabilities).

While the intention is that autoencoders capture the main axes of variation in the input dataset, it is possible for an autoencoder to learn something far less useful—the identity function of the input.

Denoising autoencoders

While autoencoders can work well in some applications, they can be challenging to apply to problems where the input data contains a complex distribution that must be modeled in high dimensionality. The major challenge is that, with autoencoders that have **n-dimensional** input and an encoding of at least n, there is a real likelihood that the autoencoder will just learn the identity function of the input. In such cases, the encoding is a literal copy of the input. Such autoencoders are called **overcomplete**.

Note

One of the most important properties when training a machine learning technique is to understand how the dimensionality of hidden layers affects the quality of the resulting model. In cases where the input data is complex and the hidden layer has too few nodes to capture that complexity effectively, the result is obvious—the network fails to train as well as it might with more nodes.

To capture complex distributions in input data, then, you may wish to use a large number of hidden nodes. In cases where the hidden layer has at least as many nodes as the input, there is a strong

possibility that the network will learn the identity of the input; in such cases, each element of the input is learned as a specific unique case. Naturally, a model that has been trained to do this will work very well over training data, but as it has learned a trivial pattern that cannot be generalized to unfamiliar data, it is liable to fail catastrophically when validated.

This is particularly relevant when modeling complex data, such as speech data. Such data is frequently complex in distribution, so the classification of speech signals requires multimodal encoding and a high-dimensional hidden layer. Of course, this brings an increased risk of the autoencoder (or any of a large number of models as this is not an autoencoder-specific problem) learning the identity function.

While (rather surprisingly) overcomplete autoencoders can and do learn error-minimizing representations under certain configurations (namely, ones in which the first hidden layer needs very small weights so as to force the hidden units into a linear orientation and subsequent weights have large values), such configurations are difficult to optimize for, and it has been desirable to find another way to prevent overcomplete autoencoders from learning the identity function.

There are several different ways that an overcomplete autoencoder can be prevented from learning the identity function while still capturing something useful within its representation. By far, the most popular approach is to introduce noise to the input data and force the autoencoder to train on the noisy data by learning distributions and statistical regularities rather than identity. This can be effectively achieved by multiple methods, including using sparseness constraints or dropout techniques (wherein input values are randomly set to zero).

The process that we'll be using to introduce noise to the input in this chapter is dropout. Via this method, up to half of the inputs are randomly set to zero. To achieve this, we create a stochastic corruption process that operates on our input data:

```
def get_corrupted_input(self, input, corruption_level):

    return self.theano_rng.binomial(size=input.shape, n=1, p=1 -
    corruption_level, dtype=theano.config.floatX) * input
```

In order to accurately model the input data, the autoencoder has to predict the corrupted values from the uncorrupted values, thus learning meaningful statistical properties (that is, distribution).

In addition to preventing an autoencoder from learning the identity values of data, adding a denoising process also tends to produce models that are substantially more robust to input variations or distortion. This proves to be particularly useful for input data that is inherently noisy, such as speech or image data. One commonly recognized advantage of deep learning techniques, mentioned in the preface to this book, is that deep learning algorithms minimize the need for feature engineering. Where many learning algorithms require lengthy and complicated preprocessing of input data (filtering of images or manipulation of audio signals) to reconstruct the denoised input and enable the model to train, a dA can work effectively with minimal preprocessing. This can dramatically decrease the time it takes to train a model over your input data to practical levels of accuracy.

Finally, it's worth observing that an autoencoder that learns the identity function of the input dataset is probably misconfigured in a fundamental way. As the main added value of the autoencoder is to find a lower-dimensional representation of the feature set, an autoencoder that has learned the identity function

of the input data may simply have too many nodes. If in doubt, consider reducing the number of nodes in your hidden layer.

Now that we've discussed the topology of an autoencoder—the means by which one might be effectively trained and the role of denoising in improving autoencoder performance—let's review **Theano** code for a dA so as to carry the preceding theory into practice.

Applying a dA

At this point, we're ready to step through the implementation of a dA. Once again, we're leveraging the Theano library to apply a `dA` class.

Unlike the `RBM` class that we explored in the previous chapter, the DenoisingAutoencoder is relatively simple and tying the functionality of the dA to the theory and math that we examined earlier in this chapter is relatively simple.

In Chapter 2, *Deep Belief Networks*, we applied an `RBM` class that had a number of elements that, while not necessary for the correct functioning of the RBM in itself, enabled shared parameters within multilayer, deep architectures. The `dA` class we'll be using possesses similar shared elements that will provide us with the means to build a multilayer autoencoder architecture later in the chapter.

We begin by initializing a `dA` class. We specify the number of visible units, `n_visible`, as well as the number of hidden units, `n_hidden`. We additionally specify variables for the configuration of the input (`input`) as well as the weights (`W`) and the hidden and visible bias values (`bhid` and `bvis` respectively). The four additional variables enable autoencoders to receive configuration parameters from other elements of a deep architecture:

```
class dA(object):

    def __init__(
        self,
        numpy_rng,
        theano_rng=None,
        input=None,
        n_visible=784,
        n_hidden=500,
        W=None,
        bhid=None,
        bvis=None
):

        self.n_visible = n_visible
        self.n_hidden = n_hidden
```

We follow up by initialising the weight and bias variables. We set the weight vector, `W` to an initial value, `initial_W`, which we obtain using random, uniform sampling from the range:

$$-4*\sqrt{\frac{6.}{(n_hidden+n_visible)}} \quad to \quad 4*\sqrt{\frac{6.}{(n_{hidden}+n_{visible})}}.$$

We then set the visible and hidden bias variables to arrays of zeroes using `numpy.zeros`:

```
if not theano_rng:
   theano_rng = RandomStreams(numpy_rng.randint(2 ** 30))

if not W:
   initial_W = numpy.asarray(
      numpy_rng.uniform(
         low=-4 * numpy.sqrt(6. / (n_hidden + n_visible)),
         high=4 * numpy.sqrt(6. / (n_hidden + n_visible)),
         size=(n_visible, n_hidden)
      ),
      dtype=theano.config.floatX
   )
   W = theano.shared(value=initial_W, name='W', borrow=True)

if not bvis:
   bvis = theano.shared(
      value=numpy.zeros(
         n_visible,
         dtype=theano.config.floatX
      ),
      borrow=True
   )

if not bhid:
   bhid = theano.shared(
      value=numpy.zeros(
         n_hidden,
         dtype=theano.config.floatX
      ),
      name='b',
      borrow=True
   )
```

Earlier in the chapter, we described how the autoencoder translates between visible and hidden layers via mappings such as $y = s(Wx + b)$. To enable such translation, it is necessary to define `W`, `b`, `W'`, and

b' in relation to the previously described autoencoder parameters, bhid, bvis, and W. W' and b' are referred to as W_prime and b_prime in the following code:

```
self.W = W
self.b = bhid
self.b_prime = bvis
self.W_prime = self.W.T
self.theano_rng = theano_rng
if input is None:
    self.x = T.dmatrix(name='input')
else:
    self.x = input

self.params = [self.W, self.b, self.b_prime]
```

The preceding code sets b and b_prime to bhid and bvis respectively, while W_prime is set as the transpose of W; in other words, the weights are tied. Tied weights are sometimes, but not always, used in autoencoders for several reasons:

- Tying weights improves the quality of results in several contexts (albeit often in contexts where the optimal solution is PCA, which is the solution an autoencoder with tied weights will tend to reach)
- Tying weights improves the memory consumption of the autoencoder by reducing the number of parameters that need be stored
- Most importantly, tied weights provide a regularization effect; they require one less parameter to be optimized (thus one less thing that can go wrong!)

However, in other contexts, it's both common and appropriate to use untied weights. This is true, for instance, in cases where the input data is multimodal and the optimal decoder models a nonlinear set of statistical regularities. In such cases, a linear model, such as PCA, will not effectively model the nonlinear trends and you will tend to obtain better results using untied weights.

Having configured the parameters to our autoencoder, the next step is to define the functions that enable it to learn. Earlier in this chapter, we determined that autoencoders learn effectively by adding noise to input data, then attempting to learn an encoded representation of that input that can in turn be reconstructed into the input. What we need next, then, are functions that deliver this functionality. We begin by corrupting the input data:

```
def get_corrupted_input(self, input, corruption_level):

    return self.theano_rng.binomial(size=input.shape, n=1, p=1 -
    corruption_level, dtype=theano.config.floatX) * input
```

The degree of corruption is configurable using a corruption_level parameter; as we recognized earlier, the corruption of the input through dropout typically does not exceed 50% of cases, or 0.5. The function takes a random set of cases, where the number of cases is that proportion of the input whose size is equal to corruption_level. The function produces a corruption vector of *0's* and *1's* equal

in length to the input, where a `corruption_level` sized proportion of the vector is *0*. The corrupted input vector is then simply a multiple of the autoencoder's input vector and corruption vector:

```
def get_hidden_values(self, input):
   return T.nnet.sigmoid(T.dot(input, self.W) + self.b)
```

Next, we obtain the hidden values. This is done via code that performs the equation $y = s(Wx + b)$ to obtain *y* (the hidden values). To get the autoencoder's output (*z*), we reconstruct the hidden layer via code that uses the previously defined `b_prime` and `W_prime` to perform $z = s(W'y + b')$:

```
defget_reconstructed_input(self, hidden):
   returnT.nnet.sigmoid(T.dot(hidden, self.W_prime) +
   self.b_prime)
```

The final missing piece is the calculation of cost updates. We reviewed one cost function previously, a simple squared error measure: $E = \frac{1}{2}\|z - x\|^2$. Let's use this cost function to calculate our cost updates, based on the input (`x`) and reconstruction (`z`):

```
def get_cost_updates(self, corruption_level, learning_rate):

   tilde_x = self.get_corrupted_input(self.x, corruption_level)
   y = self.get_hidden_values(tilde_x)
   z = self.get_reconstructed_input(y)
   E = (0.5 * (T.z – T.self.x)) ^ 2
   cost = T.mean(E)

   gparams = T.grad(cost, self.params)
   updates = [
      (param, param - learning_rate * gparam)
      for param, gparam in zip(self.params, gparams)
   ]

return (cost, updates)
```

At this point, we have a functional dA! It may be used to model nonlinear properties of input data and can work as an effective tool to learn valid and lower-dimensional representations of input data. However, the real power of autoencoders comes from the properties that they display when stacked together, as the building blocks of a deep architecture.

Stacked Denoising Autoencoders

While autoencoders are valuable tools in themselves, significant accuracy can be obtained by stacking autoencoders to form a deep network. This is achieved by feeding the representation created by the encoder on one layer into the next layer's encoder as the input to that layer.

Stacked denoising autoencoders (**SdAs**) are currently in use in many leading data science teams for sophisticated natural language analyses as well as a hugely broad range of signals, image, and text analysis.

The implementation of a SdA will be very familiar after the previous chapter's discussion of deep belief networks. The SdA is used in much the same way as the RBMs in our deep belief networks were used. Each layer of the deep architecture will have a dA and sigmoid component, with the autoencoder component being used to pretrain the sigmoid network. The performance measure used by a stacked denoising autoencoder is the training set error, with an intensive period of layer-to-layer (layer-wise) pretraining used to gradually align network parameters before a final period of fine-tuning. During fine-tuning, the network is trained using validation and test data, over fewer epochs but with larger update steps. The goal is to have the network converge at the end of the fine-tuning in order to deliver an accurate result.

In addition to delivering on the typical advantages of deep networks (the ability to learn feature representations for complex or high-dimensional datasets, and the ability to train a model without extensive feature engineering), stacked autoencoders have an additional, interesting property.

Correctly configured stacked autoencoders can capture a **hierarchical grouping** of their input data. Successive layers of a stacked denoised autoencoder may learn increasingly high-level features. Where the first layer might learn some first-order features from input data (such as learning edges in a photo image), a second layer may learn some grouping of first-order features (for instance, by learning given configurations of edges that correspond to contours or structural elements in the input image).

There's no golden rule to determine how many layers or how large layers should be for a given problem. The best solution is usually to experiment with these model parameters until you find an optimal point. This experimentation is best done with a hyperparameter optimization technique or genetic algorithm (subjects we'll discuss in later chapters of this book).

Higher layers may learn increasingly high-order configurations, enabling a stacked denoised autoencoder to learn to recognize facial features, alphanumerical characters, or generalized forms of objects (such as a bird). This is what gives SdAs their unique capability to learn very sophisticated, high-level abstractions of their input data.

Autoencoders can be stacked indefinitely, and it has been demonstrated that continuing to stack autoencoders can improve the effectiveness of the deep architecture (with the main constraint becoming compute cost in time). In this chapter, we'll look at stacking three autoencoders to solve a natural language processing challenge.

Applying the SdA

Now that we've had a chance to understand the advantages and power of the SdA as a deep learning architecture, let's test our skills on a real-world dataset.

For this chapter, let's step away from image datasets and work with the **OpinRank Review** dataset, a text dataset of around 259,000 hotel reviews from TripAdvisor—accessible via the UCI machine learning dataset repository. This freely-available dataset provides review scores (as floating point numbers from 1 to 5) and review text for a broad range of hotels; we'll be applying our stacked dA to attempt to identify the scoring of each hotel from its review text.

Note

We'll be applying our autoencoder to analyze a preprocessed version of this data, which is accessible from the GitHub share accompanying this chapter. We'll be discussing the techniques by which we prepare text data in an upcoming chapter. For the interested reader, the source data is available at https://archive.ics.uci.edu/ml/datasets/OpinRank+Review+Dataset.

In order to get started, we're going to need a stacked denoising autoencoder (hereafter `SdA`) class:

```
class SdA(object):


    def __init__(
        self,
        numpy_rng,
        theano_rng=None,
        n_ins=280,
        hidden_layers_sizes=[500, 500],
        n_outs=5,
        corruption_levels=[0.1, 0.1]
):
```

As we previously discussed, the `SdA` is created by feeding the encoding from one layer's autoencoder as the input to the subsequent layer. This class supports the configuration of the layer count (reflected in, but not set by, the length of the `hidden_layers_sizes` and `corruption_levels` vectors). It also supports differentiated layer sizes (in nodes) at each layer, which can be set using `hidden_layers_sizes`. As we discussed, the ability to configure successive layers of the autoencoder is critical to developing successful representations.

Next, we need parameters to store the MLP (`self.sigmoid_layers`) and dA (`self.dA_layers`) elements of the `SdA`. In order to specify the depth of our architecture, we use the `self.n_layers` parameter to specify the number of sigmoid and dA layers required:

```
self.sigmoid_layers = []
self.dA_layers = []
self.params = []
```

```
self.n_layers = len(hidden_layers_sizes)

assertself.n_layers> 0
```

Next, we need to construct our sigmoid and dA layers. We begin by setting the hidden layer size to be set either from the input vector size or by the activation of the preceding layer. Following this, `sigmoid_layer` and `dA_layer` components are created, with the dA layer drawing from the `dA` class that we discussed earlier in this chapter:

```
for i in xrange(self.n_layers):
   if i == 0:
      input_size = n_ins
   else:
      input_size = hidden_layers_sizes[i - 1]

if i == 0:
   layer_input = self.x
else:
   layer_input = self.sigmoid_layers[-1].output

sigmoid_layer = HiddenLayer(rng=numpy_rng, input=layer_input,
n_in=input_size, n_out=hidden_layers_sizes[i],
activation=T.nnet.sigmoid)

self.sigmoid_layers.append(sigmoid_layer)
self.params.extend(sigmoid_layer.params)

dA_layer = dA(numpy_rng=numpy_rng, theano_rng=theano_rng,
input=layer_input, n_visible=input_size,
n_hidden=hidden_layers_sizes[i], W=sigmoid_layer.W,
bhid=sigmoid_layer.b)

self.dA_layers.append(dA_layer)
```

Having implemented the layers of our stacked dA, we'll need a final, logistic regression layer to complete the MLP component of the network:

```
self.logLayer = LogisticRegression(
   input=self.sigmoid_layers[-1].output,
   n_in=hidden_layers_sizes[-1],
   n_out=n_outs
)

self.params.extend(self.logLayer.params)
self.finetune_cost = self.logLayer.negative_log_likelihood(self.y)
self.errors = self.logLayer.errors(self.y)
```

This completes the architecture of our SdA. Next up, we need to generate the training functions used by the `SdA` class. Each function will the minibatch index (`index`) as an argument, together with several other elements—the `corruption_level` and `learning_rate` are enabled here so that we can adjust them (for example, gradually increase or decrease them) during training. Additionally, we identify variables that help identify where the batch starts and ends—`batch_begin` and `batch_end`, respectively:

Note

The ability to dynamically adjust the learning rate is particularly very helpful and may be applied in one of two ways. Once a technique has begun to converge on an appropriate solution, it is very helpful to be able to reduce the learning rate. If you do not do this, you risk creating a situation in which the network oscillates between values located around the optimum without ever hitting it. In some contexts, it can be helpful to tie the learning rate to the network's performance measure. If the error rate is high, it makes sense to make larger adjustments until the error rate begins to decrease!

```
def pretraining_functions(self, train_set_x, batch_size):
    index = T.lscalar('index')
    corruption_level = T.scalar('corruption')
    learning_rate = T.scalar('lr')
    batch_begin = index * batch_size
    batch_end = batch_begin + batch_size

    pretrain_fns = []
    for dA in self.dA_layers:
        cost, updates = dA.get_cost_updates(corruption_level,
learning_rate)
        fn = theano.function(
            inputs=[
                index,
                theano.Param(corruption_level, default=0.2),
                theano.Param(learning_rate, default=0.1)
            ],
            outputs=cost,
            updates=updates,
            givens={
                self.x: train_set_x[batch_begin: batch_end]
            }
         )
         pretrain_fns.append(fn)

    return pretrain_fns
```

The pretraining functions that we've created takes the minibatch `index` and can optionally take the corruption level or learning rate. It performs one step of pretraining and outputs the cost value and vector of weight updates.

In addition to pretraining, we need to build functions to support the fine-tuning stage, wherein the network is run iteratively over the validation and test data to optimize network parameters. The training function (`train_fn`) seen in the code below implements a single step of fine-tuning. The `valid_score` is a Python function that computes a validation score using the error measure produced by the `SdA` over validation data. Similarly, `test_score` computes the error score over test data.

To get this process off the ground, we first need to set up training, validation, and test datasets. Each stage requires two datasets (set `x` and set `y`) containing the features and class labels, respectively. The required number of minibatches for validation and test is determined, and an index is created to track the batch size (and provide a means of identifying at which entries a batch starts and ends). Training, validation, and testing occurs for each batch and afterward, both `valid_score` and `test_score` are calculated across all batches:

```
def build_finetune_functions(self, datasets,
batch_size,learning_rate):

    (train_set_x, train_set_y) = datasets[0]
    (valid_set_x, valid_set_y) = datasets[1]
    (test_set_x, test_set_y) = datasets[2]

   n_valid_batches = valid_set_x.get_value(borrow=True).shape[0]
   n_valid_batches /= batch_size
   n_test_batches = test_set_x.get_value(borrow=True).shape[0]
   n_test_batches /= batch_size

   index = T.lscalar('index')


   gparams = T.grad(self.finetune_cost, self.params)


   updates = [
        (param, param - gparam * learning_rate)
       For param, gparam in zip(self.params, gparams)
]

train_fn = theano.function(
   inputs=[index],
   outputs=self.finetune_cost,
   updates=updates,
   givens={
      self.x: train_set_x[
         index * batch_size: (index + 1) * batch_size
      ],
      self.y: train_set_y[
         index * batch_size: (index + 1) * batch_size
      ]
```

```
    },
    name='train'
)

test_score_i = theano.function(
     [index],
    self.errors,
    givens={
        self.x: test_set_x[
        index * batch_size: (index + 1) * batch_size
    ],
        self.y: test_set_y[
        index * batch_size: (index + 1) * batch_size
    ]
},
    name='test'
)

valid_score_i = theano.function(
    [index],
    self.errors,
    givens={
        self.x: valid_set_x[
            index * batch_size: (index + 1) * batch_size
        ],
        self.y: valid_set_y[
            index * batch_size: (index + 1) * batch_size
        ]
    },
    name='valid'
)


def valid_score():
    return [valid_score_i(i) for i inxrange(n_valid_batches)]

def test_score():
    return [test_score_i(i) for i inxrange(n_test_batches)]

return train_fn, valid_score, test_score
```

With the training functionality in place, the following code initiates our stacked dA:

```
numpy_rng = numpy.random.RandomState(89677)
print '... building the model'
    sda = SdA(
        numpy_rng=numpy_rng,
```

```
        n_ins=280,
        hidden_layers_sizes=[240, 170, 100],
        n_outs=5
    )
```

It should be noted that, at this point, we should be trying an initial configuration of layer sizes to see how we do. In this case, the layer sizes used are the product of some initial testing. As we discussed, training the `SdA` occurs in two stages. The first is a layer-wise pretraining process that loops over all of the `SdA's` layers. The second is a process of fine-tuning over validation and test data.

To pretrain the `SdA`, we provide the required corruption levels to train each layer and iterate over the layers using our previously defined `pretraining_fns`:

```
print '... getting the pretraining functions'
pretraining_fns = sda.pretraining_functions(train_set_x=train_set_x,
batch_size=batch_size)

print '... pre-training the model'
start_time = time.clock()
corruption_levels = [.1, .2, .2]
for i in xrange(sda.n_layers):

    for epoch in xrange(pretraining_epochs):
        c = []
        for batch_index in xrange(n_train_batches):
            c.append(pretraining_fns[i](index=batch_index,
            corruption=corruption_levels[i],
            lr=pretrain_lr))
print 'Pre-training layer %i, epoch %d, cost ' % (i, epoch),

print numpy.mean(c)

end_time = time.clock()

print(('The pretraining code for file ' +
os.path.split(__file__)[1] + ' ran for %.2fm' % ((end_time -
start_time) / 60.)), file = sys.stderr)
```

At this point, we're able to initialize our `SdA` class via calling the preceding code stored within this book's GitHub repository: `MasteringMLWithPython/Chapter3/SdA.py`

Assessing SdA performance

The SdA will take a significant length of time to run. With 15 epochs per layer and each layer typically taking an average of 11 minutes, the network will run for around 500 minutes on a modern desktop system with GPU acceleration and a single-threaded GotoBLAS.

On a system without GPU acceleration, the network will take substantially longer to train, and it is recommended that you use the alternative, which runs over a significantly smaller input dataset: `MasteringMLWithPython/Chapter3/SdA_no_blas.py`

The results are of high quality, with a validation error score of 3.22% and test error score of 3.14%. These results are particularly impressive given the ambiguous and sometimes challenging nature of natural language processing applications.

It was noticeable that the network classified more correctly for the 1-star and 5-star rating cases than for the intermediate levels. This is largely due to the ambiguous nature of unpolarized or unemotional language.

Part of the reason that this input data was classifiable was via significant feature engineering. While time-consuming and sometimes problematic, we've seen that well-executed feature engineering combined with an optimized model can deliver an excellent level of accuracy. In Chapter 6, *Text Feature Engineering*, we'll be applying the techniques used to prepare this dataset ourselves.

Further reading

A well-informed overview of autoencoders (amongst other subjects) is provided by Quoc V. Le from the Google Brain team. Read about it at https://cs.stanford.edu/~quocle/tutorial2.pdf.

This chapter used the Theano documentation available at http://deeplearning.net/tutorial/contents.html as a base for discussion as Theano was the main library used in this chapter.

Summary

In this chapter, we introduced the autoencoder, an effective dimensionality reduction technique with some unique applications. We focused on the theory behind the stacked denoised autoencoder, an extension of autoencoders whereby any number of autoencoders are stacked in a deep architecture. We were able to apply the stacked denoised autoencoder to a challenging natural language processing problem and met with great success, delivering highly accurate sentiment analysis of hotel reviews.

In the next chapter, we will discuss supervised deep learning methods, including **Convolutional Neural Networks** (**CNN**).

Chapter 4. Convolutional Neural Networks

In this chapter, you'll be learning how to apply the convolutional neural network (also referred to as the CNN or convnet), perhaps the best-known deep architecture, via the following steps:

- Taking a look at the convnet's topology and learning processes, including convolutional and pooling layers
- Understanding how we can combine convnet components into successful network architectures
- Using Python code to apply a convnet architecture so as to solve a well-known image classification task

Introducing the CNN

In the field of machine learning, there is an enduring preference for developing structures in code that parallel biological structures. One of the most obvious examples is that of the MLP neural network, whose topology and learning processes are inspired by the neurons of the human brain.

This preference has turned out to be highly efficient; the availability of specialized, optimized biological structures that excel at specific sets of tasks gives us a wealth of templates and clues from which to design and create effective learning models.

The design of convolutional neural networks takes inspiration from the visual cortex—the area of the brain that processes visual input. The visual cortex has several specializations that enable it to effectively process visual data; it contains many receptor cells that detect light in overlapping regions of the visual field. All receptor cells are subject to the same convolution operation, which is to say that they all process their input in the same way. These specializations were incorporated into the design of convnets, making their topology noticeably distinct from that of other neural networks.

It's safe to say that CNN (convnets for short) are underpinning many of the most impactful current advances in artificial intelligence and machine learning. Variants of CNN are applied to some of the most sophisticated visual, linguistic, and problem-solving applications in existence. Some examples include the following:

- Google has developed a range of specialized convnet architectures, including **GoogLeNet**, a 22-layer convnet architecture. In addition, Google's DeepDream program, which became well-known for its overtrained, hallucinogenic imagery, also uses a convolutional neural network.
- Convolutional nets have been taught to play the game **Go** (a long-standing AI challenge), achieving win-rates ranging between 85% and 91% against highly-ranked players.
- Facebook uses convolutional nets in face verification (**DeepFace**).
- Baidu, Microsoft research, IBM, and Twitter are among the many other teams using convnets to tackle the challenges around trying to deliver next-generation intelligent applications.

In recent years, object recognition challenges, such as the 2014 **ImageNet** challenge, have been dominated by winners employing specialized convnet implementations or multiple-model ensembles that combine convnets with other architectures.

While we'll cover how to create and effectively apply ensembles in Chapter 8, *Ensemble Methods*, this chapter focuses on the successful application of convolutional neural networks to large-scale visual classification contexts.

Understanding the convnet topology

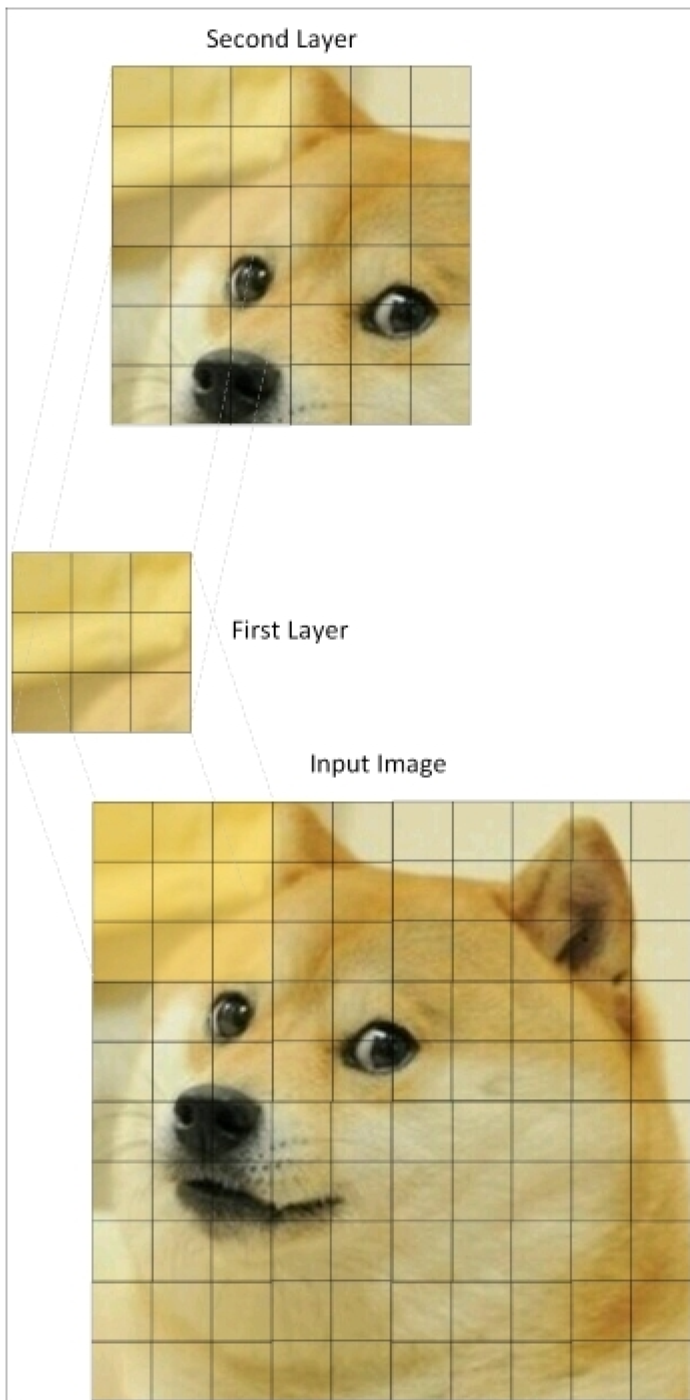
The convolutional neural network's architecture should be fairly familiar; the network is an acyclic graph composed of layers of increasingly few nodes, where each layer feeds into the next. This will be very familiar from many well-known network topologies such as the MLP.

Perhaps the most immediate difference between a convolutional neural network and most other networks is that all of the neurons in a convnet are identical! All neurons possess the same parameters and weight values. As you can see, this will immediately reduce the number of parameter values controlled by the network, bringing substantial efficiency savings. It also typically improves network learning rate as there are fewer free parameters to be managed and computed over. As we'll see later in this chapter, shared weights also enable a convnet to learn features irrespective of their position in the input (for example, the input image or audio signal).

Another big difference between convolutional networks and other architectures is that the connectivity between nodes is limited such as to develop a spatially local connectivity pattern. In other words, the inputs to a given node will be limited to only those nodes whose receptor fields are contiguous. This may be spatially contiguous, as in the case of image data; in such cases, each neuron's inputs will ultimately draw from a continuous subset of the image. In the case of audio signal data, the input might instead be a continuous window of time.

To illustrate this more clearly, let's take an example input image and discuss how a convolutional network might process parts of that image across specific nodes. Nodes in the first layer of a convolutional neural network will be assigned subsets of the input image. In this case, let's say that they take a 3 x 3 pixel subset of the image each. Our coverage covers the entire image without any overlap between the areas taken as input by nodes and without any gaps. (Note that none of these conditions are automatically true for convnet implementations.) Each node is assigned a 3 x 3 pixel subset of the image (the receptive field of the node) and outputs a transformed version of that input. We'll disregard the specifics of that transformation for now.

This output is usually then picked up by a second layer of nodes. In this case, let's say that our second layer is taking a subset of all of the outputs from nodes in the first layer. For example, it might be taking a contiguous 6 x 6 pixel subset of the original image; that is, it has a receptive field that covers the outputs of exactly four nodes from the preceding layer. This becomes a little more intuitive when explained visually:



Each layer is **composable**; the output of one convolutional layer may be fed into the next layer as an input. This provides the same effect that we saw in the Chapter 3, *Stacked Denoising Autoencoders*; successive layers develop representations of increasingly high-level, abstract features. Furthermore, as we build downward—adding layers—the representation becomes responsive to a larger region of pixel space. Ultimately, by stacking layers, we can work our way toward global representations of the entire input.

Understanding convolution layers

As described, in order to prevent each node from learning an unpredictable (and difficult to tune!) set of very local, free parameters, weights in a layer are shared across the entire layer. To be completely precise, the filters applied in a convolutional layer are a single set of filters, which are slid (convolved) across the input dataset. This produces a two-dimensional activation map of the input, which is referred to as the feature map.

The filter itself is subject to four hyperparameters: size, depth, stride, and zero-padding. The size of the filter is fairly self-explanatory, being the area of the filter (obviously, found by multiplying height and width; a filter need not be square!). Larger filters will tend to overlap more, and as we'll see, this can improve the accuracy of classification. Crucially, however, increasing the filter size will create increasingly large outputs. As we'll see, managing the size of outputs from convolutional layers is a huge factor in controlling the efficiency of a network.

Depth defines the number of nodes in the layer that connect to the same region of the input. The trick to understanding depth is to recognize that looking at an image (for people or networks) involves processing multiple different types of property. Anyone who has ever looked at all the image adjustment sliders in Photoshop has an idea of what this might entail. Depth is sometimes referred to as a dimension in its own right; it almost relates to the complexity of an image, not in terms of its contents but in terms of the number of channels needed to accurately describe it.

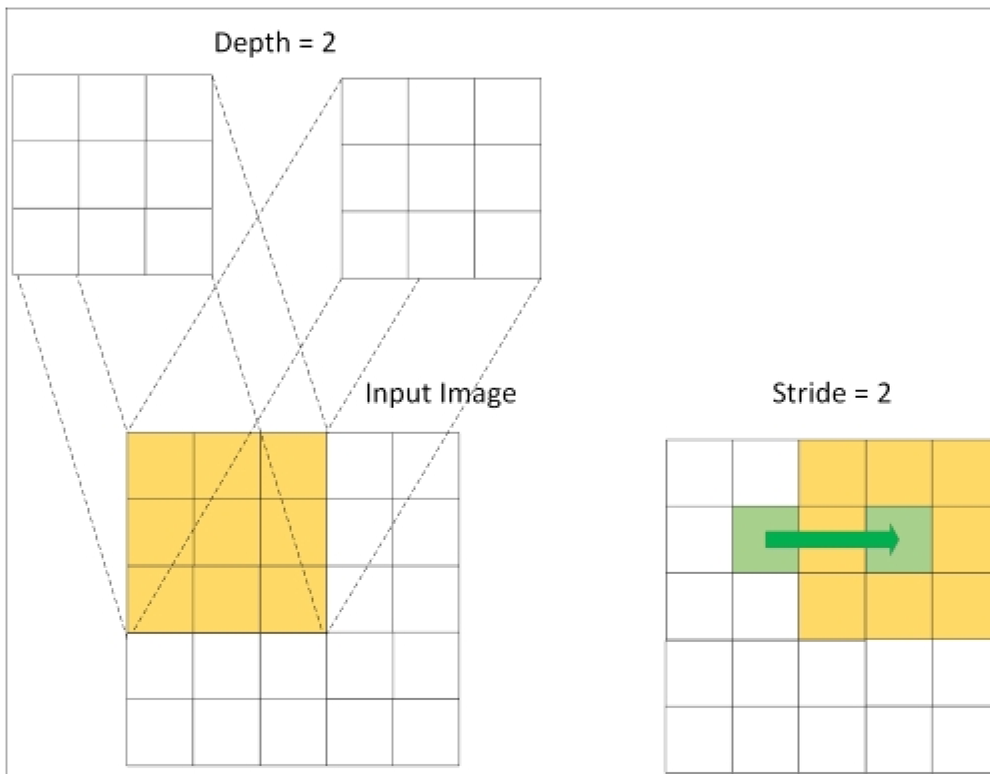
It's possible that the depth might describe color channels, with nodes mapped to recognize green, blue, or red in the input. This, incidentally, leads to a common convention where depth is set to three (particularly at the first convolution layer). It's very important to recognize that some nodes commonly learn to express less easily-described properties of input images that happen to enable a convnet to learn that image more accurately. Increasing the depth hyperparameter tends to enable nodes to encode more information about inputs, with the attendant problems and benefits that you might expect.

As a result, setting the depth parameter to too small a value tends to lead to poor results because the network doesn't have the expressive depth (in terms of channel count) required to accurately characterize input data. This is a problem analogous to not having enough features, except that it's more easily fixed; one can tune the depth of the network upward to improve the expressive depth of the convnet.

Equally, setting the depth parameter to too small a value can be redundant or harmful to performance, thereafter. If in doubt, consider testing the appropriate depth value during network configuration via hyperparameter optimization, the elbow method, or another technique.

Stride is a measure of spacing between neurons. A stride value of one will lead every element of the input (for an image, potentially every pixel) to be the center of a filter instance. This naturally leads to a high degree of overlap and very large outputs. Increasing the stride causes less of an overlap in the receptive fields and the output's size is reduced. While tuning the stride of a convnet is a question of weighing accuracy against output size, it can generally be a good idea to use smaller strides, which tend to work better. In addition, a stride value of one enables us to manage down-sampling and scale reduction at pooling layers (as we'll discuss later in the chapter).

The following diagram graphically displays both **Depth** and **Stride**:



The final hyperparameter, zero-padding, offers an interesting convenience. Zero-padding is the process of setting the outer values (the border) of each receptive field to zero, which has the effect of reducing the output size for that layer. It's possible to set one, or multiple, pixels around the border of the field to zero, which reduces the output size accordingly. There are, of course, limits; obviously, it's not a good idea to set zero-padding and stride such that areas of the input are not touched by a filter! More generally, increasing the degree of zero-padding can cause a decrease in effectiveness, which is tied to the increased difficulty of learning features via coarse coding. (Refer to the *Understanding pooling layers* section in this chapter.)

However, zero-padding is very helpful because it enables us to adjust the input and output sizes to be the same. This is a very common practice; using zero-padding to ensure that the size of the input layer and output layer are equal, we are able to easily manage the stride and depth values. Without using zero-padding in this way, we would need to do a lot of work tracking input sizes and managing network parameters simply to make the network function correctly. In addition, zero-padding also improves performance as, without it, a convnet will tend to gradually degrade content at the edges of the filter.

In order to calibrate the number of nodes, appropriate stride, and padding for successive layers when we define our convnet, we need to know the size of the output from the preceding layer. We can calculate the spatial size of a layer's output (O) as a function of the input image size (W), filter size (F), stride (S), and the amount of zero-padding applied (P), as follows:

$$O = \frac{W - F + 2P}{S + 1}$$

If O is not an integer, the filters do not tile across the input neatly and instead extend over the edge of the input. This can cause some problematic issues when training (normally involving thrown exceptions)! By adjusting the stride value, one can find a whole-number solution for O and train effectively. It is normal for the stride to be constrained to what is possible given the other hyperparameter values and size of the input.

We've discussed the hyperparameters involved in correctly configuring the convolutional layer, but we haven't yet discussed the convolution process itself. Convolution is a mathematical operator, like addition or derivation, which is heavily used in signal processing applications and in many other contexts where its application helps simplify complex equations.

Loosely speaking, convolution is an operation over two functions, such as to produce a third function that is a modified version of one of the two original functions. In the case of convolution within a convnet, the first component is the network's input. In the case of convolution applied to images, convolution is applied in two dimensions (the width and height of the image). The input image is typically three matrices of pixels—one for each of the red, blue, and green color channels, with values ranging between *0* and *255* in each channel.

Note

At this point, it's worth introducing the concept of a **tensor**. Tensor is a term commonly used to refer to an n-dimensional array or matrix of input data, commonly applied in deep learning contexts. It's effectively analogous to a matrix or array. We'll be discussing tensors in more detail, both in this chapter and in Chapter 9, *Additional Python Machine Learning Tools* (where we review the **TensorFlow** library). It's worth noting that the term tensor is noticing a resurgence of use in the machine learning community, largely through the influence of Google machine intelligence research teams.

The second input to the convolution operation is the convolution kernel, a single matrix of floating point numbers that acts as a filter on the input matrices. The output of this convolution operation is the feature map. The convolution operation works by sliding the filter across the input, computing the dot product of the two arguments at each instance, which is written to the feature map. In cases where the stride of the convolutional layer is one, this operation will be performed across each pixel of the input image.

The main advantage of convolution is that it reduces the need for feature engineering. Creating and managing complex kernels and performing the highly specialized feature engineering processes needed is a demanding task, made more challenging by the fact that feature engineering processes that work well in one context can work poorly in most others. While we discuss feature engineering in detail in Chapter 7, *Feature Engineering Part II*, convolutional nets offer a powerful alternative.

CNN, however, incrementally improve their kernel's ability to filter a given input, thus automatically optimizing their kernel. This process is accelerated by learning multiple kernels in parallel at once. This

is feature learning, which we've encountered in previous chapters. Feature learning can offer tremendous advantages in time and in increasing the accessibility of many problems. As with our earlier SDA and DBN implementations, we would look to pass our learned features to a much simpler, shallow neural network, which uses these features to classify the input image.

Understanding pooling layers

Stacking convolutional layers allows us to create a topology that effectively creates features as feature maps for complex, noisy input data. However, convolutional layers are not the only component of a deep network. It is common to weave convolutional layers in with pooling layers. Pooling is an operation over feature maps, where multiple feature values are aggregated into a single value—mostly using a max (**max-pooling**), mean (**mean-pooling**), or summation (**sum-pooling**) operation.

Pooling is a fairly natural approach that offers substantial advantages. If we do not aggregate feature maps, we tend to find ourselves with a huge amount of features. The **CIFAR-10** dataset that we'll be classifying later in this chapter contains 60,000 32 x 32 pixel images. If we hypothetically learned *200* features for each image—over 8 x 8 inputs—then at each convolution, we'd find ourselves with an output vector of size *(32 – 8+1) * (32 – 8+1) * 200*, or *125,000* features per image. Convolution produces a huge amount of features that tend to make computation very expensive and can also introduce significant overfitting problems.

The other major advantage provided by a pooling operation is that it provides a level of robustness against the many, small deviations and variances that occur in modeling noisy, high-dimensional data. Specifically, pooling prevents the network learning the position of features too specifically (overfitting), which is obviously a critical requirement in image processing and recognition settings. With pooling, the network no longer fixates on the precise location of features in the input and gains a greater ability to generalize. This is called **translation-invariance**.

Max-pooling is the most commonly applied pooling operation. This is because it focuses on the most responsive features in question that should, in theory, make it the best candidate for image recognition and classification purposes. By a similar logic, min-pooling tends to be applied in cases where it is necessary to take additional steps to prevent an overly sensitive classification or overfitting from occurring.

For obvious reasons, it's prudent to begin modeling using a quickly applied and straightforward pooling method such as max-pooling. However, when seeking additional gains in network performance during later iterations, it's important to look at whether your pooling operations can be improved on. There isn't any real restriction in terms of defining your own pooling operation. Indeed, finding a more effective subsampling method or alternative aggregation can substantially improve the performance of your model.

In terms of `theano` code, a max-pooling implementation is pretty straightforward and may look like this:

```
from theano.tensor.signal import downsample

input = T.dtensor4('input')
maxpool_shape = (2, 2)
```

```
pool_out = downsample.max_pool_2d(input, maxpool_shape,
ignore_border=True)
f = theano.function([input],pool_out)
```

The `max_pool_2d` function takes an n-dimensional `tensor` and downscaling factor, in this case, `input` and `maxpool_shape`, with the latter being a tuple of length `2`, containing width and height downscaling factors for the input image. The `max_pool_2d` operation then performs max-pooling over the two trailing dimensions of the vector:

```
invals = numpy.random.RandomState(1).rand(3, 2, 5, 5)

pool_out = downsample.max_pool_2d(input, maxpool_shape,
ignore_border=False)
f = theano.function([input],pool_out)
```

The `ignore_border` determines whether the border values are considered or discarded. This max-pooling operation produces the following, given that `ignore_border = True`:

```
[[ 0.72032449  0.39676747]
[ 0.6852195   0.87811744]]
```

As you can see, pooling is a straightforward operation that can provide dramatic results (in this case, the input was a 5 x 5 matrix, reduced to 2 x 2). However, pooling is not without critics. In particular, *Geoffrey Hinton* offered this pretty delightful soundbite:

> *"The pooling operation used in convolutional neural networks is a big mistake and the fact that it works so well is a disaster.*
>
> *If the pools do not overlap, pooling loses valuable information about where things are. We need this information to detect precise relationships between the parts of an object. Its true that if the pools overlap enough, the positions of features will be accurately preserved by "coarse coding" (see my paper on "distributed representations" in 1986 for an explanation of this effect). But I no longer believe that coarse coding is the best way to represent the poses of objects relative to the viewer (by pose I mean position, orientation, and scale)."*

This is a bold statement, but it makes sense. Hinton's telling us that the pooling operation, as an aggregation, does what any aggregation necessarily does—it reduces the data to a simpler and less informationally-rich format. This wouldn't be too damaging, except that Hinton goes further.

Even if we'd reduced the data down to single values for each pool, we could still hope that the fact that multiple pools overlap spatially would still present feature encodings. (This is the coarse coding referred to by Hinton.) This is also quite an intuitive concept. Imagine that you're listening in to a signal on a noisy radio frequency. Even if you only caught one word in three, it's probable that you'd be able to distinguish a distress signal from the shipping forecast!

However, Hinton follows up by observing that coarse coding is not as effective in learning pose (position, orientation, and scale). There are so many permutations in viewpoint relative to an object that

it's unlikely two images would be alike and the sheer variety of possible poses becomes a challenge for a convolutional network using pooling. This suggests that an architecture that does not overcome this challenge may not be able to break past an upper limit for image classification.

However, the general consensus, at least for now, is that even after acknowledging all of this, it is still highly advantageous in terms of efficiency and translation-invariance to continue using pooling operations in convnets. Right now, the argument goes that it's the best we have!

Meanwhile, Hinton proposed an alternative to convnets in the form of the **transforming autoencoder**. The transforming autoencoder offers accuracy improvements on learning tasks that require a high level of precision (such as facial recognition), where pooling operations would cause a reduction in precision. The *Further reading* section of this chapter contains recommendations if you are interested in learning more about the transforming autoencoder.

So, we've spent quite a bit of time digging into the convolutional neural network—its components, how they work, and their hyperparameters. Before we move on to put the theory into action, it's worth discussing how all of these theoretical components fit together into a working architecture. To do this, let's discuss what training a convnet looks like.

Training a convnet

The means of training a convolutional network will be familiar to readers of the preceding chapters. The convolutional architecture itself is used to pretrain a simpler network structure (for example, an MLP). The backpropagation algorithm is the standard method to compute the gradient when pretraining. During this process, every layer undertakes three tasks:

- **Forward pass**: Each feature map is computed as a sum of all feature maps convolved with the corresponding weight kernel
- **Backward pass**: The gradients respective to inputs are calculated by convolving the transposed weight kernel with the gradients, with respect to the outputs
- The loss for each kernel is calculated, enabling the individual weight adjustment of every kernel as needed

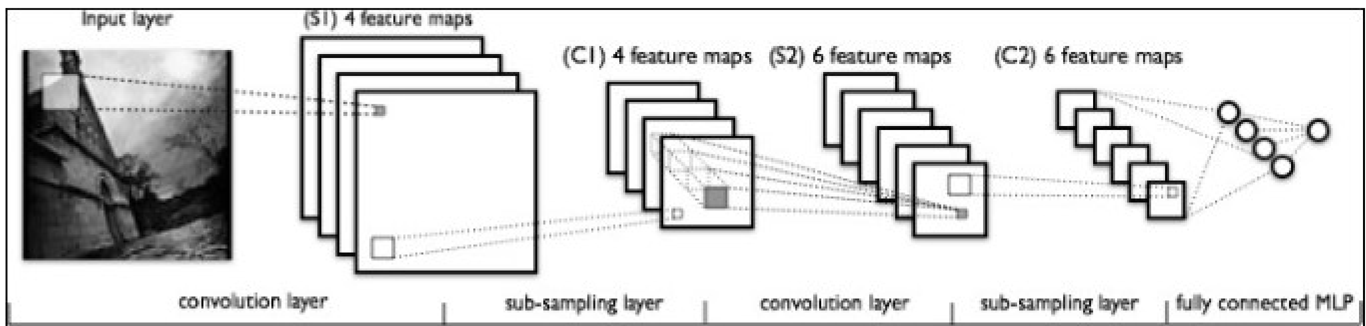
Repetition of this process allows us to achieve increasing kernel performance until we reach a point of convergence. At this point, we will hope to have developed a set of features sufficient that the capping network is able to effectively classify over these features.

This process can execute slowly, even on a fairly advanced GPU. Some recent developments have helped accelerate the training process, including the use of the Fast **Fourier Transform** to accelerate the convolution process (for cases where the convolution kernel is of roughly equal size to the input image).

Putting it all together

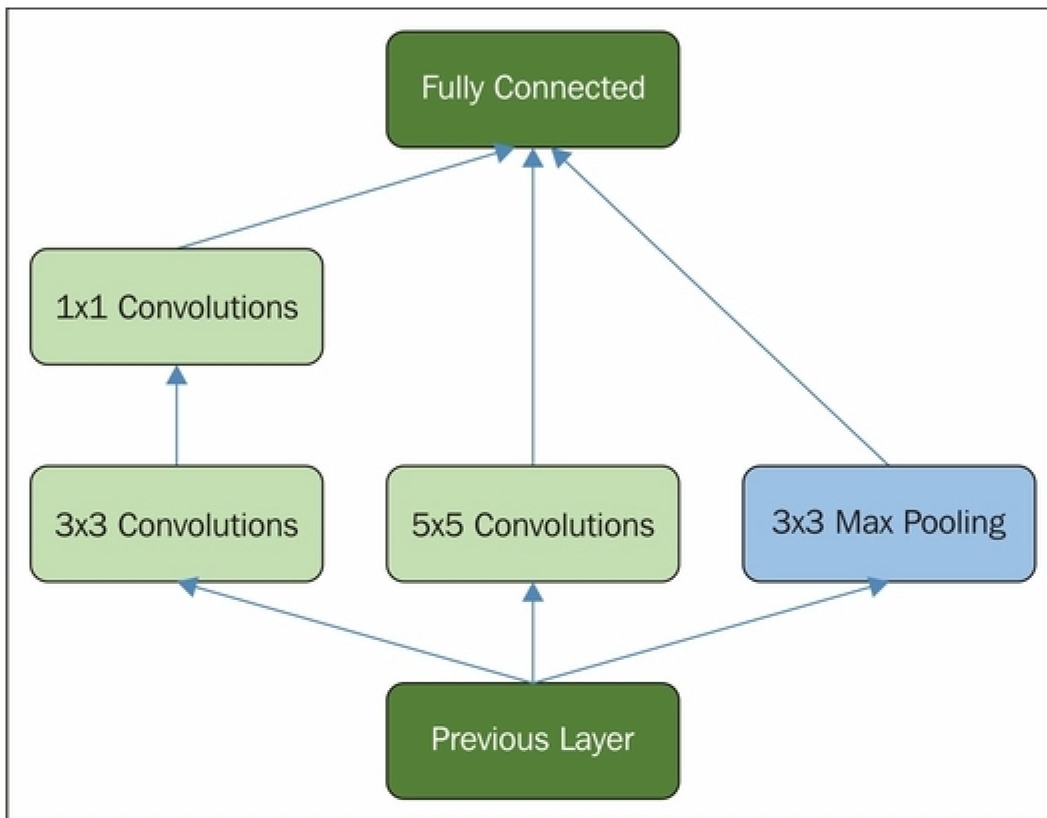
So far, we've discussed some of the elements required to create a CNN. The next subject of discussion should be how we go about combining these components to create capable convolutional nets as well as which combinations of components can work well. We'll draw guidance from a number of forerunning convnet implementations as we build an understanding of what is commonly done as well as what is possible.

Probably the best-known convolutional network implementation is Yann LeCun's **LeNet**. LeNet has gone through several iterations since LeNet-1 in late 1980, but has been increasingly effective at performing tasks including handwritten digit and image classification. LeNet is structured using alternating convolution and pooling layers capped by an MLP, as follows:



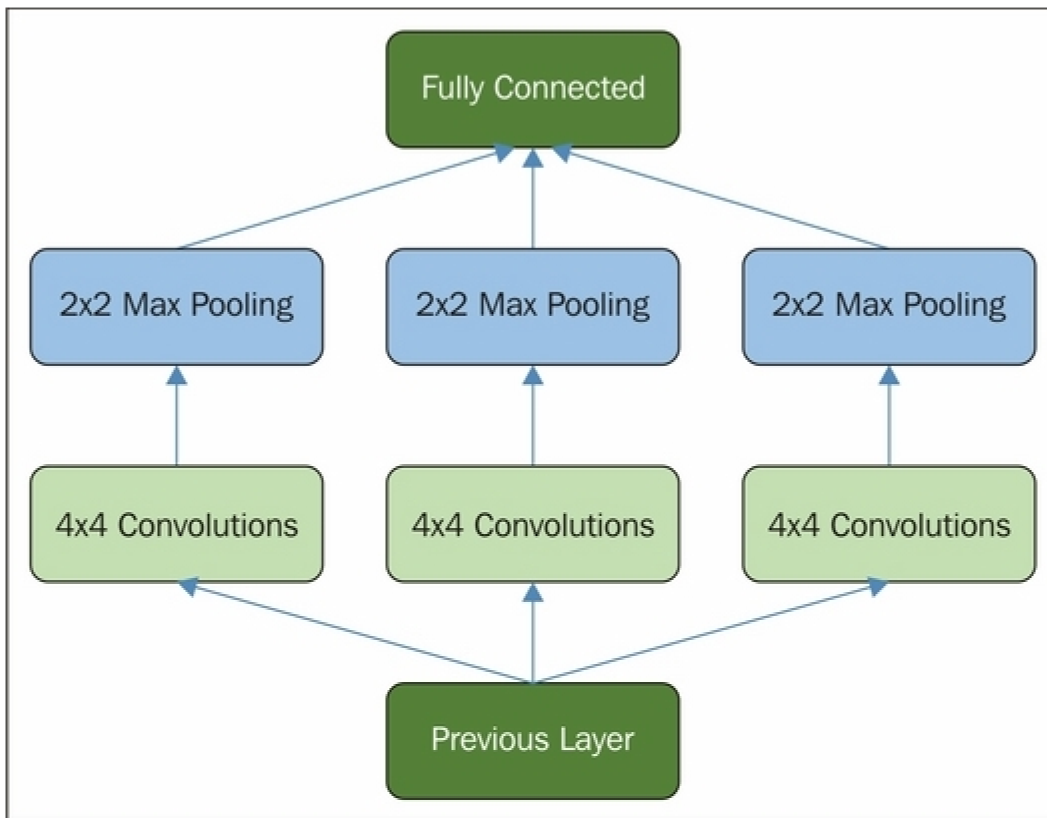
Each layer is partially-connected, as we discussed earlier, with the MLP being a fully connected layer. At each layer, multiple feature maps (channels) are employed; this gives us the advantage of being able to create more complex sets of filters. As we'll see, using multiple channels within a layer is a powerful technique employed in advanced use cases.

It's common to use max-pooling layers to reduce the dimensionality of the output to match the input as well as generally manage output volumes. How pooling is implemented, particularly in regard to the relative position of convolutional and pooling layers, is an element that tends to vary between implementations. It's generally common to develop a layer as a set of operations that feed into, and are fed into, a single **Fully Connected** layer, as shown in the following example:



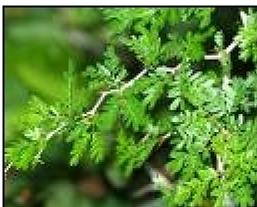
While this network structure wouldn't work in practice, it's a helpful illustration of the fact that a network can be constructed from the components you've learned about in a number of ways. How this network is structured and how complex it becomes should be motivated by the challenge the network is intended to solve. Different problems can call for very different solutions.

In the case of the LeNet implementation that we'll be working with later in this chapter, each layer contains multiple convolutional layers in parallel with a max-pooling layer following each. Diagrammatically, a LeNet layer looks like the following image:

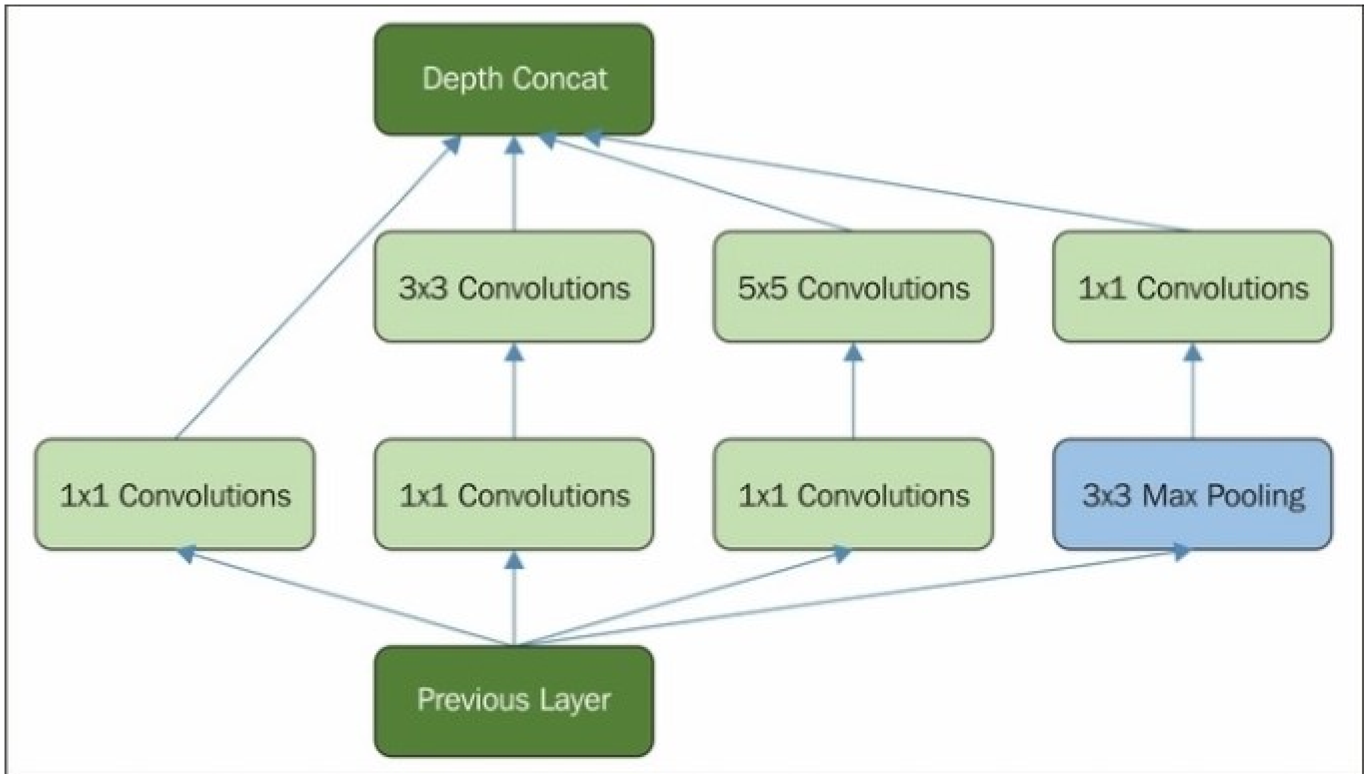


This architecture will enable us to start looking at some initial use cases quickly and easily, but in general won't perform well for some of the state-of-the-art applications we'll run into later in this book. Given this fact, there are some more extensive deep learning architectures designed to tackle the most challenging problems, whose topologies are worth discussing. One of the best-known convnet architectures is Google's **Inception** network, now more commonly known as GoogLeNet.

GoogLeNet was designed to tackle computer vision challenges involving Internet-quality image data, that is, images that have been captured in real contexts where the pose, lighting, occlusion, and clutter of images vary significantly. GoogLeNet was applied to the 2014 ImageNet challenge with noteworthy success, achieving only 6.7% error rate on the test dataset. ImageNet images are small, high-granularity images taken from many, varied classes. Multiple classes may appear very similar (such as varieties of tree) and the network architecture must be able to find increasingly challenging class distinctions to succeed. For a concrete example, consider the following ImageNet image:



Given the demands of this problem, the GoogLeNet architecture used to win ImageNet 14 departs from the LeNet model in several key ways. GoogLeNet's basic layer design is known as the Inception module and is made up of the following components:



The 1 x 1 convolutional layers used here are followed by **Rectified Linear Units** (**ReLU**). This approach is heavily used in speech and audio modeling contexts as ReLU can be used to effectively train deep models without pretraining and without facing some of the gradient vanishing problems that challenge other activation types. More information on ReLU is provided in the *Further reading* section of this chapter. The **DepthConcat** element provides a concatenation function, which consolidates the outputs of multiple units and substantially improves training time.

GoogLeNet chains layers of this type to create a full network. Indeed, the repetition of inception modules through GoogLeNet (nine times!) suggests that **Network In Network** (**NIN**) (deep architectures created from chained network modules) approaches are going to continue to be a serious contender in deep learning circles. The paper describing GoogLeNet and demonstrating how inception models were integrated into the network is provided in the Further reading section of this chapter.

Beyond the regularity of Inception module stacking, GoogLeNet has a few further surprises to throw at us. The first few layers are typically more straightforward with single-channel convolutional and max-pooling layers used at first. Additionally, at several points, GoogLeNet introduced a branch off the main structure using an average-pool layer, feeding into auxiliary softmax classifiers. The purpose of these classifiers was to improve the gradient signal that gets propagated back in lower layers of the network, enabling stronger performance at the early and middle network layers. Instead of one huge and

potentially vague backpropagation process stemming from the final layer of the network, GoogLeNet instead has several intermediary update sources.

What's really important to take from this implementation is that GoogLeNet and other top convnet architectures are mainly successful because they are able to find effective configurations using the highly available components that we've discussed in this chapter. Now that we've had a chance to discuss the architecture and components of a convolutional net and the opportunity to discuss how these components are used to construct some highly advanced networks, it's time to apply the techniques to solve a problem of our own!

Applying a CNN

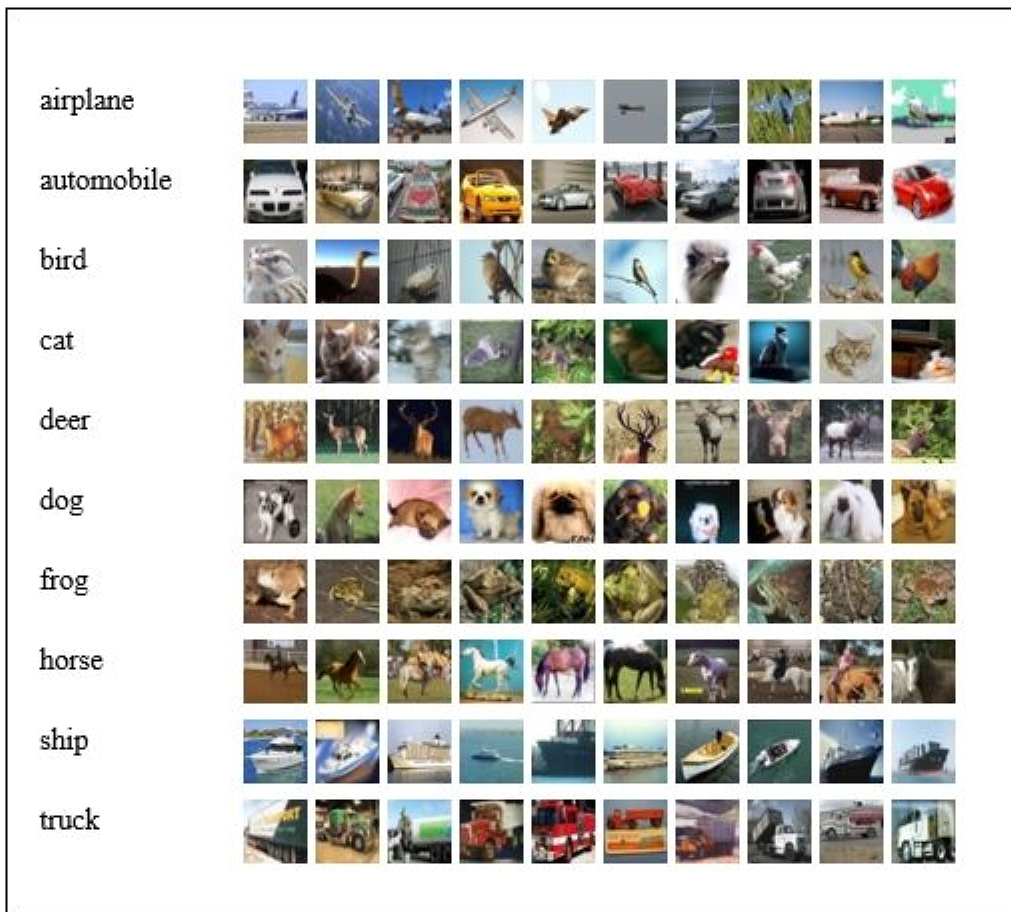
We'll be working with image data to try out our convnet. The image data that we worked with in earlier chapters, including the MNIST digit dataset, was a useful training dataset (with many valuable real-world applications such as automated check reading!). However, it differs from almost all photographic or video data in an important way; most visual data is highly noisy.

Problem variables can include pose, lighting, occlusion, and clutter, which may be expressed independently or in conjunction in huge variety. This means that the task of creating a function that is invariant to all properties of noise in the dataset is challenging; the function is typically very complex and nonlinear. In Chapter 7, *Feature Engineering Part II*, we'll discuss how techniques such as whitening can help mitigate some of these challenges, but as we'll see, even such techniques by themselves are insufficient to yield good classification (at least, without a very large investment of time!). By far, the most efficient solution to the problem of noise in image data, as we've already seen in multiple contexts, is to use a deep architecture rather than a broad one (that is, a neural network with few, high-dimensional layers, which is vulnerable to problematic overfitting and generalizability problems).

From discussions in previous chapters, the reasons for a deep architecture may already be clear; successive layers of a deep architecture reuse the reasoning and computation performed in preceding layers. Deep architectures can thus build a representation that is sequentially improved by successive layers of the network without performing extensive recalculation on any individual layer. This makes the challenging task of classifying large datasets of noisy photograph data achievable to a high level of accuracy in a relatively short time, without extensive feature engineering.

Now that we've discussed the challenges of modeling image data and advantages of a deep architecture in such contexts, let's apply a convnet to a real-world classification problem.

As in preceding chapters, we're going to start out with a toy example, which we'll use to familiarize ourselves with the architecture of our deep network. This time, we're going to take on a classic image processing challenge, CIFAR-10. CIFAR-10 is a dataset of 60,000 32 x 32 color images in 10 classes, with each class containing 6,000 images. The data is already split into five training batches, with one test batch. The classes and some images from each dataset are as follows:



While the industry has—to an extent—moved on to tackle other datasets such as ImageNet, CIFAR-10 was long regarded as the bar to reach in terms of image classification, with a great many data scientists attempting to create architectures that classify the dataset to human levels of accuracy, where human error rate is estimated at around 6%.

In November 2014, Kaggle ran a contest whose objective was to classify CIFAR-10 as accurately as possible. This contest's highest-scoring entry produced 95.55% classification accuracy, with the result using convolutional networks and a Network-in-Network approach. We'll discuss the challenge of classifying this dataset, as well as some of the more advanced techniques we can bring to bear, in Chapter 8, *Ensemble Methods*; for now, let's begin by having a go at classification with a convolutional network.

For our first attempt, we'll apply a fairly simple convolutional network with the following objectives:

- Applying a filter to the image and view the output
- Seeing the weights that our convnet created
- Understanding the difference between the outputs of effective and ineffective networks

In this chapter, we're going to take an approach that we haven't taken before, which will be of huge importance to you when you come to use these techniques in the wild. We saw earlier in this chapter how the deep architectures developed to solve different problems may differ structurally in many ways.

It's important to be able to create problem-specific network architectures so that we can adapt our implementation to fit a range of real-world problems. To do this, we'll be constructing our network using components that are modular and can be recombined in almost any way necessary, without too much additional effort. We saw the impact of modularity earlier in this chapter, and it's worth exploring how to apply this effect to our own networks.

As we discussed earlier in the chapter, convnets become particularly powerful when tasked to classify very large and varied datasets of up to tens or hundreds of thousands of images. As such, let's be a little ambitious and see whether we can apply a convnet to classify CIFAR-10.

In setting up our convolutional network, we'll begin by defining a useable class and initializing the relevant network parameters, particularly weights and biases. This approach will be familiar to readers of the preceding chapters.

```
class LeNetConvPoolLayer(object):

    def __init__(self, rng, input, filter_shape, image_shape,
    poolsize=(2, 2)):

        assert image_shape[1] == filter_shape[1]
        self.input = input

        fan_in = numpy.prod(filter_shape[1:])
        fan_out = (filter_shape[0] *
numpy.prod(filter_shape[2:])
                   numpy.prod(poolsize))

        W_bound = numpy.sqrt(6. / (fan_in + fan_out))
        self.W = theano.shared(
            numpy.asarray(
                rng.uniform(low=-W_bound, high=W_bound,
                size=filter_shape),
                dtype=theano.config.floatX
            ),
            borrow=True
        )
```

Before moving on to create the biases, it's worth reviewing what we have thus far. The `LeNetConvPoolLayer` class is intended to implement one full convolutional and pooling layer as per the LeNet layer structure. This class contains several useful initial parameters.

From previous chapters, we're familiar with the `rng` parameter used to initialize weights to random values. We can also recognize the `input` parameter. As in most cases, image input tends to take the

form of a symbolic image tensor. This image input is shaped by the `image_shape` parameter; this is a tuple or list of length 4 describing the dimensions of the input. As we move through successive layers, `image_shape` will reduce increasingly. As a tuple, the dimensions of `image_shape` simply specify the height and width of the input. As a list of length 4, the parameters, in order, are as follows:

- The batch size
- The number of input feature maps
- The height of the input image
- The width of the input image

While `image_shape` specifies the size of the input, `filter_shape` specifies the dimensions of the filter. As a list of length 4, the parameters, in order, are as follows:

- The number of filters (channels) to be applied
- The number of input feature maps
- The height of the filter
- The width of the filter

However, the height and width may be entered without any additional parameters. The final parameter here, `poolsize`, describes the downsizing factor. This is expressed as a list of length 2, the first element being the number of rows and the second—the number of columns.

Having defined these values, we immediately apply them to define the `LeNetConvPoolLayer` class better. In defining `fan_in`, we set the inputs to each hidden unit to be a multiple of the number of input feature maps—the filter height and width. Simply enough, we also define `fan_out`, a gradient that's calculated as a multiple of the number of output feature maps—the feature height and width—divided by the pooling size.

Next, we move on to defining the bias as a set of one-dimensional tensors, one for each output feature map:

```
b_values = numpy.zeros((filter_shape[0],),
dtype=theano.config.floatX)
self.b = theano.shared(value=b_values, borrow=True)

conv_out = conv.conv2d(
    input=input,
    filters=self.W,
    filter_shape=filter_shape,
    image_shape=image_shape
)
```

With this single function call, we've defined a convolution operation that uses the filters we previously defined. At times, it can be a little staggering to see how much theory needs to be known to effectively apply a single function! The next step is to create a similar pooling operation using `max_pool_2d`:

```
pooled_out = downsample.max_pool_2d(
    input=conv_out,
```

```
            ds=poolsize,
            ignore_border=True
        )


        self.output = T.tanh(pooled_out + self.b.dimshuffle('x',
                      0, 'x', 'x'))

        self.params = [self.W, self.b]

        self.input = input
```

Finally, we add the bias term, first reshaping it to be a tensor of shape (`1, n_filters, 1, 1`). This has the simple effect of causing the bias to affect every feature map and minibatch. At this point, we have all of the components we need to build a basic convnet. Let's move on to create our own network:

```
    x = T.matrix('x')
    y = T.ivector('y')
```

This process is fairly simple. We build the layers in order, passing parameters to the class that we previously specified. Let's start by building our first layer:

```
    layer0_input = x.reshape((batch_size, 1, 32, 32))

    layer0 = LeNetConvPoolLayer(
        rng,
        input=layer0_input,
        image_shape=(batch_size, 1, 32, 32),
        filter_shape=(nkerns[0], 1, 5, 5),
        poolsize=(2, 2)
    )
```

We begin by reshaping the input to spread it across all of the intended minibatches. As the CIFAR-10 images are of a 32 x 32 dimension, we've used this input size for the height and width dimensions. The filtering process reduces the size of this input to 32- 5+1 in each dimension, or 28. Pooling reduces this by half in each dimension to create an output layer of shape `(batch_size, nkerns[0], 14, 14)`.

This is a completed first layer. Next, we can attach a second layer to this using the same code:

```
    layer1 = LeNetConvPoolLayer(
        rng,
        input=layer0.output,
        image_shape=(batch_size, nkerns[0], 14, 14),
        filter_shape=(nkerns[1], nkerns[0], 5, 5),
        poolsize=(2, 2)
    )
```

As per the previous layer, the output shape for this layer is (`batch_size, nkerns[1], 5, 5`). So far, so good! Let's feed this output to the next, fully-connected sigmoid layer. To begin with, we need to flatten the input shape to two dimensions. With the values that we've fed to the network so far, the input will be a matrix of shape *(500, 1250)*. As such, we'll set up an appropriate `layer2`:

```
layer2_input = layer1.output.flatten(2)

layer2 = HiddenLayer(
    rng,
    input=layer2_input,
    n_in=nkerns[1] * 5 * 5
    n_out=500,
    activation=T.tanh
)
```

This leaves us in a good place to finish this network's architecture, by adding a final, logistic regression layer that calculates the values of the fully-connected sigmoid layer.

Let's try out this code:

```
x = T.matrix(CIFAR-10_train)
y = T.ivector(CIFAR-10_test)
```

`Chapter_4/convolutional_mlp.py`

The results that we obtained were as follows:

```
Optimization complete.
Best validation score of 0.885725 % obtained at iteration 17400,
with test performance 0.902508 %
The code for file convolutional_mlp.py ran for 26.50m
```

This accuracy score, at validation, is reasonably good. It's not at a human level of accuracy, which, as we established, is roughly 94%. Equally, it is not the best score that we could achieve with a convnet.

For instance, the Further Reading section of this chapter refers to a convnet implemented in Torch using a combination of dropout (which we studied in Chapter 3, *Stacked Denoising Autoencoders*) and **Batch Normalization** (a normalization technique intended to reduce covariate drift during the training process; refer to the Further Reading section for further technical notes and papers on this technique), which scored 92.45% validation accuracy.

A score of 88.57% is, however, in the same ballpark and can give us confidence that we're within striking distance of an effective network architecture for the CIFAR-10 problem. More importantly, you've learned a lot about how to configure and train a convolutional neural network effectively.

Further Reading

The glut of recent interest in Convolutional Networks means that we're spoiled for choice for further reading. One good option for an unfamiliar reader is the course notes from Andrej Karpathy's course: http://cs231n.github.io/convolutional-networks/.

For readers with an interest in the deeper details of specific best-in-class implementations, some of the networks referenced in this chapter were the following:

Google's GoogLeNet (http://www.cs.unc.edu/~wliu/papers/GoogLeNet.pdf)

Google Deepmind's Go-playing program AlphaGo (https://gogameguru.com/i/2016/03/deepmind-mastering-go.pdf)

Facebook's DeepFace architecture for facial recognition (https://www.cs.toronto.edu/~ranzato/publications/taigman_cvpr14.pdf)

The ImageNet LSVRC-2010 contest winning network, described here by Krizhevsky, Sutskever and Hinton (http://www.cs.toronto.edu/~fritz/absps/imagenet.pdf)

Finally, Sergey Zagoruyko's Torch implementation of a ConvNet with Batch normalization is available here: http://torch.ch/blog/2015/07/30/cifar.html.

Summary

In this chapter, we covered a lot of ground. We began by introducing a new kind of neural network, the convnet. We explored the theory and architecture of a convnet in the most ubiquitous form and also by discussing some state-of the-art network design principles that have been developing as recently as mid-2015 in organizations such as Google and Baidu. We built an understanding of the topology and also of how the network operates.

Following this, we began to work with the convnet itself, applying it to the CIFAR-10 dataset. We used modular convnet code to create a functional architecture that reached a reasonable level of accuracy in classifying 10-class image data. While we're definitely still at some distance from human levels of accuracy, we're gradually closing the gap! Chapter 8, *Ensemble Methods* will pick up from what you learned here, taking these techniques and their application to the next level.

Chapter 5. Semi-Supervised Learning

Introduction

In previous chapters, we've tackled a range of data challenges using advanced techniques. In each case, we've applied our techniques to datasets with reasonable success.

In many regards, though, we've had it pretty easy. Our data has been largely derived from canonical and well-prepared sources so we haven't had to do a great deal of preparation. In the real world, though, there are few datasets like this (except, perhaps, the ones that we're able to specify ourselves!). In particular, it is rare and improbable to come across a dataset in the wild, which has class labels available. Without labels on a sufficient portion of the dataset, we find ourselves unable to build a classifier that can accurately predict labels on validation or test data. So, what do we do?

The common solution is attempt to tag our data manually; not only is this time-consuming, but it also suffers from certain types of human error (which are especially common with high-dimensional datasets, where a human observer is unable to identify class boundaries as well as a computational approach might).

A fairly new and quite exciting alternative approach is to use **semi-supervised learning** to apply labels to unlabeled data via capturing the shape of underlying distributions. Semi-supervised learning has been growing in popularity over the last decade for its ability to save large amounts of annotation time, where annotation, if possible, may potentially require human expertise or specialist equipment. Contexts where this has proven to be particularly valuable have been natural language parsing and speech signal analysis; in both areas, manual annotation has proven to be complex and time-consuming.

In this chapter, you're going to learn how to apply several semi-supervised learning techniques, including, **Contrastive Pessimistic Likelihood Estimation** (**CPLE**), self learning, and S3VM. These techniques will enable us to label training data in a range of otherwise problematic contexts. You'll learn to identify the capabilities and limitations of semi-supervised techniques. We'll use a number of recent Python libraries developed on top of scikit-learn to apply semi-supervised techniques to several use cases, including audio signal data.

Let's get started!

Understanding semi-supervised learning

The most persistent cost in performing machine learning is the creation of tagged data for training purposes. Datasets tend not to come with class labels provided due to the circularity of the situation; one needs a trained classification technique to generate class labels, but cannot train the technique without labeled training and test data. As mentioned, tagging data manually or via test processes is one option, but this can be prohibitively time-consuming, costly (particularly for medical tests), challenging to organize, and prone to error (with large or complex datasets). Semi-supervised techniques suggest a better way to break this deadlock.

Semi-supervised learning techniques use both unlabeled and labeled data to create better learning techniques than can be created with either unlabeled or labeled data individually. There is a family of techniques that exists in a space between supervised (with labeled data) and unsupervised (with unlabeled data) learning.

The main types of technique that exist in this group are semi-supervised techniques, transductive techniques, and active learning techniques, as well as a broad set of other methods.

Semi-supervised techniques leave a set of test data out of the training process so as to perform testing at a later stage. Transductive techniques, meanwhile, are purely intended to develop labels for unlabeled data. There may not be a test process embedded in a transductive technique and there may not be labeled data available for use.

In this chapter, we'll focus on a set of semi-supervised techniques that deliver powerful dataset labeling capability in very familiar formats. A lot of the techniques that we'll be discussing are useable as wrappers around familiar, pre-existing classifiers, from linear regression classifiers to SVMs. As such, many of them can be run using estimators from Scikit-learn. We'll begin by applying a linear regression classifier to test cases before moving on to apply an SVM with semi-supervised extensions.

Semi-supervised algorithms in action

We've discussed what semi-supervised learning is, why we want to engage in it, and what some of the general realities of employing semi-supervised algorithms are. We've gone about as far as we can with general descriptions. Over the next few pages, we'll move from this general understanding to develop an ability to use a semi-supervised application effectively.

Self-training

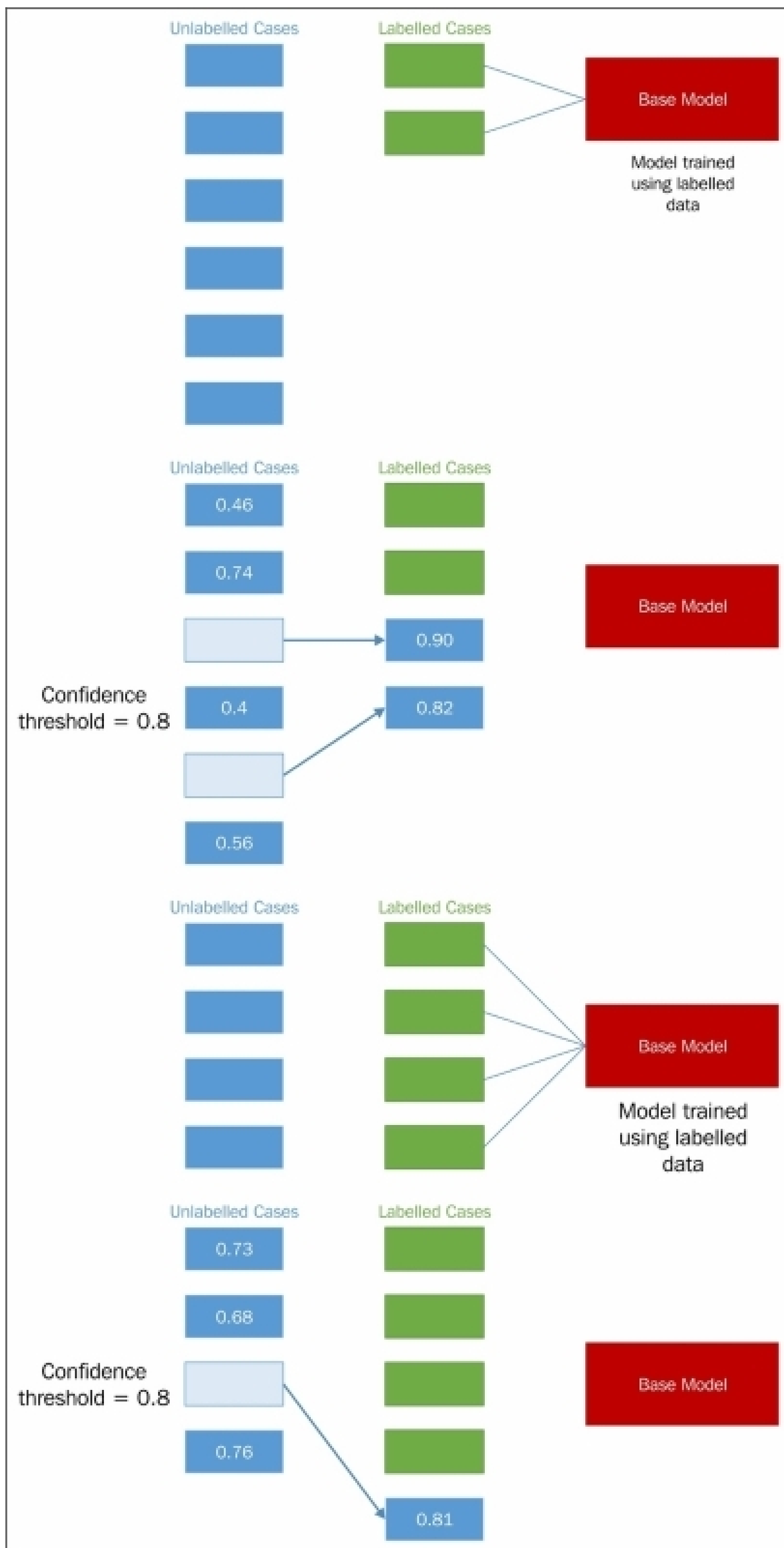
Self-training is the simplest semi-supervised learning method and can also be the fastest. Self-training algorithms see an application in multiple contexts, including NLP and computer vision; as we'll see, they can present both substantial value and significant risks.

The objective of self-training is to combine information from unlabeled cases with that of labeled cases to iteratively identify labels for the dataset's unlabeled examples. On each iteration, the labeled training set is enlarged until the entire dataset is labeled.

The self-training algorithm is typically applied as a wrapper to a base model. In this chapter, we'll be using an SVM as the base for our self-training model. The self-training algorithm is quite simple and contains very few steps, as follows:

1. A set of labeled data is used to predict labels for a set of unlabeled data. (This may be all unlabeled data or part of it.)
2. Confidence is calculated for all newly labeled cases.
3. Cases are selected from the newly labeled data to be kept for the next iteration.
4. The model trains on all labeled cases, including cases selected in previous iterations.
5. The model iterates through steps 1 to 4 until it successfully converges.

Presented graphically, this process looks as follows:


Unlabelled Cases
Labelled Cases
Base Model
Model trained using labelled data
Unlabelled Cases
Labelled Cases
0.46
0.74
0.90
Base Model
Confidence threshold = 0.8
0.4
0.82
0.56
Unlabelled Cases
Labelled Cases
Base Model
Model trained using labelled data
Unlabelled Cases
Labelled Cases
0.73
0.68
Confidence threshold = 0.8
Base Model
0.76
0.81

Upon completing training, the self-trained model would be tested and validated. This may be done via cross-validation or even using held-out, labeled data, should this exist.

Self-training provides real power and time saving, but is also a risky process. In order to understand what to look out for and how to apply self-training to your own classification algorithms, let's look in more detail at how the algorithm works.

To support this discussion, we're going to work with code from the semisup-learn GitHub repository. In order to use this code, we'll need to clone the relevant GitHub repository. Instructions for this are located in Appendix A.

Implementing self-training

The first step in each iteration of self-training is one in which class labels are generated for unlabeled cases. This is achieved by first creating a `SelfLearningModel` class, which takes a base supervised model (`basemodel`) and an iteration limit as arguments. As we'll see later in this chapter, an iteration limit can be explicitly specified or provided as a function of classification accuracy (that is, convergence). The `prob_threshold` parameter provides a minimum quality bar for label acceptance; any projected label that scores at less than this level will be rejected. Again, we'll see in later examples that there are alternatives to providing a hardcoded threshold value.

```
class SelfLearningModel(BaseEstimator):


def __init__(self, basemodel, max_iter = 200, prob_threshold = 0.8):
   self.model = basemodel
   self.max_iter = max_iter
   self.prob_threshold = prob_threshold
```

Having defined the shell of the `SelfLearningModel` class, the next step is to define functions for the process of semi-supervised model fitting:

```
def fit(self, X, y):
   unlabeledX = X[y==-1, :]
   labeledX = X[y!=-1, :]
   labeledy = y[y!=-1]

   self.model.fit(labeledX, labeledy)
   unlabeledy = self.predict(unlabeledX)
   unlabeledprob = self.predict_proba(unlabeledX)
   unlabeledy_old = []

   i = 0
```

The `X` parameter is a matrix of input data, whose shape is equivalent to `[n_samples, n_features]`. `X` is used to create a matrix of `[n_samples, n_samples]` size. The `y` parameter, meanwhile, is an array of labels. Unlabeled points are marked as `-1` in `y`. From `X`, the `unlabeledX`

and `labeledX` parameters are created quite simply by operations over `X` that select elements in `X` whose position corresponds to a `-1` label in `y`. The `labeledy` parameter performs a similar selection over `y`. (Naturally, we're not that interested in the unlabeled samples of `y` as a variable, but we need the labels that do exist for classification attempts!)

The actual process of label prediction is achieved, first, using sklearn's predict operation. The `unlabeledy` parameter is generated using sklearn's `predict` method, while the `predict_proba` method is used to calculate probabilities for each projected label. These probabilities are stored in `unlabeledprob`.

Note

Scikit-learn's `predict` and `predict_proba` methods work to `predict` class labels and the probability of class labeling being correct, respectively. As we'll be applying both of these methods within several of our semi-supervised algorithms, it's informative to understand how they actually work.

The `predict` method produces class predictions for input data. It does so via a set of binary classifiers (that is, classifiers that attempt to differentiate only two classes). A full model with n-many classes contains a set of binary classifiers as follows:

$$\frac{n*(n-1)}{2}$$

In order to make a prediction for a given case, all classifiers whose scores exceed zero, vote for a class label to apply to that case. The class with the most votes (and not, say, the highest sum classifier score) is identified. This is referred to as a one-versus-one prediction method and is a fairly common approach.

Meanwhile, `predict_proba` works by invoking **Platt calibration**, a technique that allows the outputs of a classification model to be transformed into a probability distribution over the classes. This involves first training the base model in question, fitting a regression model to the classifier's scores:

$$P(y|X) = \frac{1}{(1+\exp(A*f(X)+B))}$$

This model can then be optimized (through scalar parameters *A* and *B*) using a maximum likelihood method. In the case of our self-training model, `predict_proba` allows us to fit a regression model to the classifier's scores and thus calculate probabilities for each class label. This is extremely helpful!

Next, we need a loop for iteration. The following code describes a `while` loop that executes until there are no cases left in `unlabeledy_old` (a copy of `unlabeledy`) or until the max iteration count is

reached. On each iteration, a labeling attempt is made for each case that does not have a label whose probability exceeds the probability threshold (`prob_threshold`):

```
   while (len(unlabeledy_old) == 0 or
      numpy.any(unlabeledy!=unlabeledy_old)) and i < self.max_iter:
      unlabeledy_old = numpy.copy(unlabeledy)
      uidx = numpy.where((unlabeledprob[:, 0] >
self.prob_threshold)
      | (unlabeledprob[:, 1] > self.prob_threshold))[0]
```

The `self.model.fit` method then attempts to fit a model to the unlabeled data. This unlabeled data is presented in a matrix of size `[n_samples, n_samples]` (as referred to earlier in this chapter). This matrix is created by appending (with `vstack` and `hstack`) the unlabeled cases:

```
      self.model.fit(numpy.vstack((labeledX, unlabeledX[uidx,
:])),
      numpy.hstack((labeledy, unlabeledy_old[uidx])))
```

Finally, the iteration performs label predictions, followed by probability predictions for those labels.

```
      unlabeledy = self.predict(unlabeledX)
      unlabeledprob = self.predict_proba(unlabeledX)
      i += 1
```

On the next iteration, the model will perform the same process, this time taking the newly labeled data whose probability predictions exceeded the threshold as part of the dataset used in the `model.fit` step.

If one's model does not already include a classification method that can generate label predictions (like the `predict_proba` method available in sklearn's SVM implementation), it is possible to introduce one. The following code checks for the `predict_proba` method and introduces `Platt scaling` of generated labels if this method is not found:

```
if not getattr(self.model, "predict_proba", None):
   self.plattlr = LR()
   preds = self.model.predict(labeledX)
   self.plattlr.fit( preds.reshape( -1, 1 ), labeledy )

return self



def predict_proba(self, X):
        if getattr(self.model, "predict_proba", None):
        return self.model.predict_proba(X)
        else:
```

```
            preds = self.model.predict(X)
            return self.plattlr.predict_proba(preds.reshape( -1, 1 ))
```

Once we have this much in place, we can begin applying our self-training architecture. To do so, let's grab a dataset and start working!

For this example, we'll use a simple linear regression classifier, with **Stochastic Gradient Descent** (**SGD**) as our learning component as our base model (`basemodel`). The input dataset will be the statlog `heart` dataset, obtained from www.mldata.org. This dataset is provided in the GitHub repository accompanying this chapter.

The `heart` dataset is a two-class dataset, where the classes are the absence or presence of a heart disease. There are no missing values across the 270 cases for any of its 13 features. This data is unlabeled and many of the variables needed are usually captured via expensive and sometimes inconvenient tests. The variables are as follows:

- `age`
- `sex`
- `chest pain type (4 values)`
- `resting blood pressure`
- `serum cholestoral in mg/dl`
- `fasting blood sugar > 120 mg/dl`
- `resting electrocardiographic results (values 0,1,2)`
- `maximum heart rate achieved`
- `exercise induced angina`
- `10. oldpeak = ST depression induced by exercise relative to rest`
- `the slope of the peak exercise ST segment`
- `number of major vessels (0-3) colored by flourosopy`
- `thal: 3 = normal; 6 = fixed defect; 7 = reversable defect`

Lets get started with the `Heart` dataset by loading in the data, then fitting a model to it:

```
heart = fetch_mldata("heart")
X = heart.data
ytrue = np.copy(heart.target)
ytrue[ytrue==-1]=0

labeled_N = 2
ys = np.array([-1]*len(ytrue)) # -1 denotes unlabeled point
random_labeled_points = random.sample(np.where(ytrue == 0)[0],
labeled_N/2)+\random.sample(np.where(ytrue == 1)[0], labeled_N/2)
ys[random_labeled_points] = ytrue[random_labeled_points]

basemodel = SGDClassifier(loss='log', penalty='l1')

basemodel.fit(X[random_labeled_points, :], ys[random_labeled_points])
print "supervised log.reg. score", basemodel.score(X, ytrue)
```

```
ssmodel = SelfLearningModel(basemodel)
ssmodel.fit(X, ys)
print "self-learning log.reg. score", ssmodel.score(X, ytrue)
```

Attempting this yields moderate, but not excellent, results:

`self-learning log.reg. score 0.470347`

However, over 1,000 trials, we find that the quality of our outputs is quite variant:

Given that we're looking at classification accuracy scores for sets of real-world and unlabeled data, this isn't a terrible result, but I don't think we should be satisfied with it. We're still labeling more than half of our cases incorrectly!

We need to understand the problem a little better; right now, it isn't clear what's going wrong or how we can improve on our results. Let's figure this out by returning to the theory around self-training to understand how we can diagnose and improve our implementation.

Finessing your self-training implementation

In the previous section, we discussed the creation of self-training algorithms and tried out an implementation. However, what we saw during our first trial was that our results, while demonstrating the potential of self-training, left room for growth. Both the accuracy and variance of our results were questionable.

Self-training can be a fragile process. If an element of the algorithm is ill-configured or the input data contains peculiarities, it is very likely that the iterative process will fail once and continue to compound that error by reintroducing incorrectly labeled data to future labeling steps. As the self-training algorithm iteratively feeds itself, garbage in, garbage out is a very real concern.

There are several quite common flavors of risk that should be called out. In some cases, labeled data may not add more useful information. This is particularly common in the first few iterations, and understandably so! In general, unlabeled cases that are most easily labeled are the ones that are most similar to existing labeled cases. However, while it's easy to generate high-probability labels for these cases, there's no guarantee that their addition to the labeled set will make it easier to label during subsequent iterations.

Unfortunately, this can sometimes lead to a situation in which cases are being added that have no real effect on classification while classification accuracy in general deteriorates. Even worse, adding cases that are similar to pre-existing cases in enough respects to make them easy to label, but that actually misguide the classifier's decision boundary, can introduce misclassification increases.

Diagnosing what went wrong with a self-training model can sometimes be difficult, but as always, a few well-chosen plots add a lot of clarity to the situation. As this type of error occurs particularly often within the first few iterations, simply adding an element to the label prediction loop that writes the current classification accuracy allows us to understand how accuracy trended during early iterations.

Once the issue has been identified, there are a few possible solutions. If enough labeled data exists, a simple solution is to attempt to use a more diverse set of labeled data to kick-start the process.

While the impulse might be to use all of the labeled data, we'll see later in this chapter that self-training models are vulnerable to overfitting—a risk that forces us to hold on to some data for validation purposes. A promising option is to use multiple subsets of our dataset to train multiple self-training model instances. Doing so, particularly over several trials, can help us understand the impact of our input data on our self-training models performance.

In Chapter 8, *Ensemble Methods*, we'll explore some options around ensembles that will enable us to use multiple self-training models together to yield predictions. When ensembling is accessible to us, we can even consider applying multiple sampling techniques in parallel.

If we don't want to solve this problem with quantity, though, perhaps we can solve it by improving quality. One solution is to create an appropriately diverse subset of the labeled data through selection. There isn't a hard limit on the number of labeled cases that works well as a minimum amount to start up a self-training implementation. While you could hypothetically start working with even one labeled case per class (as we did in our preceding training example), it'll quickly become obvious that training against a more diverse and overlapping set of classes benefits from more labeled data.

Another class of error that a self-training model is particularly vulnerable to is biased selection. Our naïve assumption is that the selection of data during each iteration is, at worst, only slightly biased (favoring one class only slightly more than others). The reality is that this is not a safe assumption. There are several factors that can influence the likelihood of biased selection, with the most likely culprit being disproportionate sampling from one class.

If the dataset as a whole, or the labeled subsets used, are biased toward one class, then the risk increases that your self-training classifier will overfit. This only compounds the problem as the cases provided for the next iteration are liable to be insufficiently diverse to solve the problem; whatever incorrect decision boundary was set up by the self-training algorithm will be set where it is—overfit to a subset of the data.

Numerical disparity between each class' count of cases is the main symptom here, but the more usual methods to spot overfitting can also be helpful in diagnosing problems around selection bias.

Note

This reference to the usual methods of spotting overfitting is worth expanding on because techniques to identify overfitting are highly valuable! These techniques are typically referred to as validation techniques. The fundamental concept underpinning validation techniques is that one has two sets of data—one that is used to build a model, and the other is used to test it.

The most effective validation technique is independent validation, the simplest form of which involves waiting to determine whether predictions are accurate. This obviously isn't always (or even, often) possible!

Given that it may not be possible to perform independent validation, the best bet is to hold out a subset of your sample. This is referred to as sample splitting and is the foundation of modern validation techniques. Most machine learning implementations refer to training, test, and validation datasets; this is a case of multilayered validation in action.

A third and critical validation tool is resampling, where subsets of the data are iteratively used to repeatedly validate the dataset. In Chapter 1, *Unsupervised Machine Learning*, we saw the use of v-fold cross-validation; cross-validation techniques are perhaps the best examples of resampling in action.

Beyond applicable techniques, it's a good idea to be mindful of the needed sample size required for the effective modeling of your data. There are no universal principles here, but I always rather liked the following rule of thumb:

If m points are required to determine a univariate regression line with sufficient precision, then it will take at least mn observations and perhaps $n!mn$ observations to appropriately characterize and evaluate a regression model with n variables.

Note that there is some tension between the suggested solutions to this problem (resampling, sample splitting, and validation techniques including cross-validation) and the preceding one. Namely, overfitting requires a more restrained use of subsets of the labeled training data, while bad starts are less likely to occur using more training data. For each specific problem, depending on the complexity of the data under analysis, there will be an appropriate balance to strike. By monitoring for signs of either type of problem, the appropriate action (whether that is an increase or decrease in the amount of labeled data used simultaneously in an iteration) can be taken at the right time.

A further class of risk introduced by self-training is that the introduction of unlabeled data almost always introduces noise. If dealing with datasets where part or all of the unlabeled cases are highly noisy, the amount of noise introduced may be sufficient to degrade classification accuracy.

Note

The idea of using data complexity and noise measures to understand the degree of noise in one's dataset is not new. Fortunately for us, quite a lot of good estimators already exist that we can take advantage of.

There are two main groups of relative complexity measures. Some attempt to measure the overlap of values of different classes, or separability; measures in this group attempt to describe the degree of ambiguity of each class relative to the other classes. One good measure for such cases is the maximum **Fisher's discriminant ratio**, though maximum individual feature efficiency is also effective.

Alternatively (and sometimes more simply), one can use the error function of a linear classifier to understand how separable the dataset's classes are from one another. By attempting to train a simple linear classifier on your dataset and observing the training error, one can immediately get a good understanding as to how linearly separable the classes are. Furthermore, measures related to this classifier (such as the fraction of points in the class boundary or the ratio of average intra/inter class nearest neighbor distance) can also be extremely helpful.

There are other data complexity measures that specifically measure the density or geometry of the dataset. One good example is the fraction of maximum covering spheres. Again, helpful measures can be accessed by applying a linear classifier and including the nonlinearity of that classifier.

Improving the selection process

The key to the self-training algorithm working correctly is the accurate calculation of confidence for each label projection. Confidence calculation is the key to successful self-training.

During our first explanation of self-training, we used some simplistic values for certain parameters, including a parameter closely tied to confidence calculation. In selecting our labeled cases, we used a fixed confidence level for comparison against predicted probabilities, where we could've adopted any one of several different strategies:

- Adding all of the projected labels to the set of labeled data
- Using a confidence threshold to select only the few most confident labels to the set
- Adding all the projected labels to the labeled dataset and weighing each label by confidence

All in all, we've seen that self-training implementations present quite a lot of risk. They're prone to a number of training failures and are also subject to overfitting. To make matters worse, as the amount of unlabeled data increases, the accuracy of a self-training classifier becomes increasingly at risk.

Our next step will be to look at a very different self-training implementation. While conceptually similar to the algorithm that we worked with earlier in this chapter, the next technique we'll be looking at operates under different assumptions to yield very different results.

Contrastive Pessimistic Likelihood Estimation

In our preceding discovery and application of self-training techniques, we found self-training to be a powerful technique with significant risks. Particularly, we found a need for multiple diagnostic tools and some quite restrictive dataset conditions. While we can work around these problems by subsetting, identifying optimal labeled data, and attentively tracking performance for some datasets, some of these actions continue to be impossible for the very data that self-training would bring the most benefit to—data where labeling requires expensive tests, be those medical or scientific, with specialist knowledge and equipment.

In some cases, we end up with some self-training classifiers that are outperformed by their supervised counterparts, which is a pretty terrible state of affairs. Even worse, while a supervised classifier with labeled data will tend to improve in accuracy with additional cases, semi-supervised classifier performance can degrade as the dataset size increases. What we need, then, is a less naïve approach to semi-supervised learning. Our goal should be to find an approach that harnesses the benefits of semi-supervised learning while maintaining performance at least comparable with that of the same classifier under a supervised approach.

A very recent (May 2015) approach to self-supervised learning, CPLE, provides a more general way to perform semi-supervised parameter estimation. CPLE provides a rather remarkable advantage: it produces label predictions that have been demonstrated to consistently outperform those created by equivalent semi-supervised classifiers or by supervised classifiers working from the labeled data! In other words, when performing a linear discriminant analysis, for instance, it is advised that you perform a CPLE-based, semi-supervised analysis instead of a supervised one, as you will always obtain at least equivalent performance.

This is a pretty big claim and it needs substantiating. Let's start by building an understanding of how CPLE works before moving on to demonstrate its superior performance in real cases.

CPLE uses the familiar measure of maximized log-likelihood for parameter optimization. This can be thought of as the success condition; the model we'll develop is intended to optimize the maximized log-likelihood of our model's parameters. It is the specific guarantees and assumptions that CPLE incorporates that make the technique effective.

In order to create a better semi-supervised learner—one that improves on it's supervised alternative—CPLE takes the supervised estimates into account explicitly, using the loss incurred between the semi-supervised and supervised models as a training performance measure:

CPLE calculates the relative improvement of any semi-supervised estimate over the supervised solution. Where the supervised solution outperforms the semi-supervised estimate, the loss function shows this and the model can train to adjust the semi-supervised model to reduce this loss. Where the semi-supervised solution outperforms the supervised solution, the model can learn from the semi-supervised model by adjusting model parameters.

However, while this sounds excellent so far, there is a flaw in the theory that has to be addressed. The fact that data labels don't exist for a semi-supervised solution means that the posterior distribution (that CPLE would use to calculate loss) is inaccessible. CPLE's solution to this is to be pessimistic. The CPLE algorithm takes the **Cartesian product** of all label/prediction combinations and then selects the posterior distribution that minimizes the gain in likelihood.

In real-world machine learning contexts, this is a very safe approach. It delivers the classification accuracy of a supervised approach with semi-supervised performance improvement derived via conservative assumptions. In real applications, these conservative assumptions enable high performance

under testing. Even better, CPLE can deliver particular performance improvements on some of the most challenging unsupervised learning cases, where the labeled data is a poor representation of the unlabeled data (by virtue of poor sampling from one or more classes or just because of a shortage of unlabeled cases).

In order to understand how much more effective CPLE can be than semi-supervised or supervised approaches, let's apply the technique to a practical problem. We'll once again work with the semisup-learn library, a specialist Python library, focused on semi-supervised learning, which extends scikit-learn to provide CPLE across any scikit-learn-provided classifier. We begin with a CPLE class:

```
class CPLELearningModel(BaseEstimator):

    def __init__(self, basemodel, pessimistic=True,
predict_from_probabilities = False, use_sample_weighting = True,
max_iter=3000, verbose = 1):
        self.model = basemodel
        self.pessimistic = pessimistic
        self.predict_from_probabilities = predict_from_probabilities
        self.use_sample_weighting = use_sample_weighting
        self.max_iter = max_iter
        self.verbose = verbose
```

We're already familiar with the concept of `basemodel`. Earlier in this chapter, we employed S3VMs and semi-supervised LDE's. In this situation, we'll again use an LDE; the goal of this first assay will be to try and exceed the results obtained by the semi-supervised LDE from earlier in this chapter. In fact, we're going to blow those results out of the water!

Before we do so, however, let's review the other parameter options. The `pessimistic` argument gives us an opportunity to use a non-pessimistic (optimistic) model. Instead of following the `pessimistic` method of minimizing the loss between unlabeled and labeled discriminative likelihood, an optimistic model aims to maximize likelihood. This can yield better results (mostly during training), but is significantly more risky. Here, we'll be working with pessimistic models.

The `predict_from_probabilities` parameter enables optimization by allowing a prediction to be generated from the probabilities of multiple data points at once. If we set this as true, our CPLE will set the prediction as `1` if the probability we're using for prediction is greater than the mean, or `0` otherwise. The alternative is to use the base model probabilities, which is generally preferable for performance reasons, unless we'll be calling `predict` across a number of cases.

We also have the option to `use_sample_weighting`, otherwise known as **soft labels** (but most familiar to us as posterior probabilities). We would normally take this opportunity, as soft labels enable greater flexibility than hard labels and are generally preferred (unless the model only supports hard class labels).

The first few parameters provide a means of stopping CPLE training, either at maximum iterations or after log-likelihood stops improving (typically because of convergence). The `bestdl` provides the best

discriminative likelihood value and corresponding soft labels; these values are updated on each training iteration:

```
        self.it = 0
        self.noimprovementsince = 0
        self.maxnoimprovementsince = 3

        self.buffersize = 200
        self.lastdls = [0]*self.buffersize

        self.bestdl = numpy.infty
        self.bestlbls = []


        self.id = 
str(unichr(numpy.random.randint(26)+97))+str(unichr(numpy.random.rand
int(26)+97))
```

The `discriminative_likelihood` function calculates the likelihood (for discriminative models—that is, models that aim to maximize the probability of a target—$y = 1$, conditional on the input, X) of an input.

Note

In this case, it's worth drawing your attention to the distinction between generative and discriminative models. While this isn't a basic concept, it can be fundamental in understanding why many classifiers have the goals that they do.

A classification model takes input data and attempts to classify cases, assigning each case a label. There is more than one way to do this.

One approach is to take the cases and attempt to draw a decision boundary between them. Then we can take each new case as it appears and identify which side of the boundary it falls on. This is a **discriminative** learning approach.

Another approach is to attempt to model the distribution of each class individually. Once a model has been generated, the algorithm can use Bayes' rule to calculate the posterior distribution on the labels given input data. This approach is **generative** and is a very powerful approach with significant weaknesses (most of which tie into the question of how well we can model our classes). Generative approaches include Gaussian discriminant models (yes, that is a slightly confusing name) and a broad range of Bayesian models. More information, including some excellent recommended reading, is provided in the *Further reading* section of this chapter.

In this case, the function will be used on each iteration to calculate the likelihood of the predicted labels:

```
    def discriminative_likelihood(self, model, labeledData, labeledy 
= None, unlabeledData = None, unlabeledWeights = None, 
unlabeledlambda = 1, gradient=[], alpha = 0.01):
```

```
        unlabeledy = (unlabeledWeights[:, 0]<0.5)*1
        uweights = numpy.copy(unlabeledWeights[:, 0])

        uweights[unlabeledy==1] = 1-uweights[unlabeledy==1]

        weights = numpy.hstack((numpy.ones(len(labeledy)), uweights))
        labels = numpy.hstack((labeledy, unlabeledy))
```

Having defined this much of our CPLE, we also need to define the fitting process for our supervised model. This uses familiar components, namely, `model.fit` and `model.predict_proba`, for probability prediction:

```
        if self.use_sample_weighting:
            model.fit(numpy.vstack((labeledData, unlabeledData)),
labels, sample_weight=weights)
        else:
            model.fit(numpy.vstack((labeledData, unlabeledData)),
labels)


        P = model.predict_proba(labeledData)
```

In order to perform pessimistic CPLE, we need to derive both the labeled and unlabeled discriminative log likelihood. In order, we then perform `predict_proba` on both the labeled and unlabeled data:

```
        try:

            labeledDL = -sklearn.metrics.log_loss(labeledy, P)
        except Exception, e:
            print e
            P = model.predict_proba(labeledData)


        unlabeledP = model.predict_proba(unlabeledData)




        try:
            eps = 1e-15
            unlabeledP = numpy.clip(unlabeledP, eps, 1 - eps)
            unlabeledDL =
numpy.average((unlabeledWeights*numpy.vstack((1-unlabeledy,
unlabeledy)).T*numpy.log(unlabeledP)).sum(axis=1))
        except Exception, e:
            print e
            unlabeledP = model.predict_proba(unlabeledData)
```

Once we're able to calculate the discriminative log likelihood for both the labeled and unlabeled classification attempts, we can set an objective via the `discriminative_likelihood_objective` function. The goal here is to use the pessimistic (or optimistic, by choice) methodology to calculate `dl` on each iteration until the model converges or the maximum iteration count is hit.

On each iteration, a t-test is performed to determine whether the likelihoods have changed. Likelihoods should continue to change on each iteration preconvergence. Sharp-eyed readers may have noticed earlier in the chapter that three consecutive t-tests showing no change will cause the iteration to stop (this is configurable via the `maxnoimprovementsince` parameter):

```
        if self.pessimistic:
            dl = unlabeledlambda * unlabeledDL - labeledDL
        else:
            dl = - unlabeledlambda * unlabeledDL - labeledDL

        return dl


    def discriminative_likelihood_objective(self, model,
labeledData, labeledy = None, unlabeledData = None, unlabeledWeights
= None, unlabeledlambda = 1, gradient=[], alpha = 0.01):
        if self.it == 0:
            self.lastdls = [0]*self.buffersize

        dl = self.discriminative_likelihood(model, labeledData,
labeledy, unlabeledData, unlabeledWeights, unlabeledlambda,
gradient, alpha)

        self.it += 1
        self.lastdls[numpy.mod(self.it, len(self.lastdls))] = dl

        if numpy.mod(self.it, self.buffersize) == 0: # or True:
            improvement =
numpy.mean((self.lastdls[(len(self.lastdls)/2):])) -
numpy.mean((self.lastdls[:(len(self.lastdls)/2)]))

            _, prob =
scipy.stats.ttest_ind(self.lastdls[(len(self.lastdls)/2):],
self.lastdls[:(len(self.lastdls)/2)])

            noimprovement = prob > 0.1 and
numpy.mean(self.lastdls[(len(self.lastdls)/2):]) <
numpy.mean(self.lastdls[:(len(self.lastdls)/2)])
            if noimprovement:
                self.noimprovementsince += 1
                if self.noimprovementsince >=
```

```
self.maxnoimprovementsince:

                    self.noimprovementsince = 0
                    raise Exception(" converged.")
            else:
                self.noimprovementsince = 0
```

On each iteration, the algorithm saves the best discriminative likelihood and the best weight set for use in the next iteration:

```
        if dl < self.bestdl:
            self.bestdl = dl
            self.bestlbls = numpy.copy(unlabeledWeights[:, 0])

        return dl
```

One more element worth discussing is how the soft labels are created. We've discussed these earlier in the chapter. This is how they look in code:

```
f = lambda softlabels, grad=[]:
self.discriminative_likelihood_objective(self.model, labeledX,
labeledy=labeledy, unlabeledData=unlabeledX,
unlabeledWeights=numpy.vstack((softlabels, 1-softlabels)).T,
gradient=grad)

lblinit = numpy.random.random(len(unlabeledy))
```

In a nutshell, `softlabels` provide a probabilistic version of the discriminative likelihood calculation. In other words, they act as weights rather than hard, binary class labels. Soft labels are calculable using the `optimize` method:

```
        try:
            self.it = 0
            opt = nlopt.opt(nlopt.GN_DIRECT_L_RAND, M)
            opt.set_lower_bounds(numpy.zeros(M))
            opt.set_upper_bounds(numpy.ones(M))
            opt.set_min_objective(f)
            opt.set_maxeval(self.max_iter)
            self.bestsoftlbl = opt.optimize(lblinit)
            print " max_iter exceeded."
        except Exception, e:
            print e
            self.bestsoftlbl = self.bestlbls

        if numpy.any(self.bestsoftlbl != self.bestlbls):
            self.bestsoftlbl = self.bestlbls
        ll = f(self.bestsoftlbl)
```

```
        unlabeledy = (self.bestsoftlbl<0.5)*1
        uweights = numpy.copy(self.bestsoftlbl)

        uweights[unlabeledy==1] = 1-uweights[unlabeledy==1]

        weights = numpy.hstack((numpy.ones(len(labeledy)), uweights))
        labels = numpy.hstack((labeledy, unlabeledy))
```

Note

For interested readers, optimize uses the **Newton conjugate gradient** method of calculating gradient descent to find optimal weight values. A reference to Newton conjugate gradient is provided in the Further reading section at the end of this chapter.

Once we understand how this works, the rest of the calculation is a straightforward comparison of the best supervised labels and soft labels, setting the `bestsoftlabel` parameter as the best label set. Following this, the discriminative likelihood is computed against the best label set and a `fit` function is calculated:

```
        if self.use_sample_weighting:
            self.model.fit(numpy.vstack((labeledX, unlabeledX)),
labels, sample_weight=weights)
        else:
            self.model.fit(numpy.vstack((labeledX, unlabeledX)),
labels)

        if self.verbose > 1:
            print "number of non-one soft labels: ",
numpy.sum(self.bestsoftlbl != 1), ", balance:",
numpy.sum(self.bestsoftlbl<0.5), " / ", len(self.bestsoftlbl)
            print "current likelihood: ", ll
```

Now that we've had a chance to understand the implementation of CPLE, let's get hands-on with an interesting dataset of our own! This time, we'll change things up by working with the University of Columbia's Million Song Dataset.

The central feature of this algorithm is feature analysis and metadata for one million songs. The data is preprepared and made up of natural and derived features. Available features include things such as the artist's name and ID, duration, loudness, time signature, and tempo of each song, as well as other measures including a crowd-rated danceability score and tags associated with the audio.

This dataset is generally labeled (via tags), but our objective in this case will be to generate genre labels for different songs based on the data provided. As the full million song dataset is a rather forbidding 300 GB, let's work with a 1% (1.8 GB) subset of 10,000 records. Furthermore, we don't particularly need this data as it currently exists; it's in an unhelpful format and a lot of the fields are going to be of little use to us.

The `10000_songs` dataset residing in the Chapter 6, *Text Feature Engineering* folder of our *Mastering Python Machine Learning* repository is a cleaned, prepared (and also rather large) subset of music data from multiple genres. In this analysis, we'll be attempting to predict genre from the genre tags provided as targets. We'll take a subset of tags as the labeled data used to kick-start our learning and will attempt to generate tags for unlabelled data.

In this iteration, we're going to raise our game as follows:

- Using more labeled data. This time, we'll use 1% of the total dataset size (100 songs), taken at random, as labeled data.
- Using an SVM with a linear kernel as our classifier, rather than the simple linear discriminant analysis we used with our naïve self-training implementation earlier in this chapter.

So, let's get started:

```
import sklearn.svm
import numpy as np
import random

from frameworks.CPLELearning import CPLELearningModel
from methods import scikitTSVM
from examples.plotutils import evaluate_and_plot

kernel = "linear"

songs = fetch_mldata("10000_songs")
X = songs.data
ytrue = np.copy(songs.target)
ytrue[ytrue==-1]=0

labeled_N = 20
ys = np.array([-1]*len(ytrue))
random_labeled_points = random.sample(np.where(ytrue == 0)[0],
labeled_N/2)+\
                        random.sample(np.where(ytrue == 1)[0],
labeled_N/2)
ys[random_labeled_points] = ytrue[random_labeled_points]
```

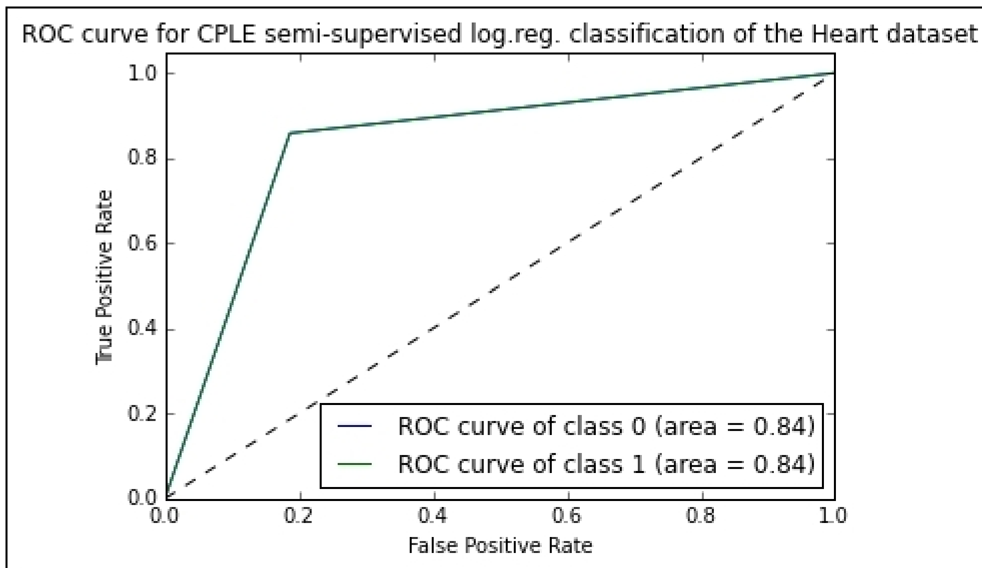
For comparison, we'll run a supervised SVM alongside our CPLE implementation. We'll also run the naïve self-supervised implementation, which we saw earlier in this chapter, for comparison:

```
basemodel = SGDClassifier(loss='log', penalty='l1') # scikit
logistic regression
basemodel.fit(X[random_labeled_points, :], ys[random_labeled_points])
print "supervised log.reg. score", basemodel.score(X, ytrue)


ssmodel = SelfLearningModel(basemodel)
```

```
ssmodel.fit(X, ys)
print "self-learning log.reg. score", ssmodel.score(X, ytrue)


ssmodel = CPLELearningModel(basemodel)
ssmodel.fit(X, ys)
print "CPLE semi-supervised log.reg. score", ssmodel.score(X, ytrue)
```

The results that we obtain on this iteration are very strong:

```
# supervised log.reg. score 0.698
# self-learning log.reg. score 0.825
# CPLE semi-supervised log.reg. score 0.833
```



The CPLE semi-supervised model succeeds in classifying with 84% accuracy, a score comparable to human estimation and over 10% higher than the naïve semi-supervised implementation. Notably, it also outperforms the supervised SVM.

Further reading

A solid place to start understanding Semi-supervised learning methods is Xiaojin Zhu's very thorough literature survey, available at http://pages.cs.wisc.edu/~jerryzhu/pub/ssl_survey.pdf.

I also recommend a tutorial by the same author, available in the slide format at http://pages.cs.wisc.edu/~jerryzhu/pub/sslicml07.pdf.

The key paper on Contastive Pessimistic Likelihood Estimation is Loog's 2015 paper http://arxiv.org/abs/1503.00269.

This chapter made a reference to the distinction between generative and discriminative models. A couple of relatively clear explanations of the distinction between generative and discriminative algorithms are provided by Andrew Ng (http://cs229.stanford.edu/notes/cs229-notes2.pdf) and Michael Jordan (http://www.ics.uci.edu/~smyth/courses/cs274/readings/jordan_logistic.pdf).

For readers interested in Bayesian statistics, Allen Downey's book, *Think Bayes*, is a marvelous introduction (and one of my all-time favorite statistics books): https://www.google.co.uk/#q=think+bayes.

For readers interested in learning more about gradient descent, I recommend Sebastian Ruder's blog at http://sebastianruder.com/optimizing-gradient-descent/.

For readers interested in going a little deeper into the internals of conjugate descent, Jonathan Shewchuk's introduction provides clear and enjoyable definitions for a number of key concepts at https://www.cs.cmu.edu/~quake-papers/painless-conjugate-gradient.pdf.

Summary

In this chapter, we tapped into a very powerful but lesser known paradigm in machine learning—semi-supervised learning. We began by exploring the underlying concepts of transductive learning and self-training, and improved our understanding of the latter class of techniques by working with a naïve self-training implementation.

We quickly began to see weaknesses in self-training and looked for an effective solution, which we found in the form of CPLE. CPLE is a very elegant and highly applicable framework for semi-supervised learning that makes no assumptions beyond those of the classifier that it uses as a base model. In return, we found CPLE to consistently offer performance in excess of naïve semi-supervised and supervised implementations, at minimal risk. We've gained a significant amount of understanding regarding one of the most useful recent developments in machine learning.

In the next chapter, we'll begin discussing data preparation skills that significantly increase the effectiveness of all of the models that we've previously discussed.

Chapter 6. Text Feature Engineering

Introduction

In preceding chapters, we've spent time assessing powerful techniques that enable the analysis of complex or challenging data. However, for the most difficult problems, the right technique will only get you so far.

The persistent challenge that deep learning and supervised learning try to solve for is that finding solutions often requires multiple big investments from the team in question. Under the old paradigm, one often has to perform specific preparation tasks, requiring time, specialist skills, and knowledge. Often, even the techniques used were domain-specific and/or data type-specific. This process, via which features are derived, is referred to as feature engineering.

Most of the deep learning algorithms which we've studied so far are intended to help find ways around needing to perform extensive feature engineering. However, at the same time, feature engineering continues to be seen as a hugely important skill for top-level ML practitioners. The following quotes come from leading Kaggle competitors, via David Kofoed Wind's contribution to the Kaggle blog:

> *"The features you use influence more than everything else the result. No algorithm alone, to my knowledge, can supplement the information gain given by correct feature engineering."*
>
> *--(Luca Massaron)*

> *"Feature engineering is certainly one of the most important aspects in Kaggle competitions and it is the part where one should spend the most time on. There are often some hidden features in the data which can improve your performance by a lot and if you want to get a good place on the leaderboard you have to find them. If you screw up here you mostly can't win anymore; there is always one guy who finds all the secrets. However, there are also other important parts, like how you formulate the problem. Will you use a regression model or classification model or even combine both or is some kind of ranking needed. This, and feature engineering, are crucial to achieve a good result in those competitions. There are also some competitions where (manual) feature engineering is not needed anymore; like in image processing competitions. Current state of the art deep learning algorithms can do that for you."*
>
> *--(Josef Feigl)*

There are a few key themes here; feature engineering is powerful and even a very small amount of feature engineering can have a big impact on one's classifiers. It is also frequently necessary to employ feature engineering techniques if one wishes to deliver the best possible results. Maximising the effectiveness of your machine learning algorithms requires a certain amount of both domain-specific and data type-specific knowledge (secrets).

One more quote:

"For most Kaggle competitions the most important part is feature engineering, which is pretty easy to learn how to do."

--(Tim Salimans)

Tim's not wrong; most of what you'll learn in this chapter is intuitive, effective tricks, and transformations. This chapter will introduce you to a few of the most effective and commonly-used preparation techniques applied to text and time series data, drawing from NLP and financial time series applications. We'll walk through how the techniques work, what one should expect to see, and how one can diagnose whether they're working as desired.

Text feature engineering

In preceding sections, we've discussed some of the methods by which we might take a dataset and extract a subset of valuable features. These methods have broad applicability but are less helpful when dealing with non-numerical/non-categorical data, or data that cannot be easily translated into numerical or categorical data. In particular, we need to apply different techniques when working with text data.

The techniques that we'll study in this section fall into two main categories—cleaning techniques and feature preparation techniques. These are typically implemented in roughly that order and we'll study them accordingly.

Cleaning text data

When we work with natural text data, a different set of approaches apply. This is because in real-world contexts, the idea of a naturally clean text dataset is pretty unsafe; text data is rife with misspellings, non-dictionary constructs like emoticons, and in some cases, HTML tagging. As such, we need to be very thorough with our cleaning.

In this section, we'll work with a number of effective text-cleaning techniques, using a pretty gnarly real-world dataset. Specifically, we'll be using the Impermium dataset from a 2012 Kaggle competition, where the competition's goal was to create a model which accurately detects insults in social commentary.

Yes, I do mean Internet troll detection.

Let's get started!

Text cleaning with BeautifulSoup

Our first step should be manually checking the input data. This is pretty critical; with text data, one needs to try and understand what issues exist in the data initially so as to identify the cleaning needed.

It's kind of painful to read through a dataset full of hateful Internet commentary, so here's an example entry:

ID	Date	Comment
132	20120531031917Z	"""\xa0@Flip\xa0how are you not ded"""

We have an ID field and date field which don't seem to need much work. The text fields, however, are quite challenging. From this one case, we can already see misspelling and HTML inclusion. Furthermore, many entries in the dataset contain attempts to bypass swear filtering, usually by including a space or punctuation element mid-word. Other data quality issues include multiple vowels (to extend a word), non-ascii characters, hyperlinks... the list goes on.

One option for cleaning this dataset is to use regular expressions, which run over the input data to scrub out data quality issues. However, the quantity and variety of problem formats make it impractical to use a regex-based approach, at least to begin with. We're likely both to miss a lot of cases and also to misjudge the amount of preparation needed, leading us to clean too aggressively, or not aggressively enough; in specific terms we risk cutting into real text content or leaving parts of tags in place. What we need is a solution that will wash out the majority of common data quality problems to begin with so that we can focus on the remaining issues with a script-based approach.

Enter `BeautifulSoup`. `BeautifulSoup` is a very powerful text cleaning library which can, among other things, remove HTML markup. Let's take a look at this library in action on our troll data:

```
from bs4 import BeautifulSoup
import csv

trolls = []
with open('trolls.csv',  'rt') as f:
    reader = csv.DictReader(f)
    for line in reader:
        trolls.append(BeautifulSoup(str(line["Comment"]),
"html.parser"))

print(trolls[0])

eg = BeautifulSoup(str(trolls), "html.parser")

print(eg.get_text())
```

ID	Date	Comments
132	20120531031917Z	@Flip how are you not ded

As we can see, we've already made headway on improving the quality of our text data. Yet, it's also clear from these examples that there's a lot of work left to do! As discussed, let's move on to using regular expressions to help further clean and tokenize our data.

Managing punctuation and tokenizing

Tokenisation is the process of creating a set of tokens from a stream of text. Many tokens are words, while others might be character sets (such as smilies or other punctuation strings, for example, `????????`).

Now that we've removed a lot of the HTML ugliness from our initial dataset, we can take steps to further improve the cleanliness of our text data. To do this, we'll leverage the `re` module, which allows us to use operations over regular expressions, such as substring replacement. We'll perform a series of

operations over our input text on this pass, which mostly focus on replacing variable or problematic text elements with tokens. Let's begin with a simple example, replacing e-mail addresses with an `_EM` token:

```
text = re.sub(r'[\w\-][\w\-\.]+@[\w\-][\w\-\.]+[a-zA-Z]{1,4}',
'_EM', text)
```

Similarly, we can remove URLs, replacing them with the `_U` token:

```
text = re.sub(r'\w+:\/\/\S+', r'_U', text)
```

We can automatically remove extra or problematic whitespace and newline characters, hyphens, and underscores. In addition, we'll begin managing the problem of multiple characters, often used for emphasis in informal conversation. Extended series of punctuation characters are encoded here using codes such as `_BQ` and `BX`; these longer tags are used as a means of differentiating from the more straightforward `_Q` and `_X` tags (which refer to the use of a question mark and exclamation mark, respectively).

We can also use regular expressions to manage extra letters; by cutting down such strings to two characters at most, we're able to reduce the number of combinations to a manageable amount and tokenize that reduced group using the `_EL` token:

```
# Format whitespaces
text = text.replace('"', ' ')
text = text.replace('\'', ' ')
text = text.replace('_', ' ')
text = text.replace('-', ' ')
text = text.replace('\n', ' ')
text = text.replace('\\n', ' ')
text = text.replace('\'', ' ')
text = re.sub(' +',' ', text)
text = text.replace('\'', ' ')


#manage punctuation
text = re.sub(r'([^!\?])(\?{2,})(\Z|[^!\?])', r'\1 _BQ\n\3', text)
text = re.sub(r'([^\.])(\.{2,})', r'\1 _SS\n', text)
text = re.sub(r'([^!\?])(\?|!){2,}(\Z|[^!\?])', r'\1 _BX\n\3', text)
text = re.sub(r'([^!\?])\?(\Z|[^!\?])', r'\1 _Q\n\2', text)
text = re.sub(r'([^!\?])!(\Z|[^!\?])', r'\1 _X\n\2', text)
text = re.sub(r'([a-zA-Z])\1\1+(\w*)', r'\1\1\2 _EL', text)
text = re.sub(r'([a-zA-Z])\1\1+(\w*)', r'\1\1\2 _EL', text)
text = re.sub(r'(\w+)\.(\w+)', r'\1\2', text)
text = re.sub(r'[^a-zA-Z]','', text)
```

Next, we want to begin creating other tokens of interest. One of the more helpful indicators available is the `_SW` token for swearing. We'll also use regular expressions to help identify and tokenize smileys into

one of four buckets; big and happy smileys (`_BS`), small and happy ones (`_S`), big and sad ones (`_BF`), and small and sad ones (`_F`):

```
text = re.sub(r'([#%&\*\$]{2,})(\w*)', r'\1\2 _SW', text)


text = re.sub(r' [8x;:=]-?(?:\)|\}|\]|>){2,}', r' _BS', text)
text = re.sub(r' (?:[;:=]-?[\)\}\]d>])|(?:<3)', r' _S', text)
text = re.sub(r' [x:=]-?(?:\(|\[|\||\\|/|\{|<){2,}', r' _BF', text)
text = re.sub(r' [x:=]-?[\(\[\|\\/\{<]', r' _F', text)
```

Note

Smileys are complicated by the fact that their uses change frequently; while this series of characters is reasonably current, it's by no means complete; for example, see emojis for a range of non-ascii representations. For several reasons, we'll be removing non-ascii text from this example (a similar approach is to use a dictionary to force compliance), but both approaches have the obvious drawback that they remove cases from the dataset, meaning that any solution will be imperfect. In some cases, this approach may lead to the removal of a substantial amount of data. In general, then, it's prudent to be aware of the general challenge around character-based images in text content.

Following this, we want to begin splitting text into phrases. This is a simple application of `str.split`, which enables the input to be treated as a vector of words (words) rather than as long strings (`re`):

```
phrases = re.split(r'[;:\.()\n]', text)
phrases = [re.findall(r'[\w%\*&#]+', ph) for ph in phrases]
phrases = [ph for ph in phrases if ph]

words = []

for ph in phrases:
      words.extend(ph)
```

This gives us the following:

ID	Date	Comments
132	20120531031917Z	[['Flip', 'how', 'are', 'you', 'not', 'ded']]

Next, we perform a search for single-letter sequences. Sometimes, for emphasis, Internet communication involves the use of spaced single-letter chains. This may be attempted as a method of avoiding curse word detection:

```
tmp = words
words = []
```

```
new_word = ''
for word in tmp:
   if len(word) == 1:
      new_word = new_word + word
   else:
      if new_word:
         words.append(new_word)
         new_word = ''
      words.append(word)
```

So far, then, we've gone a long way toward cleaning and improving the quality of our input data. There are still outstanding issues, however. Let's reconsider the example we began with, which now looks like the following:

ID	Date	Words
132	20120531031917Z	['_F', 'how', 'are', 'you', 'not', 'ded']

Much of our early cleaning has passed this example by, but we can see the effect of vectorising the sentence content as well as the now-cleaned HTML tags. We can also see that the emote used has been captured via the `_F` tag. When we look at a more complex test case, we see even more substantial change results:

Raw	Cleaned and split
GALLUP DAILY\nMay 24-26, 2012 \u2013 Updates daily at 1 p.m. ET; reflects one-day change\nNo updates Monday, May 28; next update will be Tuesday, May 29.\nObama Approval48%-\nObama Disapproval45%-1\nPRESIDENTIAL ELECTION\nObama47%-\nRomney45%-\n7-day rolling average\n\n It seems the bump Romney got is over and the president is on his game.	['GALLUP', 'DAILY', 'May', 'u', 'Updates', 'daily', 'pm', 'ET', 'reflects', 'one', 'day', 'change', 'No', 'updates', 'Monday', 'May', 'next', 'update', 'Tuesday', 'May', 'Obama', 'Approval', 'Obama', 'Disapproval', 'PRESIDENTIAL', 'ELECTION', 'Obama', 'Romney', 'day', 'rolling', 'average', 'It', 'seems', 'bump', 'Romney', 'got', 'president', 'game']

However, there are two significant problems still obvious in both examples. In the first case, we have a misspelled word; we need to find a way to eliminate this. Secondly, a lot of the words in both examples (for example. are, pm) aren't terribly informative in and of themselves. The problem we find, particularly for shorter text samples, is that what's left after cleaning may contain only one or two

meaningful terms. If these terms are not terribly common in the corpus as a whole, it can prove to be very difficult to train a classifier to recognise these terms' significance.

Tagging and categorising words

I expect that we all know that English language words come in several types—nouns, verbs, adverbs, and so on. These are commonly referred to as **parts of speech**. If we know that a certain word is an adjective, as opposed to a verb or stop word (such as a, the, or of), we can treat it differently or more importantly, our algorithm can!

If we can perform part of speech tagging by identifying and encoding word classes as categorical variables, we're able to improve the quality of our data by retaining only the valuable content. The full range of text tagging options and techniques is too broad to be effectively covered in one section of this chapter, so we'll look at a subset of the applicable tagging techniques. Specifically, we'll focus on n-gram tagging and backoff taggers, a pair of complimentary techniques that allow us to create powerful recursive tagging algorithms.

We'll be using a Python library called the **Natural Language Toolkit** (**NLTK**). NLTK offers a wide array of functionality and we'll be relying on it at several points in this chapter. For now, we'll be using NLTK to perform tagging and removal of certain word types. Specifically, we'll be filtering out stop words.

To answer the obvious first question (why eliminate stop words?), it tends to be true that stop words add little to nothing to most text analysis and can be responsible for a degree of noise and training variance. Fortunately, filtering stop words is pretty straightforward. We'll simply import NLTK, download and import the dictionaries, then perform a scan over all words in our pre-existing word vector, removing any stop words found:

```
import nltk
nltk.download()
from nltk.corpus import stopwords

words = [w for w in words if not w in stopwords.words("english")]
```

I'm sure you'll agree that this was pretty straightforward! Let's move on to discuss more NLTK functionality, specifically, tagging.

Tagging with NLTK

Tagging is the process of identifying parts of speech, as we described previously, and applying tags to each term.

In its simplest form, tagging can be as straightforward as applying a dictionary over our input data, just as we did previously with stopwords:

```
tagged = ntlk.word_tokenize(words)
```

However, even brief consideration will make it obvious that our use of language is a lot more complicated than this allows. We may use a word (such as ferry) as one of several parts of speech and it may not be straightforward to decide how to treat each word in every utterance. A lot of the time, the correct tag can only be understood contextually given the other words and their positioning within the phrase.

Thankfully, we have a number of useful techniques available to help us solve linguistic challenges.

Sequential tagging

A sequential tagging algorithm is one that works by running through the input dataset, left-to-right and token-by-token (hence sequential!), tagging each token in succession. The decision over which token to assign is made based on that token, the tokens that preceded it, and the predicted tags for those preceding tokens.

In this section, we'll be using an **n-gram tagger**. An n-gram tagger is a type of sequential tagger, which is pretrained to identify appropriate tags. The n-gram tagger takes *(n-1)-many* preceding POS tags and the current token into consideration in producing a tag.

Note

For clarity, an n-gram is the term used for a contiguous sequence of n-many elements from a given set of elements. This may be a contiguous sequence of letters, words, numerical codes (for example, for state changes), or other elements. N-grams are widely used as a means of capturing the conjunct meaning of sets of elements—be those phrases or encoded state transitions—using n-many elements.

The simplest form of n-gram tagger is one where $n = 1$, referred to as a **unigram tagger**. A unigram tagger operates quite simply, by maintaining a conditional frequency distribution for each token. This conditional frequency distribution is built up from a training corpus of terms; we can implement training using a helpful train method belonging to the `NgramTagger` class in NLTK. The tagger assumes that the tag which occurs most frequently for a given token in a given sequence is likely to be the correct tag for that token. If the term **carp** is in the training corpus as a noun four times and as a verb twice, a unigram tagger will assign the noun tag to any token whose type is carp.

This might suffice for a first-pass tagging attempt but clearly, a solution that only ever serves up one tag for each set of homonyms isn't always going to be ideal. The solution we can tap into is using n-grams with a larger value of n. With $n = 3$ (a **trigram tagger**), for instance, we can see how the tagger might more easily distinguish the input *He tends to carp on a lot* from *He caught a magnificent carp*!

However, once again there is a trade-off here between accuracy of tagging and ability to tag. As we increase n, we're creating increasingly long n-grams which become increasingly rare. In a very short time, we end up in a situation where our n-grams are not occurring in the training data, causing our tagger to be unable to find any appropriate tag for the current token!

In practice, we find that what we need is a set of taggers. We want our most reliably accurate tagger to have the first shot at trying to tag a given dataset and, for any case that fails, we're comfortable with having a more reliable but potentially less accurate tagger have a try.

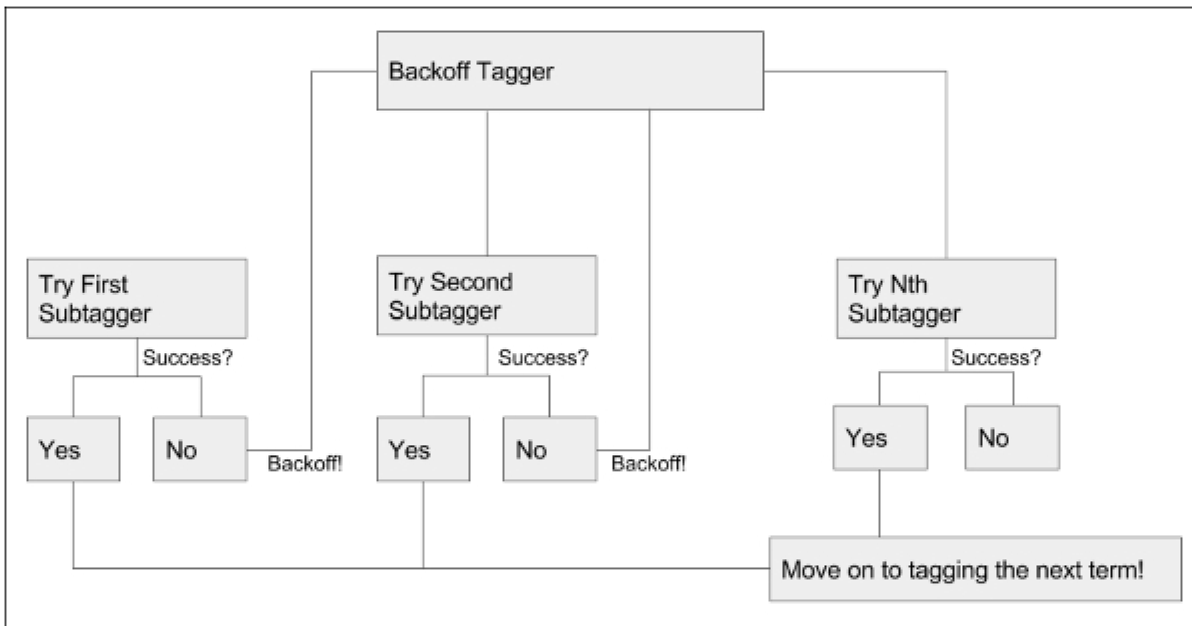
Happily, what we want already exists in the form of backoff tagging. Let's find out more!

Backoff tagging

Sometimes, a given tagger may not perform reliably. This is particularly common when the tagger has high accuracy demands and limited training data. At such times, we usually want to build an ensemble structure that lets us use several taggers simultaneously.

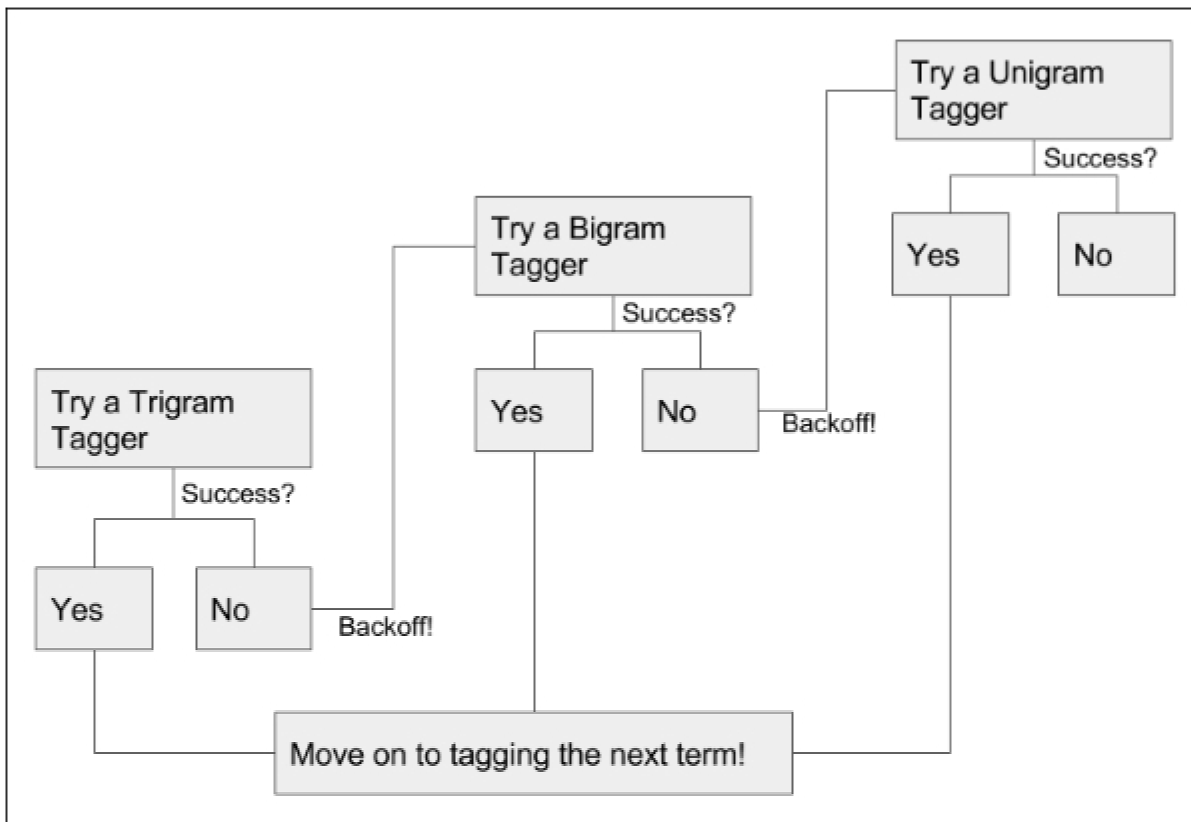
To do this, we create a distinction between two types of taggers: **subtaggers** and **backoff taggers**. Subtaggers are taggers like the ones we saw previously, **sequential** and **Brill taggers**. Tagging structures may contain one or multiple of each kind of tagger.

If a subtagger is unable to determine a tag for a given token, then a backoff tagger may be referred to instead. A backoff tagger is specifically used to combine the results of an ensemble of (one or more) subtaggers, as shown in the following example diagram:



In simple implementations, the backoff tagger will simply poll the subtaggers in order, accepting the first none-null tag provided. If all subtaggers return null for a given token, the backoff tagger will assign a none tag to that token. The order can be determined.

Backoffs are typically used with multiple subtaggers of different types; this enables a data scientist to harness the benefits of multiple types of tagger simultaneously. Backoffs may refer to other backoffs as needed, potentially creating highly redundant or sophisticated tagging structures:



In general terms, backoff taggers provide redundancy and enable you to use multiple taggers in a composite solution. To solve our immediate problem, let's implement a nested series of n-gram taggers. We'll start with a trigram tagger, which will use a bigram tagger as its backoff tagger. If neither of these taggers has a solution, we'll have a unigram tagger as an additional backoff. This can be done very simply, as follows:

```
brown_a = nltk.corpus.brown.tagged_sents(categories= 'a')

tagger = None
for n in range(1,4):
  tagger = NgramTagger(n, brown_a, backoff = tagger)

words  = tagger.tag(words)
```

Creating features from text data

Once we've engaged in well-thought-out text cleaning practices, we need to take additional steps to ensure that our text becomes useful features. In order to do this, we'll look at another set of staple techniques in NLP:

- Stemming
- Lemmatising

- Bagging using random forests

Stemming

Another challenge when working with linguistic datasets is that multiple word forms exist for many word stems. For example, the root dance is the stem of multiple other words—dancing, dancer, dances, and so on. By finding a way to reduce this plurality of forms into stems, we find ourselves able to improve our n-gram tagging and apply new techniques such as lemmatisation.

The techniques that enable us to reduce words to their stems are called stemmers. Stemmers work by parsing words as consonant/vowel strings and applying a series of rules. The most popular stemmer is the **porter stemmer**, which works by performing the following steps;

1. Simplifying the range of suffixes by reducing (for example, *ies* becomes *i*) to a smaller set.
2. Removing suffixes in several passes, with each pass removing a set of suffix types (for example, past particple or plural suffixes such as ousness or alism).
3. Once all suffixes are removed, cleaning up word endings by adding 'e's where needed (for example, ceas becomes cease).
4. Removing double 'l's.

The porter stemmer works very effectively. In order to understand exactly how well it works, let's see it in action!

```
from nltk.stem import PorterStemmer

stemmer = PorterStemmer()

stemmer.stem(words)
```

The output of this `stemmer`, as demonstrated on our pre-existing example, is the root form of the word. This may be a real word, or it may not; dancing, for instance, becomes danci. This is okay, but it's not really ideal. We can do better than this!

To consistently reach a real word form, let's apply a slightly different technique, lemmatisation. Lemmatisation is a more complex process to determine word stems; unlike porter stemming, it uses a different normalisation process for different parts of speech. Unlike Porter Stemming it also seeks to find actual roots for words. Where a stem does not have to be a real word, a lemma does. Lemmatization also takes on the challenge of reducing synonyms down to their roots. For example, where a stemmer might turn the term books into the term book, it isn't equipped to handle the term tome. A lemmatizer can process both books and tome, reducing both terms to book.

As a necessary prerequisite, we need the POS for each input token. Thankfully, we've already applied a POS tagger and can work straight from the results of that process!

```
from nltk.stem import PorterStemmer, WordNetLemmatizer

lemmatizer = WordNetLemmatizer()
```

```
words = lemmatizer.lemmatize(words, pos = 'pos')
```

The output is now what we'd expect to see:

Source Text	**Post-lemmatisation**
`The laughs you two heard were triggered by memories of his own high-flying exits off moving beasts`	`['The', 'laugh', 'two', 'hear', 'trigger', 'memory', 'high', 'fly', 'exit', 'move', 'beast']`

We've now successfully stemmed our input text data, massively improving the effectiveness of lookup algorithms (such as many dictionary-based approaches) in handling this data. We've removed stop words and tokenized a range of other noise elements with regex methods. We've also removed any HTML tagging. Our text data has reached a reasonably processed state. There's one more linchpin technique that we need to learn, which will let us generate features from our text data. Specifically, we can use bagging to help quantify the use of terms.

Let's find out more!

Bagging and random forests

Bagging is part of a family of techniques that may collectively be referred to as subspace methods. There are several forms of method, with each having a separate name. If we draw random subsets from the sample cases, then we're performing pasting. If we're sampling from cases with replacement, it's referred to as bagging. If instead of drawing from cases, we work with a subset of features, then we're performing attribute bagging. Finally, if we choose to draw from both sample cases and features, we're employing what's known as a **random patches** technique.

The feature-based techniques, attribute bagging, and Random Patch methods are very valuable in certain contexts, particularly very high-dimensional ones. Medical and genetics contexts both tend to see a lot of extremely high-dimensional data, so feature-based methods are highly effective within those contexts.

In NLP contexts, it's common to work with bagging specifically. In the context of linguistic data, what we'll be dealing with is properly called a bag of words. A bag of words is an approach to text data preparation that works by identifying all of the distinct words (or tokens) in a dataset and then counting their occurrence in each sample. Let's begin with a demonstration, performed over a couple of example cases from our dataset:

ID	**Date**	**Words**
`132`	`20120531031917Z`	`['_F', 'how', 'are', 'you', 'not', 'ded']`

ID	Date	Words
69	20120531173030Z	['you', 'are', 'living', 'proof', 'that', 'bath', 'salts', 'effect', 'thinking']

This gives us the following 12-part list of terms:

```
[
  "_F"
  "how"
  "are"
  "you"
  "not"
  "ded"
  "living"
  "proof"
  "that"
  "bath"
  "salts"
  "effect"
  "thinking"
]
```

Using the indices of this list, we can create a 12-part vector for each of the preceding sentences. This vector's values are filled by traversing the preceding list and counting the number of times each term occurs for each sentence in the dataset. Given our pre-existing example sentences and the list we created from them, we end up creating the following bags:

ID	Date	Comment	Bag of words
132	20120531031917Z	_F how are you not ded	[1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0]
69	20120531173030Z	you are living proof that bath salts effect thinking	[0, 0, 1, 1, 0, 0, 1, 1, 1, 1, 1, 1, 1]

This is the core of a bag of words implementation. Naturally, once we've translated the linguistic content of text into numerical vectors, we're able to start using techniques that add sophistication to how we use this text in classification.

One option is to use weighted terms. We can use a term weighting scheme to modify the values within each vector so that terms that are indicative or helpful for classification are emphasized. Weighting schemes may be straightforward masks, such as a binary mask that indicates presence versus absence.

Binary masking can be useful if certain terms are used much more frequently than normal; in such cases, specific scaling (for example, log-scaling) may be needed if a binary mask is not used. At the same time, though, frequency of term use can be informative (it may indicate emphasis, for instance) and the decision over whether to apply a binary mask is not always made simply.

Another weighting option is term frequency-inverse document frequency, or tf-idf. This scheme compares frequency of usage within a specific sentence and the dataset as a whole and uses values that increase if a term is used more frequently within a given sample than within the whole corpus.

Variations on tf-idf are frequently used in text mining contexts, including search engines. Scikit-learn provides a tf-idf implementation, `TfidfVectoriser`, which we'll shortly use to employ tf-idf for ourselves.

Now that we have an understanding of the theory behind bag of words and can see the range of technical options available to us once we develop vectors of word use, we should discuss how a bag of words implementation can be undertaken. Bag of words can be easily applied as a wrapper to a familiar model. While in general, subspace methods may use any of a range of base models (SVMs and linear regression models are common), it is very common to use random forests in a bag of words implementation, wrapping up preparation and learning into a concise script. In this case, we'll employ bag of words independently for now, saving classification via a random forest implementation for the next section!

Note

While we'll discuss random forests in greater detail in Chapter 8, *Ensemble Methods*, (which describes the various types of ensemble that we can create), it is helpful for now to note that a random forest is a set of decision trees. They are powerful ensemble models that are created either to run in parallel (yielding a vote or other net outcome) or boost one another (by iteratively adding a new tree to model the parts of the solution that the pre-existing set of trees couldn't model well).

Due to the power and ease of use of random forests, they are commonly used as benchmarking algorithms.

The process of implementing bag of words is, again, fairly straightforward. We initialize our bagging tool (matter-of-factly referred to as a vectorizer). Note that for this example, we're putting a limit on the size of the feature vector. This is largely to save ourselves some time; each document must be compared against each item in the feature list, so when we get to running our classifier this could take a little while!

```
from sklearn.feature_extraction.text import TfidfVectorizer

vectorizer = TfidfVectorizer(analyzer = "word",   \
                             tokenizer = None,    \
                             preprocessor = None, \
```

```
                                      stop_words = None,   \
                                      max_features = 5000)
```

Our next step is to fit the vectorizer on our word data via `fit_transform`; as part of the fitting process, our data is transformed into feature vectors:

```
train_data_features = vectorizer.fit_transform(words)

train_data_features = train_data_features.toarray()
```

This completes the pre-processing of our text data. We've taken this dataset through a full battery of text mining techniques, walking through the theory and reasoning behind each technique as well as employing some powerful Python scripts to process our test dataset.We're in a good position now to take a crack at Kaggle's insult detection challenge!

Testing our prepared data

So, now that we've done some initial preparation of the dataset, let's give the real problem a shot and see how we do. To help set the scene, let's consider Impermium's guidance and data description:

This is a single-class classification problem. The label is either 0 meaning a neutral comment, or 1 meaning an insulting comment (neutral can be considered as not belonging to the insult class. Your predictions must be a real number in the range [0,1] where 1 indicates 100% confident prediction that comment is an insult.

- *We are looking for comments that are intended to be insulting to a person who is a part of the larger blog/forum conversation.*
- *We are NOT looking for insults directed to non-participants (such as celebrities, public figures etc.).*
- *Insults could contain profanity, racial slurs, or other offensive language. But often times, they do not.*
- *Comments which contain profanity or racial slurs, but are not necessarily insulting to another person are considered not insulting.*
- *The insulting nature of the comment should be obvious, and not subtle.*
- *There may be a small amount of noise in the labels as they have not been meticulously cleaned. However, contestants can be confident the error in the training and testing data is < 1%.*

Contestants should also be warned that this problem tends to strongly overfit. The provided data is generally representative of the full test set, but not exhaustive by any measure. Impermium will be conducting final evaluations based on an unpublished set of data drawn from a wide sample.

This is pretty nice guidance, in that it raises two particular points of caution. The desired score is the **area under the curve** (**AUC**), which is a measure that is very sensitive both to false positives and to incorrect negative results (specificity and sensitivity).

The guidance clearly states that continuous predictions are desired rather than binary *0/1* outputs. This becomes critically important when using AUC; even a small amount of incorrect predictions given will radically decrease one's score if you only use categorical values. This suggests that rather than using the

`RandomForestClassifier` algorithm, we'll want to use the `RandomForestRegressor`, a regression-focused alternative, and then rescale the results between zero and one.

Real Kaggle contests are run in a much more challenging and realistic environment—one where the correct solution is not available. In Chapter 8, *Ensemble Methods*, we'll explore how top data scientists react and thrive in such environments. For now, we'll take advantage of having the ability to confirm whether we're doing well on the test dataset. Note that this advantage also presents a risk; if the problem overfits strongly, we'll need to be disciplined to ensure that we're not overtraining on the test data!

In addition, we also have the benefit of being able to see how well real contestants did. While we'll save the real discussion for Chapter 8, *Ensemble Methods*, it's reasonable to expect each highly-ranking contestant to have submitted quite a large number of failed attempts; having a benchmark will help us tell whether we're heading in the right direction.

Specifically, the top 14 participants on the private (test) leaderboard managed to reach an AUC score of over *0.8*. The top scorer managed a pretty impressive *0.84*, while over half of the 50 teams who entered scored above *0.77*.

As we discussed earlier, let's begin with a random forest regression model.

Note

A random forest is an ensemble of decision trees.

While a single decision tree is likely to suffer from variance- or bias-related issues, random forests are able to use a weighted average over multiple parallel trials to balance the results of modeling.

Random forests are very straightforward to apply and are a good first-pass technique for a new data challenge; applying a random forest classifier to the data early on enables you to develop a good understanding as to what initial, baseline classification accuracy will look like as well as giving valuable insight into how the classification boundaries were formed; during the initial stages of working with a dataset, this kind of insight is invaluable.

Scikit-learn provides `RandomForestClassifier` to enable the easy application of a random forest algorithm.

For this first pass, we'll use 100 trees; increasing the number of trees can improve classification accuracy but will take additional time. Generally speaking, it's sensible to attempt fast iteration in the early stages of model creation; the faster you can repeatedly run your model, the faster you can learn what your results are looking like and how to improve them!

We begin by initializing and training our model:

```
trollspotter = RandomForestRegressor(n_estimators = 100, max_depth =
10, max_features = 1000)

y = trolls["y"]
```

```
trollspotted = trollspotter.fit(train_data_features, y)
```

We then grab the test data and apply our model to predict a score for each test case. We rescale these scores using a simple stretching technique:

```
moretrolls = pd.read_csv('moretrolls.csv', header=True, names=['y',
'date', 'Comment', 'Usage'])
moretrolls["Words"] = moretrolls["Comment"].apply(cleaner)

y = moretrolls["y"]

test_data_features = vectorizer.fit_transform(moretrolls["Words"])
test_data_features = test_data_features.toarray()

pred = pred.predict(test_data_features)
pred = (pred - pred.min())/(pred.max() - pred.min())
```

Finally, we apply the `roc_auc` function to calculate an AUC score for the model:

```
fpr, tpr, _ = roc_curve(y, pred)
roc_auc = auc(fpr, tpr)
print("Random Forest benchmark AUC score, 100 estimators")
print(roc_auc)
```

As we can see, the results are definitely not at the level that we want them to be at:

```
Random Forest benchmark AUC score, 100 estimators
0.537894912105
```

Thankfully, we have a number of options that we can try to configure here:

- Our approach to how we work with the input (preprocessing steps and normalisation)
- The number of estimators in our random forest
- The classifier we choose to employ
- The properties of our bag of words implementation (particularly the maximum number of terms)
- The structure of our n-gram tagger

On our next pass, let's adjust the size of our bag of words implementation, increasing the term cap from a slightly arbitrary 5,000 to anywhere up to 8,000 terms; rather than picking just one value, we'll run over a range and see what we can learn. We'll also increase the number of trees to a more reasonable number (in this case, we stepped up to `1000`):

```
Random Forest benchmark AUC score, 1000 estimators
0.546439310772
```

These results are slightly better than the previous set, but not dramatically so. They're definitely a fair distance from where we want to be! Let's go further and set up a different classifier. Let's try a fairly familiar option—the SVM. We'll set up our own SVM object to work with:

```
class SVM(object):

    def __init__(self, texts, classes, nlpdict=None):

        self.svm = svm.LinearSVC(C=1000, class_weight='auto')
        if nlpdict:
            self.dictionary = nlpdict
        else:
            self.dictionary = NLPDict(texts=texts)
        self._train(texts, classes)

    def _train(self, texts, classes):
        vectors = self.dictionary.feature_vectors(texts)
        self.svm.fit(vectors, classes)

    def classify(self, texts):
        vectors = self.dictionary.feature_vectors(texts)
        predictions = self.svm.decision_function(vectors)
        predictions = p.transpose(predictions)[0:len(predictions)]
        predictions = predictions / 2 + 0.5
        predictions[predictions > 1] = 1
        predictions[predictions < 0] = 0
        return predictions
```

While the workings of `SVM` are almost impenetrable to human assessment, as an algorithm it operates effectively, iteratively translating the dataset into multiple additional dimensions in order to create complex hyperplanes at optimal class boundaries. It isn't a huge surprise, then, to see that the quality of our classification has increased:

```
SVM AUC score
0.625245653817
```

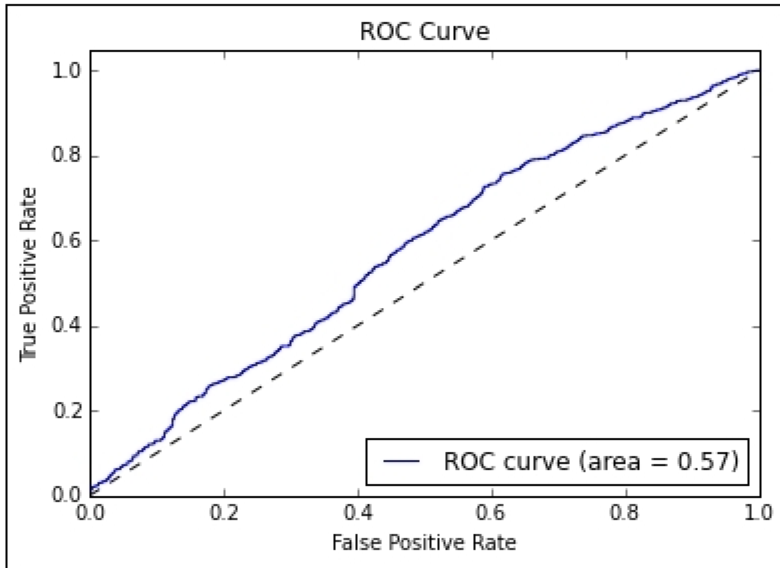
Perhaps we're not getting enough visibility into what's happening with our results. Let's try shaking things up with a different take on performance measurement. Specifically, let's look at the difference between the model's label predictions and actual targets to see whether the model is failing more frequently with certain types of input.

So we've taken our prediction quite far. While we still have a number of options on the table, it's worth considering the use of a more sophisticated ensemble of models as being a solid option. In this case, leveraging multiple models instead of just one can enable us to obtain the relative advantages of each. To try out an ensemble against this example, run the `score_trolls_blendedensemble.py` script.

Note

This ensemble is a blended/stacked ensemble. We'll be spending more time discussing how this ensemble works in Chapter 8, *Ensemble Methods*!

Plotting our results, we can see that performance has improved, but by significantly less than we'd hoped:



We're clearly having some issues with building a model against this data, but at this point, there isn't a lot of value in throwing a more developed model at the problem. We need to go back to our features and aim to extend the feature set.

At this point, it's worth taking some pointers from one of the most successful entrants of this particular Kaggle contest. In general, top-scoring entries tend to be developed by finding all of the tricks around the input data. The second-place contestant in the official Kaggle contest that this dataset was drawn from was a user named tuzzeg. This contestant provided a usable code repository at https://github.com/tuzzeg/detect_insults.

Tuzzeg's implementation differs from ours by virtue of much greater thoroughness. In addition to the basic features that we built using POS tagging, he employed POS-based bigrams and trigrams as well as subsequences (created from sliding windows of N-many terms). He worked with n-grams up to 7-grams and created character n-grams of lengths 2, 3, and 4.

Furthermore, tuzzeg took the time to create two types of composite model, both of which were incorporated into his solution—sentence level and ranking models. Ranking took our rationalization around the nature of the problem a step further by turning the cases in our data into ranked continuous values.

Meanwhile, the innovative sentence-level model that he developed was trained specifically on single-sentence cases in the training data. For prediction on test data, he split the cases into sentences, evaluated each separately, and took only the highest score for sentences within the case. This was to accommodate the expectation that in natural language, speakers will frequently confine insulting comments to a single part of their speech.

Tuzzeg's model created over 100 feature groups (where a stem-based bigram is an example feature group—a group in the sense that the bigram process creates a vector of features), with the most important ones (ranked by impact) being the following:

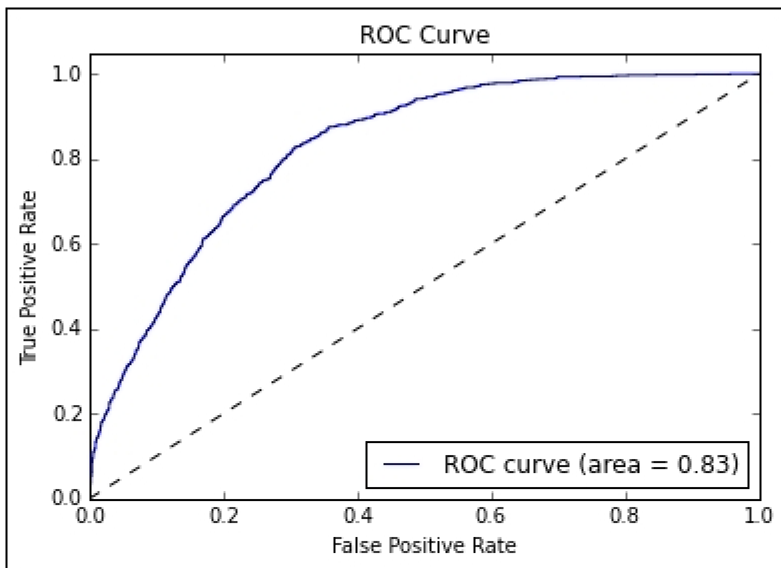
```
stem subsequence based         0.66
stem based (unigrams, bigrams) 0.18
char ngrams based (sentence)   0.07
char ngrams based              0.04
all syntax                     0.006
all language models            0.004
all mixed                      0.002
```

This is interesting, in that it suggests that a set of feature translations that we aren't currently using is important in generating a usable solution. Particularly, the subsequence-based features are only a short step from our initial feature set, making it straightforward to add the extra feature:

```
def subseq2(n, xs):
  l = len(xs)
  return ['%s %s' % (xs[i], xs[j]) for i in xrange(l-1) for j in
xrange(i+1, i+n+1) if j < l]

def getSubseq2(seqF, n):
  def f(row):
    seq = seqF(row)
    return set(seq + subseq2(n, seq))
  return f

Subseq2test = getSubseq2(line, 2)
```

This approach yields excellent results. While I'd encourage you to export Tuzzeg's own solution and apply it, you can also look at the `score_trolls_withsubseq.py` script provided in this project's repository to get a feeling for how powerful additional features can be incorporated.

With these additional features added, we see a dramatic improvement in our AUC score:



Running this code provides a very healthy `0.834` AUC score. This simply goes to show the power of thoughtful and innovative feature engineering; while the specific features generated in this chapter will serve you well in other contexts, specific hypotheses (such as hostile comments being isolated to specific sentences within a multi-sentence comment) can lead to very effective features.

As we've had the luxury of checking our reasoning against test data throughout this chapter, we can't reasonably say that we've worked under life-like conditions. We didn't take advantage of having access to the test data by reviewing it ourselves, but it's fair to say that knowing what the private leaderboard scored for this challenge made it easier for us to target the right fixes. In Chapter 8, *Ensemble Methods*, we'll be working on another tricky Kaggle problem in a more rigorous and realistic way. We'll also be discussing ensembles in depth!

Further reading

The quotes at the start of this chapter were sourced from the highly-readable Kaggle blog, No Free Hunch. Refer to http://blog.kaggle.com/2014/08/01/learning-from-the-best/.

There are many good resources for understanding NLP tasks. One fairly thorough, eight-part piece, is available online at http://textminingonline.com/dive-into-nltk-part-i-getting-started-with-nltk.

If you're keen to get started, one great option is to try Kaggle's for Knowledge NLP task, which is perfectly suited as a testbed for the techniques described in this chapter: https://www.kaggle.com/c/word2vec-nlp-tutorial/details/part-1-for-beginners-bag-of-words.

The Kaggle contest cited in this chapter is available at https://www.kaggle.com/c/detecting-insults-in-social-commentary.

For readers interested in further description of the ROC curve and the AUC measure, consider Tom Fawcett's excellent introduction, available at https://ccrma.stanford.edu/workshops/mir2009/references/ROCintro.pdf.

Summary

We've been introduced to a lot of useful and highly applicable skills in this chapter. In this chapter, we took a set of messy, complication-strewn text data and, through a series of rigorous steps, turned it into a large set of effective features. We began by picking up a set of data cleaning skills which stripped out a lot of the noise and problem elements, then we followed up by turning text into features using POS tagging and bag of words. In the process, you learned to apply a set of techniques that are widely applicable and very empowering, enabling us to solve difficult problems in many natural language processing contexts.

Through experimentation with multiple individual models and ensembles, we discovered that where a smarter algorithm might not yield a strong result, thorough and creative feature engineering can yield massive improvements in model performance.

Chapter 7. Feature Engineering Part II

Introduction

We have recognized the importance of feature engineering. In the previous chapter, we discussed techniques that enable us to select from a range of features and work effectively to transform our original data into features, which can be effectively processed by the advanced ML algorithms that we have discussed thus far.

The adage *garbage in, garbage out* is relevant in this context. In earlier chapters, we have seen how image recognition and NLP tasks require carefully-prepared data. In this chapter, we will be looking at a more ubiquitous type of data: quantitative or categorical data that is collected from real-world applications.

Data of the type that we will be working with in this chapter is common to many contexts. We could be discussing telemetry data captured from sensors in a forest, game consoles, or financial transactions. We could be working with geological survey information or bioassay data collected through research. Regardless, the core principles and techniques remain the same.

In this chapter, you will be learning how to interrogate this data to weed out or mitigate quality issues, how to transform it into forms that are conducive to machine learning, and how to creatively enhance that data.

In general terms, the concepts that we'll be discussing in this chapter are as follows:

- The different approaches to feature set creation and the limits of feature engineering
- How to use a large set of techniques to enhance and improve an initial dataset
- How to tie in and use domain knowledge to understand valid options to transform and improve the clarity of existing data
- How we can test the value of individual features and feature combinations so that we only keep what we need

While we will begin with a detailed discussion of the underlying concepts, by the end of this chapter we will be working with multiple, iterative trials and using specialized tests to understand how helpful the features that we are creating will be to us.

Creating a feature set

The most important factor involved in successful machine learning is the quality of your input data. A good model with misleading, inappropriately normalized, or uninformative data will not see the same level of success anywhere near a model run over appropriately prepared data.

In some cases, you have the ability to specify data collection or have access to a useful, sizeable, and varied set of source data. With the right knowledge and skillset, you can use this data to create highly useful feature sets.

In general, having a strong knowledge as to how to construct good feature sets is very helpful as it enables you to audit and assess any new dataset for missed opportunities. In this chapter, we will introduce a design process and technique set that make it easier to create effective feature sets.

As such, we'll begin by discussing some techniques that we can use to extend or reinterpret existing features, potentially creating a large number of useful parameters to include in our models.

However, as we will see, there are limitations on the effective use of feature engineering techniques and we need to be mindful of the risks around engineered datasets.

Engineering features for ML applications

We have discussed what you can do about patching up data quality issues in your data and we have talked about how you can creatively use dimensions in what you have to join to external data.

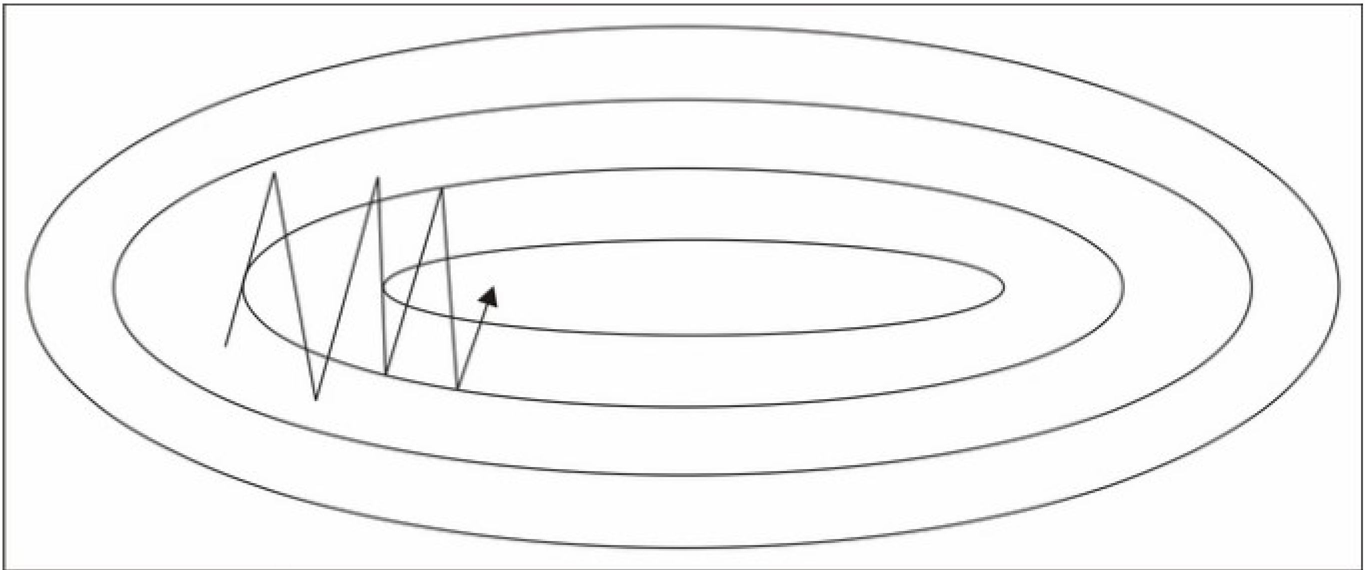
Once you have a reasonably well-understood and quality-checked set of data in front of you, there is usually still a significant amount of work needed before you can produce effective models from that data.

Using rescaling techniques to improve the learnability of features

The main challenge with directly feeding unprepared data to many machine learning models is that the algorithm is sensitive to the relative size of different variables. If your dataset has multiple parameters whose ranges differ, some algorithms will treat the variables whose variance is greater as indicative of more significant change than algorithms with smaller values and less variance.

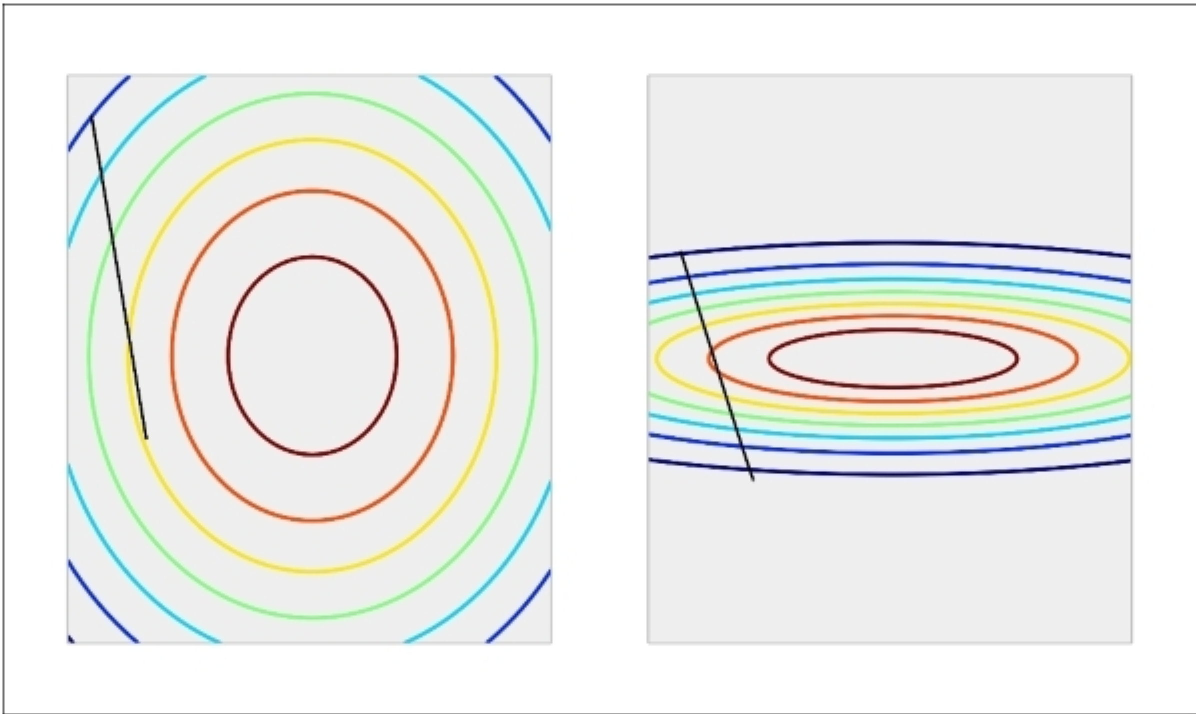
The key to resolving this potential problem is rescaling, a process by which parameter values' relative size is adjusted while retaining the initial ordering of values within each parameter (a monotonic translation).

Gradient descent algorithms (which include most deep learning algorithms—http://sebastianruder.com/optimizing-gradient-descent/) are significantly more efficient if the input data is scaled prior to training. To understand why, we'll resort to drawing some pictures. A given series of training steps may appear as follows:



When applied to unscaled data, these training steps may not converge effectively (as per the left-hand example in the following diagram).

With each parameter having a differing scale, the parameter space in which models are attempting to train can be highly distorted and complex. The more complex this space, the harder it becomes to train a model within it. This is an involved subject that can be effectively described, in general terms, through a metaphor, but for readers looking for a fuller explanation there is an excellent reference in this chapter's *Further reading* section. For now, it is not unreasonable to think in terms of gradient descent models during training as behaving like marbles rolling down a slope. These marbles are prone to *getting stuck* in saddle points or other complex geometries on the slope (which, in this context, is the surface created by our model's objective function—the learning function whose output our models typically train to minimize). With scaled data, however, the surface becomes more regularly-shaped and training can become much more effective:



The classic example is a linear rescaling between *0* and *1*; with this method, the largest parameter value is rescaled to *1*, the smallest to *0*, with intermediate values falling in the *0-1* interval, proportionate to their original size relative to the largest and smallest values. Under such a transformation, the vector *[0,10,25,20,18]*, for instance, would become *[0,0.4, 1, 0.8, 0.72]*.

The particular value of this transformation is that, for multiple data points that may vary in magnitude in its raw form, the rescaled features will sit within the same range, enabling your machine learning algorithm to train on meaningful information content.

This is the most straightforward rescaling option, but there are some nonlinear scaling alternatives that can be much more helpful in the right circumstances; these include square scaling, square root scaling, and perhaps most commonly, log-scaling.

Log-scaling of parameter values is very common in physics and in contexts where the underlying data is frequently affected by a power law (for example, an exponential growth in *y* given a linear increase in *x*).

Unlike linear rescaling, log-scaling adjusts the relative spacing between data cases. This can be a double-edged sword. On the one hand, log-scaling handles outlying cases very well. Let's take an example dataset describing individual net wealth for members of a fictional population, described by the following summary statistics:

Statistic	Wealth
Min	1
First Quartile	42.5
Mean	3205433.343
Median	600
Third Quartile	1358
Max	10000000000

Prior to rescaling, this population is hugely skewed toward that single individual with absurd net worth. The distribution of cases per decile is as follows:

Range	Count of Cases
0 > 0.1	3060
0.1 > 0.2	0
0.2 > 0.3	0
0.3 > 0.4	0
0.4 > 0.5	0
0.5 > 0.6	0
0.6 > 0.7	0
0.7 > 0.8	0
0.8 > 0.9	0
0.9 > 1	1

After log-scaling, this distribution is far friendlier:

Range	Count of Cases
0 > 0.1	740
0.1 > 0.2	1633
0.2 > 0.3	544
0.3 > 0.4	141
0.4 > 0.5	0
0.5 > 0.6	1
0.6 > 0.7	0
0.7 > 0.8	1
0.8 > 0.9	0
0.9 > 1	1

We could've chosen to take scaling further and drawn out the first half of this distribution more by doing that. In this case, log-10 normalization significantly reduces the impact of these outlying values, enabling us to retain outliers in the dataset without losing detail at the lower end.

With this said, it's important to note that in some contexts, that same enhancement of clustered cases can enhance noise in variant parameter values and create the false impression of greater spacing between values. This tends not to negatively affect how log-scaling handles outliers; the impact is usually seen for groups of smaller-valued cases whose original values are very similar.

The challenges created by introducing nonlinearities through log-scaling are significant and in general, nonlinear scaling is only recommended for variables that you understand and have a nonlinear relationship or trend underlying them.

Creating effective derived variables

Rescaling is a standard part of preprocessing in many machine learning applications (for instance, almost all neural networks). In addition to rescaling, there are other preparatory techniques, which can improve model performance by strategically reducing the number of parameters input to the model. The most common example is of a derived measure that takes multiple existing data points and represents them within a single measure.

These are extremely prevalent; examples include acceleration (as a function of velocity values from two points in time), body mass index (as a function of height, weight, and age), and **price-earnings** (**P/E**) ratio for stock scoring. Essentially, any derived score, ratio, or complex measure that you ever encounter is a combination score formed from multiple components.

For datasets in familiar contexts, many of these pre-existing measures will be well-known. Even in relatively well-known areas, however, looking for new supporting measures or transformations using a mix of domain knowledge and existing data can be very effective. When thinking through derived measure options, some useful concepts are as follows:

- **Two variable combinations**: Multiplication, division, or normalization of the *n* parameter as a function of the *m* parameter.
- **Measures of change over time**: A classic example here is acceleration or 7D change in a measure. In more complex contexts, the slope of an underlying time series function can be a helpful parameter to work with instead of working directly with the current and past values.
- **Subtraction of a baseline**: Using a base expectation (a flat expectation such as the *baseline churn rate*) to recast a parameter in terms of that baseline can be a more immediately informative way of looking at the same variable. For the churn example, we could generate a parameter that describes churn in terms of deviation from an expectation. Similarly, in stock trading cases, we might look at closing price in terms of the opening price.
- **Normalization**: Following on from the previous case, normalization of parameter values based on the values of another parameter or baseline that is dynamically calculated given properties of other variables. One example here is *failed transaction rate*; in addition to looking at this value as a raw (or rescaled) count, it often makes sense to normalize this in terms of attempted transactions.

Creative recombination of these different elements lets us build very effective scores. Sometimes, for instance, a parameter that tells us the slope of customer engagement (declining or increasing) over time needs to be conditioned on whether that customer was previously highly engaged or hardly engaged, as a slight decline in engagement might mean very different things in each context. It is the data scientist's job to effectively and creatively feature sets that capture these subtleties for a given domain.

So far, this discussion has focused on numerical data. Often, however, useful data is locked up inside non-numeric parameters such as codes or categorical data. Accordingly, we will next discuss a set of effective techniques to turn non-numeric features into usable parameters.

Reinterpreting non-numeric features

A common challenge, which can be problematic and problem-specific, is how non-numeric features are treated. Frequently, valuable information is encoded within non-numerical shorthand values. In the case of stock trades, for instance, the identity of the stock itself (for example, AAPL) as well as that of the buyer and seller is interesting information that we expect to relate meaningfully to our problem. Taking this example further, we might also expect some stocks to trade differently from others even within the industry, and organizational differences within companies, which may occur at some or all points of time, also provide important context.

One simple option that works in some cases is building an aggregation or series of aggregations. The most obvious example is a count of occurrences with the possibility of creating extended measures (changes in count between two time windows) as described in the preceding section.

Building summary statistics and reducing the number of rows in the dataset introduces the risk of reducing the amount of information that your model has available to learn from (increasing the risk of model fragility and overfitting). As such, it's generally a bad idea to extensively aggregate and reduce

input data. This is doubly true with deep learning techniques, such as the algorithms discussed and used in Chapters 2-4.

Rather than extensively using aggregation-based approaches, let's look at an alternative way of translating string-encoded values into numerical data. Another very popular class of techniques is encoding, with the most common encoding tactic being one-hot encoding. One-hot encoding is the process of turning a series of categorical responses (for example, age groups) into a set of binary variables, with each response option (for example, 18-30) represented by its own binary variable. This is a little more intuitive when presented visually:

Case	Age	Gender
1	22	M
2	25	M
3	34	F
4	23	M
5	25	F
6	41	F

After encoding, this dataset of categorical and continuous variables becomes a tensor of binary variables:

Case	Age_22	Age_23	Age_25	Age_34	Age_41	Gender_F	Gender_M
1	1	0	0	0	0	0	1
2	0	0	1	0	0	0	1
3	0	0	0	1	0	1	0
4	0	1	0	0	0	0	1
5	0	0	0	0	0	1	0
6	0	0	0	0	1	1	0

The advantage that this presents is significant; it enables us to tap into the very valuable tag information contained within a lot of datasets without aggregation or risk of reducing the information content of the data. Furthermore, one-hot allows us to separate specific response codes for encoded variables into separate features, meaning that we can identify more or less meaningful codes for a specific variable and only retain the important values.

Another very effective technique, used primarily for text codes, is known as the **hash trick**. A hash, in simple terms, is a function that translates data into a numeric representation. Hashes will be a familiar concept to many, as they're frequently used to encode sensitive parameters and summarize otherwise bulky data. In order to get the most out of the hash trick, however, it's important to understand how the trick works and what can be done with it.

We can use hashing to turn a text phrase into a numeric value that we can use as an identifier for that phrase. While there are many applications of different hashing algorithms, in this context even a simple hash makes it straightforward to turn string keys and codes into numerical parameters that we can model effectively.

A very simple hash might turn each alphabet character into a corresponding number. *a* would become *1*, *b* would be *2*, and so on. Hashes could be generated for words and phrases by summing those values. The phrase *cat gifs* would translate under this scheme as follows:

```
Cat: 3 + 1 + 20
Gifs: 7 + 9 + 6 + 19
Total: 65
```

This is a terrible hash for two reasons (quite disregarding the fact that the input contains junk words!). Firstly, there's no real limit on how many outputs it can present. When one remembers that the whole point of the hash trick is to provide dimensionality reduction, it stands to reason that the number of possible outputs from a hash must be bounded! Most hashes limit the range of numbers that they output, so part of the decision in terms of selecting a hash is related to the number of features you'd prefer your model to have.

Note

One common behavior is to choose a power of two as the hash range; this tends to speed things up by allowing bitwise operations during the hashing process.

The other reason that this hash kind of sucks is that changes to the word have a small impact rather than a large one. If *cat* became *bat*, we'd want our hash output to change substantially. Instead, it changes by one (becoming *64*). In general, a good hash function is one where a small change in the input text will cause a large change in the output. This is partly because language structures tend to be very uniform (thus scoring similarly), but slightly different sets of nouns and verbs within a given structure tend to confer very different meanings to one another (*the cat sat on the mat* versus *the car sat on the cat*).

So we've described hashing. The hash trick takes things a little further. Hypothetically, turning every word into a hashed numerical code is going to lead to a large number of *hash collisions*—cases where two words have the same hash value. Naturally, these are rather bad.

Handily, there's a distribution underlying how frequently different terms are used that work in our favor. Called the **Zipf distribution**, it entails that the probability of encountering the nth most common term is approximated by $P(n) = 0.1/n$ up to around 1,000 (Zipf's law). This entails that each term is much less likely to be encountered than the preceding term. After $n = 1000$, terms tend to be sufficiently obscure that it's unlikely to encounter two that have the same hash in one dataset.

At the same time, a good hashing function has a limited range and is significantly affected by small changes in input. These properties make the hash collision chance largely independent of term usage frequency.

These two concepts—Zipf's law and a good hash's independence of hash collision chance and term usage frequency—mean that there is very little chance of a hash collision, and where one occurs it is overwhelmingly likely to be between two infrequently-used words.

This gives the hash trick a peculiar property. Namely, it is possible to reduce the dimensionality of a set of text input data massively (from tens of thousands of naturally occurring words to a few hundred or fewer) without reducing the performance of a model trained on hashed data, compared to training on unhashed bag-of-words features.

Proper use of the hash trick enables a lot of possibilities, including augmentations to the techniques that we discussed (specifically, bag-of-words). References to different hashing implementations are included in the *Further reading* section at the end of this chapter.

Using feature selection techniques

Now that we have a good selection of options for feature creation, as well as an understanding of the creative feature engineering possibilities, we can begin building our existing features into more effective variants. Given this new-found feature engineering skillset, we run the risk of creating extensive and hard-to-manage datasets.

Adding features without limit increases the risk of model fragility and overfitting for certain types of models. This is tied to the complexity of the trends that you're attempting to model. In the simplest case, if you're attempting to identify a significant distinction between two large groups, your model is likely to support a large number of features. However, as the model you need to fit to make this distinction becomes more complex and as the group sizes that you have to work with become smaller, adding more and more features can harm the model's ability to classify consistently and effectively.

This challenge is compounded by the fact that it isn't always obvious which parameter or variation is best-suited for the task. Suitability can vary by the underlying model; decision forests, for instance, don't perform any better with monotonic transformations (that is, transformations that retain the initial ordering of data cases; one example is log-scaling) than with the unscaled base data; however, for other algorithms, the choice to rescale and the rescaling method used are both very impactful choices.

Traditionally, the quantity of features and limits on the parameter amount were tied to the desire to develop a mathematical function that relates key inputs to the desired outcome scores. In this context, additional parameters needed to be incorporated as moving or nuisance variables.

Each new parameter introduces another dimension that makes the modeled relationship more complex and the resultant model more likely to be overfitting the data that exists. A trivial example is if you introduce a parameter that is just a unique label for each case; at this point, your algorithm will just learn those labels, making it very likely that your model fails entirely when introduced to a new dataset.

Less trivial examples are no less problematic; the proportion of cases to features becomes very important when your features are separating cases down to very small groups. In short, increasing the

complexity of the modeled function causes your model to be more liable to overfit and adding features can exacerbate this effect. According to this principle, we should be beginning with very small datasets and adding parameters only after justifying that they improve the model.

However, in recent times, an opposing methodology—now generally seen as being part of a common way of *doing data science*—has gained ground. This methodology suggests that it's a good idea to add very large feature sets to incorporate every potentially valuable feature and *work down* to a smaller feature set that does the job.

This methodology is supported by techniques that enable decisions to be made over huge feature sets (potentially hundreds or thousands of features) and that tend to operate in a *brute force* manner. These techniques will exhaustively test feature combinations, running models in series or in parallel until the most effective parameter subsets are identified.

These techniques work, which is why this methodology has become popular. It is definitely worth knowing about these techniques, if not using them, so you'll be learning how to apply them later in this chapter.

The main disadvantage around using brute force techniques for feature selection is that it becomes very easy to trust the outcomes of the algorithm, irrespective of what the features it selects actually mean. It is sensible to balance the use of highly effective, black-box algorithms against domain knowledge and an understanding of what's being undertaken. Therefore, this chapter will enable you to use techniques from both paradigms (*build up* and *build down*) so that you can adapt to different contexts. We'll begin by learning how to narrow down the feature set that you have to work with, from many features to the most valuable subset.

Performing feature selection

Having built a large dataset, often the next challenge one faces is how to narrow down the options to retain only the most effective data. In this section, we'll discuss a variety of techniques that support feature selection, working by themselves or as wrappers to familiar algorithms.

These techniques include correlation analysis, regularization techniques, and **Recursive Feature Elimination** (**RFE**). When we're done, you'll be able to confidently use these techniques to support your selection of feature sets, potentially saving yourself a significant amount of work every time you work with a new dataset!

Correlation

We'll begin our discussion of feature selection by looking for a simple source of major problems for regression models: multicollinearity. Multicollinearity is the fancy name for moderate or high degrees of correlation between features in a dataset. An obvious example is how pizza slice count is collinear with pizza price.

There are two types of multicollinearity: structural and data-based. Structural multicollinearity occurs when the creation of new features, such as feature *f1* from feature *f*, creates multiple features that may be highly correlated with one another. Data-based multicollinearity tends to occur when two variables are affected by the same causative factor.

Both kinds of multicollinearity can cause some unfortunate effects. In particular, our models' performance tends to become affected by which feature combinations are used; when collinear features are used, the performance of our model will tend to degrade.

In either case, our approach is simple: we can test for multicollinearity and remove underperforming features. Naturally, underperforming features are ones that add very little to model performance. They might be underperforming because they replicate information available in other features, or they may simply not provide data that is meaningful to the problem at hand. There are multiple ways to test for weak features as many feature selection techniques will sift out multicollinear feature combinations and recommend their removal if they're underperformant.

In addition, there is a specific multicollinearity test that's worth considering; namely, inspecting the eigenvalues of our data's correlation matrix. Eigenvectors and eigenvalues are fundamental concepts in the matrix theory with many prominent applications. More details are given at the end of this chapter. For now, suffice it to say that eigenvalues in the correlation matrix generated by our dataset provide us with a quantified measure of multicollinearity. Consider a set of eigenvalues as indicative of how much "new information content" our features bring to the dataset; a low eigenvalue suggests that the data may be correlated with other features. For an example of this at work, consider the following code, which creates a feature set and then adds collinearity to features *0*, *2*, and *4*:

```
import numpy as np

x = np.random.randn(100, 5)
noise = np.random.randn(100)
x[:,4] = 2 * x[:,0] + 3 * x[:,2] + .5 * noise
```

When we generate the correlation matrix and compute eigenvalues, we find the following:

```
corr = np.corrcoef(x, rowvar=0)
w, v = np.linalg.eig(corr)

print('eigenvalues of features in the dataset x')
print(w)

eigenvalues of features in the dataset x
[ 0.00716428  1.94474029  1.30385565  0.74699492  0.99724486]
```

Clearly, our *0th* feature is suspect! We can then inspect the eigenvalues of this feature via calling `v`:

```
print('eigenvalues of eigenvector 0')
print(v[:,0])

eigenvalues of eigenvector 0
[-0.35663659 -0.00853105 -0.62463305  0.00959048  0.69460718]
```

From the small values of features in position one and three, we can tell that features *2* and *4* are highly multicollinear with feature *0*. We ought to remove two of these three features before proceeding!

LASSO

Regularized methods are among the most helpful feature selection techniques as they provide sparse solutions: ones where weaker features return zero, leaving only a subset of features with real coefficient values.

The two most used regularization models are L1 and L2 regularization, referred to as LASSO and ridge regression respectively in linear regression contexts.

Regularized methods function by adding a penalty to the loss function. Instead of minimizing a loss function $E(X,Y)$, the penalty leads to $E(X,Y) + a||w||$. The hyperparameter a relates to the amount of regularization (enabling us to tune the strength of our regularization and thus the proportion of the original feature set that is selected).

In LASSO regularization, the specific penalty function used is $\alpha \sum ni=1|wi|$. Each non-zero coefficient adds to the size of the penalty term, forcing weaker features to return coefficients of 0. Selecting an appropriate penalty term can be achieved using scikit-learn's parameter optimization support for hyperparameters. In this case, we'll be using `estimator.get_params()` to perform a grid search for appropriate hyperparameter values. For more information on how grid searches operate, see the *Further reading* section at the end of this chapter.

In scikit-learn, logistic regression is provided with an L1 penalty for classification. Meanwhile, the LASSO module is provided for linear regression. For now, let's begin by applying LASSO to an example dataset. In this case, we'll use the Boston housing dataset:

```
fromsklearn.linear_model import Lasso
fromsklearn.preprocessing import StandardScaler
fromsklearn.datasets import load_boston

boston = load_boston()
scaler = StandardScaler()
X = scaler.fit_transform(boston["data"])
Y = boston["target"]
names = boston["feature_names"]

lasso = Lasso(alpha=.3)
lasso.fit(X, Y)

print "Lasso model: ", pretty_print_linear(lasso.coef_, names, sort
= True)

Lasso model: -3.707 * LSTAT + 2.992 * RM + -1.757 * PTRATIO + -1.081
* DIS + -0.7 * NOX + 0.631 * B + 0.54 * CHAS + -0.236 * CRIM + 0.081
* ZN + -0.0 * INDUS + -0.0 * AGE + 0.0 * RAD + -0.0 * TAX
```

Several of the features in the original set returned a correlation of `0.0`. Increasing the correlation makes the solution increasingly sparse. For instance, we see the following results when `alpha = 0.4`:

```
Lasso model: -3.707 * LSTAT + 2.992 * RM + -1.757 * PTRATIO + -1.081
* DIS + -0.7 * NOX + 0.631 * B + 0.54 * CHAS + -0.236 * CRIM + 0.081
* ZN + -0.0 * INDUS + -0.0 * AGE + 0.0 * RAD + -0.0 * TAX
```

We can immediately see the value of L1 regularization as a feature selection technique. However, it is important to note that L1 regularized regression is unstable. Coefficients can vary significantly, even with small data changes, when features in the data are correlated.

This problem is effectively addressed with L2 regularization, or ridge regression, which develops a feature coefficient with different applications. L2 normalization adds an additional penalty, the L2 norm penalty, to the loss function. This penalty takes the form ($a\sum_{i=1}^{n} w_i^2$). A sharp-eyed reader will notice that, unlike the L1 penalty ($\alpha\sum_{i=1}^{n}|w_i|$), the L2 penalty uses squared coefficients. This causes the coefficient values to be spread out more evenly and has the added effect that correlated features tend to receive similar coefficient values. This significantly improves stability as the coefficients no longer fluctuate on small data changes.

However, L2 normalization isn't as directly useful for feature selection as L1. Rather, as interesting features (with predictive power) tend to have non-zero coefficients, L2 is more useful as an exploratory tool allowing inference about the quality of features in the classification. It has the added merit of being more stable and reliable than L1 regularization.

Recursive Feature Elimination

RFE is a greedy, iterative process that functions as a wrapper over another model, such as an SVM (SVM-RFE), which it repeatedly runs over different subsets of the input data.

As with LASSO and ridge regression, our goal is to find the best-performing feature subset. As the name suggests, on each iteration a feature is set aside allowing the process to be repeated with the rest of the feature set until all features in the dataset have been eliminated. The ordering with which features are eliminated becomes their rank. After multiple iterations with incrementally smaller subsets, each feature is accurately scored and relevant subsets can be selected for use.

To get a better understanding of how this works, let's look at a simple example. We'll use the (by now familiar) digits dataset to understand how this approach works in practice:

```
print(__doc__)

from sklearn.svm import SVC
fromsklearn.datasets import load_digits
fromsklearn.feature_selection import RFE
importmatplotlib.pyplot as plt

digits = load_digits()
X = digits.images.reshape((len(digits.images), -1))
y = digits.target
```

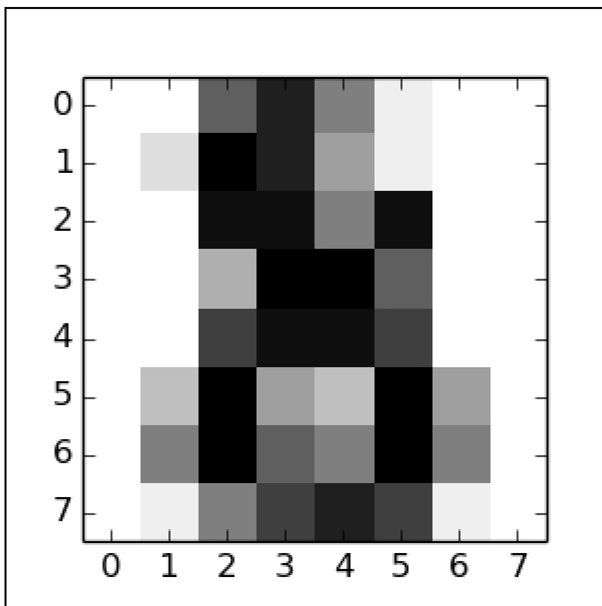
We'll use an SVM as our base estimator via the `SVC` operator for **Support Vector Classification** (**SVC**). We then apply the RFE wrapper over this model. RFE takes several arguments, with the first being a

reference to the estimator of choice. The second argument is `n_features_to_select`, which is fairly self-explanatory. In cases where the feature set contains many interrelated features whose subsets possess multivariate distributions that are highly effective classification features, it's possible to opt for feature combinations of two or more.

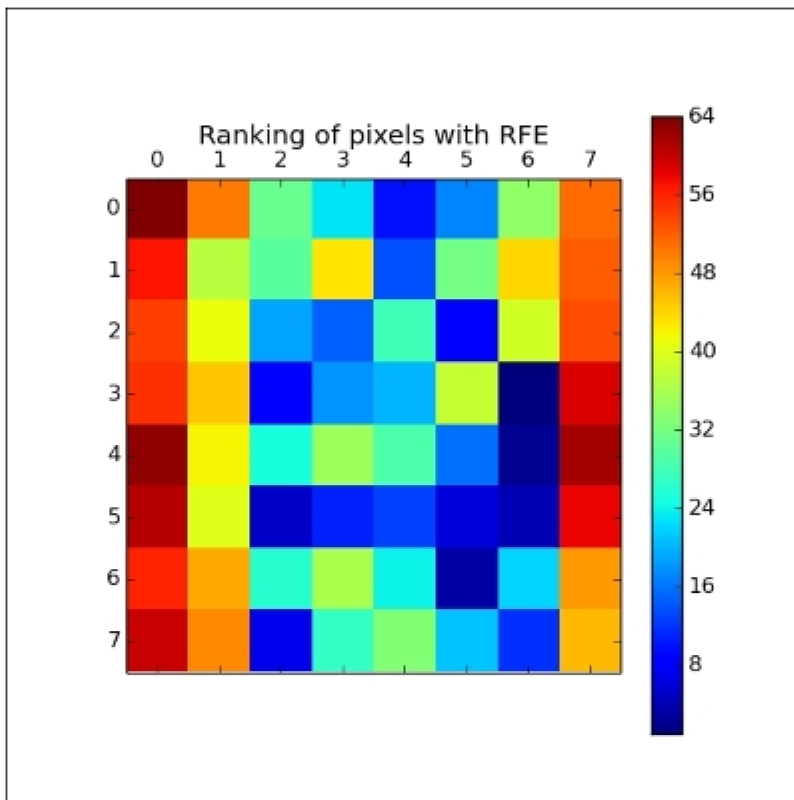
Stepping enables the removal of multiple features on each iteration. When given a value between *0.0* and *1.0*, each step enables the removal of a percentage of the feature set, corresponding to the proportion given in the step argument:

```
svc = SVC(kernel="linear", C=1)
rfe = RFE(estimator=svc, n_features_to_select=1, step=1)
rfe.fit(X, y)
ranking = rfe.ranking_.reshape(digits.images[0].shape)

plt.matshow(ranking)
plt.colorbar()
plt.title("Ranking of pixels with RFE")
plt.show()
```

Given that we're familiar with the digits dataset, we know that each instance is an 8 x 8 image of a handwritten digit, as shown in the following image. Each image is located in the center of the 8 x 8 grid:



When we apply RFE over the digits dataset, we can see that it broadly captures this information in applying a ranking:



The first pixels to be cut were in and around the (typically empty) vertical edges of the image. Next, the algorithm began culling normally whitespace areas around the vertical edges or near the top of the image. The pixels that were retained longest were those that enabled the most differentiation between the different characters—pixels that would be present for some numbers and not for others.

This example gives us great visual confirmation that RFE works. What it doesn't give us is evidence for how consistently the technique works. The stability of RFE is dependent on the stability of the base model and, in some cases, ridge regression will provide a more stable solution. (For more information on which cases and the conditions involved, consult the *Further reading* section at the end of this chapter.)

Genetic models

Earlier in this chapter, we discussed the existence of algorithms that enable feature selection with very large parameter sets. Some of the most prominent techniques of this type are genetic algorithms, which emulate natural selection to generate increasingly effective models.

A genetic solution for feature selection works roughly as follows:

- An initial set of variables (predictors is the term typically used in this context) are combined into multiple subsets (candidates) and a performance measure is calculated for each candidate
- The predictors from candidates with the best performance are randomly recombined into a new iteration (generation) of models

- During this recombination step, for each subset there is the probability of a mutation, whereby a predictor may be added or removed from a subset

This algorithm typically iterates for multiple generations. The appropriate iteration amount is dependent on the complexity of the dataset and the model required. As with gradient descent techniques, the typical relationship between the performance and iteration count is present for genetic algorithms, where performance improvement declines nonlinearly as the count of iterations increases, eventually hitting a minimum before the overfitting risk increases.

To find an effective iteration count, we can perform testing using training data; by running the model for a large number of iterations and plotting the **Root Mean Squared Error** (**RMSE**), we're able to find an appropriate amount of iterations given our input data and model configuration.

Let's talk in a little more detail about what happens within each generation. Specifically, let's talk about how candidates are created, how performance is scored, and how recombination is performed.

The candidates are initially configured to use a random sample of the available predictors. There is no hard and fast rule concerning how many predictors to use in the first generation; it depends on how many features are available, but it's common to see first generation candidates using 50% to 80% of the available features (with a smaller percentage used in cases with more features).

The fitness measure can be difficult to define, but a common practice is to use two forms of cross-validation. Internal cross-validation (testing each model solely in the context of its own parameters without comparing models) is typically used to track performance at a given iteration; the fitness measures from internal cross-validation are used to select models to recombine in the next generation. External cross-validation (testing against a dataset that was not used in validation at any iteration) is also needed in order to confirm that the search process produced a model that has not overfitted to the internal training data.

Recombination is controlled by three key parameters: mutation, cross-over probabilities, and elitism. The latter is an optional parameter that one may use to reserve n-many of the top-performing models from the current generation; by doing so, one may preserve particularly effective candidates from being lost entirely during recombination. This can be done while also using that candidate in mutated variants and/or using them as parents to next-generation candidates.

The mutation probability defines the chance of a next-generation model being randomly readjusted (via some predictors, typically one, being added or removed). Mutation tends to help the genetic algorithm maintain a broad coverage of the candidate variables, reducing the risk of falling into a parameter-local solution.

Cross-over probability defines the likelihood that a pair of candidates will be selected for recombination into a next-generation model. There are several cross-over algorithms: parts of each parent's feature set might be spliced (for example, first half/second half) into the child or a random selection of each parent's features might be used. Common features to both parents might also be used by default. Random sampling from the set of both parent's unique predictors is a common default approach.

These are the main parts of a general genetic algorithm, which can be used as a wrapper to existing models (logistic regression, SVM, and others). The technique described here can be varied in many

different ways and is related to feature selection techniques used slightly differently across multiple quantitative fields. Let's take the theory that we've covered thus far and start applying it to a practical example.

Feature engineering in practice

Depending on the modeling technique that you're using, some of this work may be more valuable than other parts. Deep learning algorithms tend to perform better on less-engineered data than shallower models and it might be that less work is needed to improve results.

The key to understanding what is needed is to iterate quickly through the whole process from dataset acquisition to modeling. On a first pass with a clear target for model accuracy, find the acceptable minimum amount of processing and perform that. Learn whatever you can about the results and make a plan for the next iteration.

To show how this looks in practice, we'll work with an unfamiliar, high-dimensional dataset, using an iterative process to generate increasingly effective modeling.

I was recently living in Vancouver. While it has many positive qualities, one of the worst things about living in the city was the somewhat unpredictable commute. Whether I was traveling by car or taking Translink's Skytrain system (a monorail-meets-rollercoaster high-speed line), I found myself subject to hard-to-predict delays and congestion issues.

In the spirit of putting our new feature engineering skillset into practice, let's take a look at whether we can improve this experience by taking the following steps:

- Writing code to harvest data from multiple APIs, including text and climate streams
- Using our feature engineering techniques to derive variables from this initial data
- Testing our feature set by generating commute delay risk scores

Unusually, in this example, we'll focus less on building and scoring a highly performant model. Instead, our focus is on creating a self-sufficient solution that you can adjust and apply for your own local area. While it suits the goals of the current chapter to take this approach, there are two additional and important motivations.

Firstly, there are some challenges around sharing and making use of Twitter data. Part of the terms of use of Twitter's API is an obligation on the developer to ensure that any adjustments to the state of a timeline or dataset (including, for instance, the deletion of a tweet) are reproduced in datasets that are extracted from Twitter and publicly shared. This makes the inclusion of real Twitter data in this chapter's GitHub repository impractical. Ultimately, this makes it difficult to provide reproducible results from any downstream model based on streamed data because users will need to build their own stream and accumulate data points and because variations in context (such as seasonal variations) are likely to affect model performance.

The second element here is simple enough: not everybody lives in Vancouver! In order to generate something of value to an end user, we should think in terms of an adjustable, general solution rather than a geographically-specific one.

The code presented in the next section is therefore intended to be something to build from and develop. It offers potential as the basis of a successful commercial app or simply a useful, data-driven life hack. With this in mind, review this chapter's content (and leverage the code in the associated code directory)

with an eye to finding and creating new applications that fit your own situation, locally available data, and personal needs.

Acquiring data via RESTful APIs

In order to begin, we're going to need to collect some data! We're going to need to look for rich, timestamped data that is captured at sufficient frequency (preferably at least one record per commute period) to enable model training.

A natural place to begin with is the Twitter API, which allows us to harvest recent tweet data. We can put this API to two uses.

Firstly, we can harvest tweets from official transit authorities (specifically, bus and train companies). These companies provide transit service information on delays and service disruptions that, helpfully for us, takes a consistent format conducive to tagging efforts.

Secondly, we can tap into commuter sentiment by listening for tweets from the geographical area of interest, using a customized dictionary to listen for terms related to cases of disruption or the causes thereof.

In addition to mining the Twitter API for data to support our model, we can leverage other APIs to extract a wealth of information. One particularly valuable source of data is the **Bing Traffic API**. This API can be easily called to provide traffic congestion or disruption incidents across a user-specified geographical area.

In addition, we can leverage weather data from the **Yahoo Weather API**. This API provides the current weather for a given location, taking zip codes or location input. It provides a wealth of local climate information including, but not limited to, temperature, wind speed, humidity, atmospheric pressure, and visibility. Additionally, it provides a text string description of current conditions as well as forecast information.

While there are other data sources that we can consider tying into our analysis, we'll begin with this data and see how we do.

Testing the performance of our model

In order to meaningfully assess our commute disruption prediction attempt, we should try to define test criteria and an appropriate performance score.

What we're attempting to do is identify the risk of commute disruption on the current day, each day. Preferably, we'd like to know the commute risk with sufficient advance notice that we can take action to mitigate it (for example, by leaving home earlier).

In order to do this, we're going to need three things:

- An understanding of what our model is going to output
- A measure we can use to quantify model performance
- Some target data we can use to score model performance according to our measure

We can have an interesting discussion about why this matters. It can be argued, effectively, that some models are information in purpose. Our commute risk score, it might be said, is useful insofar as it generates information that we didn't previously have.

The reality of the situation, however, is that there is inalienably going to be a performance criterion. In this case, it might simply be my satisfaction with the results output by my model, but it's important to be aware that there is always some performance criterion at play. Quantifying performance is therefore valuable, even in contexts where a model appears to be informational (or even better, unsupervised). This makes it prudent to resist the temptation to waive performance testing; at least this way, you have a quantified performance measure to iteratively improve on.

A sensible starting point is to assert that our model is intended to output a numerical score in a *0-1* range for outbound (home to work) commutes on a given day. We have a few options with regard to how we present this score; perhaps the most obvious option would be to apply a log rescaling to the data. There are good reasons to log-scale and in this situation it might not be a bad idea. (It's not unlikely that the distribution of commute delay time obeys a power law.) For now, we won't reshape this set of scores. Instead, we'll wait to review the output of our model.

In terms of delivering practical guidance, a *0-1* score isn't necessarily very helpful. We might find ourselves wanting to use a bucketed system (such as high risk, mid risk, or low risk) with bucket boundaries at specific boundaries in the *0-1* range. In short, we would transition to treating the problem as a multiclass classification problem with categorical output (class labels), rather than as a regression problem with a continuous output.

This might improve model performance. (More specifically, because it'll increase the margin of free error to the full breadth of the relevant bucket, which is a very generous performance measure.) Equally though, it probably isn't a great idea to introduce this change on the first iteration. Until we've reviewed the distribution of real commute delays, we won't know where to draw the boundaries between classes!

Next, we need to consider how we measure the performance of our model. The selection of an appropriate scoring measure generally depends on the characteristics of the problem. We're presented with a lot of options around classifier performance scoring. (For more information around performance measures for machine learning algorithms, see the *Further reading* section at the end of this chapter.)

One way of deciding which performance measure is suitable for the task at hand is to consider the confusion matrix. A confusion matrix is a table of contingencies; in the context of statistical modeling, they typically describe the label prediction versus actual labels. It is common to output a confusion matrix (particularly for multiclass problems with more classes) for a trained model as it can yield valuable information about classification failures by failure type and class.

In this context, the reference to a confusion matrix is more illustrative. We can consider the following simplified matrix to assess whether there is any contingency that we don't care about:

Actual Result			
		TRUE	FALSE
Prediction	TRUE	**True Positive**	**False Positive**
	FALSE	**False Negative**	**True Negative**

In this case, we care about all four contingency types. False negatives will cause us to be caught in unexpected delays, while false positives will cause us to leave for our commute earlier than necessary. This implies that we want a performance measure that values both high sensitivity (true positive rate) and high specificity (false positive rate). The ideal measure, given this, is **area under the curve** (**AUC**).

The second challenge is how to measure this score; we need some target against which to predict. Thankfully, this is quite easy to obtain. I do, after all, have a daily commute to do! I simply began self-recording my commute time using a stopwatch, a consistent start time, and a consistent route.

It's important to recognize the limitations of this approach. As a data source, I am subject to my own internal trends. I am, for instance, somewhat sluggish before my morning coffee. Similarly, my own consistent commute route may possess local trends that other routes do not. It would be far better to collect commute data from a number of people and a number of routes.

However, in some ways, I was happy with the use of this target data. Not least because I am attempting to classify disruption to my own commute route and would not want natural variance in my commute time to be misinterpreted through training, say, against targets set by some other group of commuters or routes. In addition, given the anticipated slight natural variability from day-to-day, should be disregarded by a functional model.

It's rather hard to tell what's good enough in terms of model performance. More precisely, it's not easy to know when this model is outperforming my own expectations. Unfortunately, not only do I not have any very reliable with regard to the accuracy of my own commute delay predictions, it also seems unlikely that one person's predictions are generalizable to other commutes in other locations. It seems ill-advised to train a model to exceed a fairly subjective target.

Let's instead attempt to outperform a fairly simple threshold—a model that naively suggests that every single day will not contain commute delays. This target has the rather pleasing property of mirroring our actual behavior (in that we tend to get up each day and act as though there will not be transit disruption).

Of the 85 target data cases, 14 commute delays were observed. Based on this target data and the score measure we created, our target to beat is therefore *0.5*.

Twitter

Given that we're focusing this example analysis on the city of Vancouver, we have an opportunity to tap into a second Twitter data source. Specifically, we can use service announcements from Vancouver's public transit authority, Translink.

Translink Twitter

As noted, this data is already well-structured and conducive both to text mining and subsequent analysis; by processing this data using the techniques we reviewed in the last two chapters, we can clean the text and then encode it into useful features.

We're going to apply the Twitter API to harvest Translink's tweets over an extended period. The Twitter API is a pretty friendly piece of kit that is easy enough to work with from Python. (For extended guidance around how to work with the Twitter API, see the *Further reading* section at the end of this chapter!) In this case, we want to extract the date and body text from the tweet. The body text contains almost everything we need to know, including the following:

- The nature of the tweet (delay or non-delay)
- The station affected
- Some information as to the nature of the delay

One element that adds a little complexity is that the same Translink account tweets service disruption information for Skytrain lines and bus routes. Fortunately, the account is generally very uniform in the terms that it uses to describe service issues for each service type and subject. In particular, the Twitter account uses specific hashtags **(#RiderAlert** for bus route information, **#SkyTrain** for train-related information, and **#TransitAlert** for general alerts across both services, such as statutory holidays) to differentiate the subjects of service disruption.

Similarly, we can expect a delay to always be described using the word delay, a detour by the term detour, and a diversion, using the word diversion. This means that we can filter out unwanted tweets using specific key terms. Nice job, Translink!

Note

The data used in this chapter is provided within the GitHub solution accompanying this chapter in the `translink_tweet_data.json` file. The scraper script is also provided within the chapter code; in order to leverage it, you'll need to set up a developer account with Twitter. This is easy to achieve; the process is documented here and you can sign up here.

Once we've obtained our tweet data, we know what to do next—we need to clean and regularize the body text! As per Chapter 6, *Text Feature Engineering*, let's run `BeautifulSoup` and `NLTK` over the input data:

```
from bs4 import BeautifulSoup

tweets = BeautifulSoup(train["TranslinkTweets.text"])

tweettext = tweets.get_text()
```

```
brown_a = nltk.corpus.brown.tagged_sents(categories= 'a')

tagger = None
for n in range(1,4):
   tagger = NgramTagger(n, brown_a, backoff = tagger)

taggedtweettext = tagger.tag(tweettext)
```

We probably will not need to be as intensive in our cleaning as we were with the troll dataset in the previous chapter. Translink's tweets are highly formulaic and do not include non-ascii characters or emoticons, so the specific "deep cleaning" regex script that we needed to use in Chapter 6, *Text Feature Engineering*, won't be needed here.

This gives us a dataset with lower-case, regularized, and dictionary-checked terms. We are ready to start thinking seriously about what features we ought to build out of this data.

We know that the base method of detecting a service disruption issue within our data is the use of a delay term in a tweet. Delays happen in the following ways:

- At a given location
- At a given time
- For a given reason
- For a given duration

Each of the first three factors is consistently tracked within Translink tweets, but there are some data quality concerns that are worth recognizing.

Location is given in terms of an affected street or station *at 22nd Street*. This isn't a perfect description for our purpose as we're unlikely to be able to turn a street name and route start/end points into a general *affected area* without doing substantial additional work (as no convenient reference exists that allows us to draw a bounding box based on this information).

Time is imperfectly given by the tweet datetime. While we don't have visibility on whether tweets are made within a consistent time from service disruption, it's likely that Translink has targets around service notification. For now, it's sensible to proceed under the assumption that the tweet times are likely to be sufficiently accurate.

The exception is likely to be for long-running issues or problems that change severity (delays that are expected to be minor but which become significant). In these cases, tweets may be delayed until the Translink team recognizes that the issue has become tweet-worthy. The other possible cause of data quality issues is inconsistency in Translink's internal communications; it's possible that engineering or platform teams don't always inform the customer service notifications team at the same speed.

We're going to have to take a certain amount on faith though, as there isn't a huge amount we can do to measure these delay effects without a dataset of real-time, accurate Translink service delays. (If we had that, we'd be using it instead!)

Reasons for Skytrain service delays are consistently described by Translink and can fall into one of the following categories:

- Rail
- Train
- Switch
- Control
- Unknown
- Intrusion
- Medical
- Police
- Power

With each category described within the tweet body using the specific proper term given in the preceding list. Obviously, some of these categories (Police, Power, Medical) are less likely to be relevant as they wouldn't tell us anything useful about road conditions. The rate of train, track, and switch failure may be correlated with detour likelihood; this suggests that we may want to keep those cases for classification purposes.

Meanwhile, bus route service delays contain a similar set of codes, many of which are very relevant to our purposes. These codes are as follows:

- **Motor Vehicle Accident (MVA)**
- Construction
- Fire
- Watermain
- Traffic

Encoding these incident types is likely to prove useful! In particular, it's possible that certain service delay types are more impactful than others, increasing the risk of a longer service delay. We'll want to encode service delay types and use them as parameters in our subsequent modeling.

To do this, let's apply a variant of one-hot encoding, which does the following:

- It creates a conditional variable for each of the service risk types and sets all values to zero
- It checks tweet content for each of the service risk type terms
- It sets the relevant conditional variable to 1 for each tweet that contains a specific risk term

This effectively performs one-hot encoding without taking the bothersome intermediary step of creating the factorial variable that we'd normally be processing:

```
from sklearn import preprocessing

enc = preprocessing.OneHotEncoder(categorical_features='all', dtype=
'float', handle_unknown='error', n_values='auto', sparse=True)

tweets.delayencode = enc.transform(tweets.delaytype).toarray()
```

Beyond what we have available to use as a feature on a per-incident basis, we can definitely look at the relationship between service disruption risk and disruption frequency. If we see two disruptions in a week, is a third more likely or less likely?

While these questions are interesting and potentially fruitful, it's usually more prudent to work up a limited feature set and simple model on a first pass than to overengineer a sprawling feature set. As such, we'll run with the initial incidence rate features and see where we end up.

Consumer comments

A major cultural development in 2010 was the widespread use of public online domains for self-expression. One of the happier products of this is the availability of a wide array of self-reported information on any number of subjects, provided we know how to tap into this.

Commute disruptions are frequently occurring events that inspire a personal response, which means that they tend to be quite broadly reported on social media. If we write an appropriate dictionary for key-term search, we can begin using Twitter particularly as a source of timestamped information on traffic and transit issues around the city.

In order to collect this data, we'll make use of a dictionary-based search approach. We're not interested in the majority of tweets from the period in question (and as we're using the RESTful API, there are return limits to consider). Instead, we're interested in identifying tweet data containing key terms related to congestion or delay.

Unfortunately, tweets harvested from a broad range of users tend not to conform to consistent styles that aid analysis. We're going to have to apply some of the techniques we developed in the preceding chapter to break down this data into a more easily analyzed format.

In addition to using a dictionary-based search, we could do some work to narrow the search area down. The most authoritative way to achieve this is to use a bounding box of coordinates as an argument to the Twitter API, such that any related query exclusively returns results gathered from within this area.

As always, on our first pass, we'll keep things simple. In this case, we'll count up the number of traffic disruption tweets in the current period. There is some additional work that we could benefit from doing with this data on subsequent iterations. Just as the Translink data contained clearly-defined delay cause categories, we could try to use specialized dictionaries to isolate delay types based on key terms (for example, a dictionary of construction-related terms and synonyms).

We could also look at defining a more nuanced quantification of disruption tweet rate than a simple count of recent. We could, for instance, look at creating a weighted count feature that increases the impact of multiple simultaneous tweets (potentially indicative of severe disruption) via a nonlinear weighting.

The Bing Traffic API

The next API we're going to tap into is the Bing Traffic API. This API has the advantage of being easily accessed; it's freely available (whereas some competitor APIs sit behind paywalls), returns data, and provides a good level of detail. Among other things, the API returns an incident location code, a general

description of the incident, together with congestion information, an incident type code, and start/end timestamps.

Helpfully, the incident type codes provided by this API describe a broad set of incident types, as follows:

1. `Accident`.
2. `Congestion`.
3. `DisabledVehicle`.
4. `MassTransit`.
5. `Miscellaneous`.
6. `OtherNews`.
7. `PlannedEvent`.
8. `RoadHazard`.
9. `Construction`.
10. `Alert`.
11. `Weather`.

Additionally, a severity code is provided with the severity values translated as follows:

1. `LowImpact`.
2. `Minor`.
3. `Moderate`.
4. `Serious`.

One downside, however, is that this API doesn't receive consistent information between regions. Querying in France, for instance, returns codes from multiple other incident types, (I observed 1, 3, 5, 8 for a town in northern France over a period of one month.) but doesn't seem to show every code. In other locations, even less data is available. Sadly, Vancouver tends to show data for codes 9 or 5 exclusively, but even the miscellaneous-coded incidents appear to be construction-related:

```
Closed between Victoria Dr and Commercial Dr - Closed. Construction
work. 5
```

This is a somewhat bothersome limitation. Unfortunately, it's not something that we can easily fix; Bing's API is simply not sourcing all of the data that we want! Unless we pay for a more complete dataset (or an API with fuller data capture is available in your area!), we're going to need to keep working with what we have.

An example of querying this API is as follows:

```
importurllib.request, urllib.error, urllib.parse
import json

latN = str(49.310911)
latS = str(49.201444)
```

```
lonW = str(-123.225544)
lonE = str(-122.903931)

url = 'http://dev.virtualearth.net/REST/v1/Traffic/
Incidents/'+latS+','+lonW+','+latN+','+lonE+'?key='GETYOUROWNKEYPLEAS
E'

response = urllib.request.urlopen(url).read()
data = json.loads(response.decode('utf8'))
resources = data['resourceSets'][0]['resources']

print('----------------------------------------------------')
print('PRETTIFIED RESULTS')
print('----------------------------------------------------')
for resourceItem in resources:
    description = resourceItem['description']
typeof = resourceItem['type']
    start = resourceItem['start']
    end = resourceItem['end']
print('description:', description);
print('type:', typeof);
print('starttime:', start);
print('endtime:', end);
print('----------------------------------------------------')
```

This example yields the following data;

```
----------------------------------------------------
PRETTIFIED RESULTS
----------------------------------------------------
description: Closed between Boundary Rd and PierviewCres - Closed
due to roadwork.
type: 9
severity 4
starttime: /Date(1458331200000)/
endtime: /Date(1466283600000)/
----------------------------------------------------
description: Closed between Commercial Dr and Victoria Dr - Closed
due to roadwork.
type: 9
severity 4
starttime: /Date(1458327600000)/
endtime: /Date(1483218000000)/
----------------------------------------------------
description: Closed between Victoria Dr and Commercial Dr - Closed.
Construction work.
```

```
type: 5
severity 4
starttime: /Date(1461780543000)/
endtime: /Date(1481875140000)/
----------------------------------------------------
description: At Thurlow St - Roadwork.
type: 9
severity 3
starttime: /Date(1461780537000)/
endtime: /Date(1504112400000)/
----------------------------------------------------
```

Even after recognizing the shortcomings of uneven code availability across different geographical areas, the data from this API should provide us with some value. Having a partial picture of traffic disruption incidents still gives us data for a reasonable period of dates. The ability to localize traffic incidents within an area of our own definition and returning data relevant to the current date is likely to help the performance of our model.

Deriving and selecting variables using feature engineering techniques

On our first pass over the input data, we repeatedly made the choice to keep our initial feature set small. Though we saw lots of opportunities in the data, we prioritized viewing an initial result above following up on those opportunities.

It is likely, however, that our first dataset won't help us solve the problem very effectively or hit our targets. In this event, we'll need to iterate over our feature set, both by creating new features and winnowing our feature set to reduce down to the valuable outputs of that feature creation process.

One helpful example involves one-hot encoding and RFE. In this chapter, we'll use one-hot to turn weather data and tweet dictionaries into tensors of m*n size. Having produced m-many new columns of data, we'll want to reduce the liability of our model to be misled by some of these new features (for instance, in cases where multiple features reinforce the same signal or where misleading but commonly-used terms are not cleaned out by the data cleaning processes we described in Chapter 6, *Text Feature Engineering*). This can be done very effectively by RFE, the technique for feature selection that we discussed earlier in this chapter.

In general, it can be helpful to work using a methodology that applies the techniques seen in the last two chapters using an expand-contract process. First, use techniques that can generate potentially valuable new features, such as transformations and encodings, to expand the feature set. Then, use techniques that can identify the most performant subset of those features to remove the features that do not perform well. Throughout this process, test different target feature counts to identify the best available feature set at different numbers of features.

Some data scientists interpret how this is done differently from others. Some will build all of their features using repeated iterations over the feature creation techniques we've discussed, then reduce that feature set—the motivation being that this workflow minimizes the risk of losing data. Others will perform the full process iteratively. How you choose to do this is entirely up to you!

On our initial pass over the input data, then, we have a feature set that looks as follows:

```
{
  'DisruptionInformation': {
    'Date': '15-05-2015',
    'TranslinkTwitter': [{
      'Service': '0',
      'DisruptionIncidentCount': '4'
  }, {
      'Service': '1',
      'DisruptionIncidentCount': '0'
    }]
  },
  'BingTrafficAPI': {
    'NewIncidentCount': '1',
    'SevereIncidentCount': '1',
    'IncidentCount': '3'
  },
  'ConsumerTwitter': {
    'DisruptionTweetCount': '4'
  }
}
```

It's unlikely that this dataset is going to perform well. All the same, let's run it through a basic initial algorithm and get a general idea as to how near our target we are; this way, we can learn quickly with minimal overhead!

In the interest of expedience, let's begin by running a first pass using a very simple regression algorithm. The simpler the technique, the faster we can run it (and often, the more transparent it is to us what went wrong and why). For this reason (and because we're dealing with a regression problem with a continuous output rather than a classification problem), on a first pass we'll work with a simple linear regression model:

```
from sklearn import linear_model

tweets_X_train = tweets_X[:-20]
tweets_X_test = tweets_X[-20:]


tweets_y_train = tweets.target[:-20]
tweets_y_test = tweets.target[-20:]

regr = linear_model.LinearRegression()

regr.fit(tweets_X_train, tweets_y_train)

print('Coefficients: \n', regr.coef_)
```

```
print("Residual sum of squares: %.2f" %
np.mean((regr.predict(tweets_X_test) - tweets_y_test) ** 2))

print('Variance score: %.2f' % regr.score(tweets_X_test,
tweets_y_test))

plt.scatter(tweets_X_test, tweets_y_test,  color='black')
plt.plot(tweets_X_test, regr.predict(tweets_X_test),
color='blue',linewidth=3)

plt.xticks(())
plt.yticks(())
plt.show()
```

At this point, our AUC is pretty lousy; we're looking at a model with an AUC of 0.495. We're actually doing worse than our target! Let's print out a confusion matrix to see what this model's doing wrong:

		Prediction	
		TRUE	FALSE
Actual Result	TRUE	1	9
	FALSE	18	136

According to this matrix, it's doing everything not very well. In fact, it's claiming that almost all of the records show no incidents, to the extent of missing 90% of real disruptions!

This actually isn't too bad at all, given the early stage that we're at with our model and our features, as well as the uncertain utility of some of our input data. At the same time, we should expect an incidence rate of 6% (as our training data suggests that incidents have been seen to occur roughly once every 16 commutes). We'd still be doing a little better by guessing that every day will involve a disrupted commute (if we ignore the penalty to our lifestyle entailed by leaving home early each day).

Let's consider what changes we could make in a next pass.

1. First off, we could stand to improve our input data further. We identified a number of new features that we could create from existing sources using a range of transformation techniques.
2. Secondly, we could look at extending our dataset using additional information. In particular, a weather dataset describing both temperature and humidity may help us improve our model.
3. Finally, we could upgrade our algorithm to something with a little more grunt, random forests or SVM being obvious examples. There are good reasons not to do this just yet. The main reason is that we can continue to learn a lot from linear regression; we can compare against earlier results to understand how much value our changes are adding, while retaining a fast iteration loop and

simple scoring methods. Once we begin to get minimal returns on our feature preparation, we should consider upgrading our model.

For now, we'll continue to upgrade our dataset. We have a number of options here. We can encode location into both traffic incident data from the Bing API's "description" field and into Translink's tweets. In the case of Translink, this is likely to be more usefully done for bus routes than Skytrain routes (given that we restricted the scope of this analysis to focus solely on traffic commutes).

We can achieve this goal in one of two ways;

- Using a corpus of street names/locations, we can parse the input data and build a one-hot matrix
- We can simply run one-hot encoding over the entire body of tweets and entire set of API data

Interestingly, if we intend to use dimensionality reduction techniques after performing one-hot encoding, we can encode the entire body of both pieces of text information without any significant problems. If features relating to the other words used in tweets and text are not relevant, they'll simply be scrubbed out during RFE.

This is a slightly laissez-faire approach, but there is a subtle advantage. Namely, if there is some other potentially useful content to either data source that we've so far overlooked as a potential feature, this process will yield the added benefit of creating features based on that information.

Let's encode locations in the same way we encoded delay types:

```
from sklearn import preprocessing

enc = preprocessing.OneHotEncoder(categorical_features='all', dtype=
'float', handle_unknown='error', n_values='auto', sparse=True)

tweets.delayencode = enc.transform(tweets.location).toarray()
```

Additionally, we should follow up on our intention to create recent count variables from Translink and Bing maps incident logging. The code for this aggregation is available in the GitHub repository accompanying this chapter!

Rerunning our model with this updated data produced results with a very slight improvement; the predicted variance score rose to 0.56. While not dramatic, this is definitely a step in the right direction.

Next, let's follow up on our second option—adding a new data source that provides weather data.

The weather API

We've previously grabbed data that will help us tell whether commute disruption is happening—reactive data sources that identify existing delays. We're going to change things up a little now, by trying to find data that relates to the causes of delays and congestion. Roadworks and construction information definitely falls into this category (along with some of the other Bing Traffic API codes).

One factor that is often (anecdotally!) tied to increased commute time is bad weather. Sometimes this is pretty obvious; heavy frost or high winds have a clear impact on commute time. In many other cases, though, it's not clear what the strength and nature of the relationship between climatic factors and disruption likelihood is for a given commute.

By extracting pertinent weather data from a source with sufficient granularity and geo coverage, we can hopefully use strong weather signals to help improve our correct prediction of disruption.

For our purposes, we'll use the Yahoo Weather API, which provides a range of temperature, atmospheric, pressure-related, and other climate data, both current and forecasted. We can query the Yahoo Weather API without needing a key or login process, as follows:

```
import urllib2, urllib, json

baseurl = https://query.yahooapis.com/v1/public/yql?

yql_query = "select item.condition from weather.forecast where 
woeid=9807"
yql_url = baseurl + urllib.urlencode({'q':yql_query}) + 
"&format=json"
result = urllib2.urlopen(yql_url).read()
data = json.loads(result)
print data['query']['results']
```

To get an understanding for what the API can provide, replace `item.condition` (in what is fundamentally an embedded SQL query) with `*`. This query outputs a lot of information, but digging through it reveals valuable information, including the current conditions:


```
{
   'channel': {
     'item': {
      'condition': {
         'date': 'Thu, 14 May 2015 03:00 AM PDT', 'text': 'Cloudy', 
'code': '26', 'temp': '46'
      }
    }
  }
}
```


7-day forecasts containing the following information:


```
{
    'item': {
      'forecast': {
        'code': '39', 'text': 'Scattered Showers', 'high': '60', 
'low': '44', 'date': '16 May 2015', 'day': 'Sat'
      }
```

```
    }
}
```

And other current weather information:

```
'astronomy': {
     'sunset': '8:30 pm', 'sunrise': '5:36 am'

   'wind': {
     'direction':  '270', 'speed': '4', 'chill': '46'
```

For the purpose of building a training dataset, we extracted data on a daily basis via an automated script that ran from May 2015 to January 2016. The forecasts may not be terribly useful to us as it's likely that our model will rerun over current data on a daily basis rather than being dependent on forecasts. However, we will definitely make use of the `wind.direction`, `wind.speed`, and `wind.chill` variables, as well as the `condition.temperature` and `condition.text` variables.

In terms of how to further process this information, one option jumps to mind. One-hot encoding of weather tags would enable us to use weather condition information as categorical variables, just as we did in the preceding chapter. This seems like a necessary step to take. This significantly inflates our feature set, leaving us with the following data:

```
{
  'DisruptionInformation': {
    'Date': '15-05-2015',
    'TranslinkTwitter': [{
      'Service': '0',
      'DisruptionIncidentCount': '4'
    }, {
      'Service': '1',
      'DisruptionIncidentCount': '0'
    }]
  },
  'BingTrafficAPI': {
    'NewIncidentCount': '1',
    'SevereIncidentCount': '1',
    'IncidentCount': '3'
  },
  'ConsumerTwitter': {
    'DisruptionTweetCount': '4'
  },
  'YahooWeather':{
    'temp: '45'
    'tornado': '0',
    'tropical storm': '0',
    'hurricane': '0',
    'severe thunderstorms': '0',
```

```
    'thunderstorms': '0',
    'mixed rain and snow': '0',
    'mixed rain and sleet': '0',
    'mixed snow and sleet': '0',
'freezing drizzle': '0',
'drizzle': '0',
'freezing rain': '0',
'showers': '0',
'snow flurries': '0',
'light snow showers': '0',
'blowing snow': '0',
'snow': '0',
'hail': '0',
'sleet': '0',
'dust': '0',
'foggy': '0',
'haze': '0',
'smoky': '0',
'blustery': '0',
'windy': '0',
'cold': '0',
'cloudy': '1',
'mostly cloudy (night)': '0',
'mostly cloudy (day)': '0',
'partly cloudy (night)': '0',
'partly cloudy (day)': '0',
'clear (night)': '0',
'sunny': '0',
'fair (night)': '0',
'fair (day)': '0',
'mixed rain and hail': '0',
'hot': '0',
'isolated thunderstorms': '0',
'scattered thunderstorms': '0',
'scattered showers': '0',
'heavy snow': '0',
'scattered snow showers': '0',
'partly cloudy': '0',
'thundershowers': '0',
'snow showers': '0',
'isolated thundershowers': '0',
'not available': '0',
}
```

It's very likely that a lot of time could be valuably sunk into further enriching the weather data provided by the Yahoo Weather API. For the first pass, as always, we'll remain focused on building a model that takes the features that we described previously.

Note

It's definitely worth considering how we would do further work with this data. In this case, it's important to distinguish between cross-column data transformations and cross-row transformations.

A cross-column transformation is one where variables from different features in the same input case were transformed based on one another. For instance, we might take the start date and end date of a case and use it to calculate the duration. Interestingly, the majority of the techniques that we've studied in this book won't gain a lot from many such transformations. Most machine learning techniques capable of drawing nonlinear decision boundaries tend to encode relationships between variables in their modeling of a dataset. Deep learning techniques often take this capability a step further. This is part of the reason that some feature engineering techniques (particularly basic transformations) add less value for deep learning applications.

Meanwhile, a cross-row transformation is typically an aggregation. The central tendency of the last n-many duration values, for instance, is a feature that can be derived by an operation over multiple rows. Naturally, some features can be derived by a combination of column-wise and row-wise operations. The interesting thing about cross-row transformations is that it's usually quite unlikely that a model will train to recognize them, meaning that they tend to continue to add value in very particular contexts.

The reason that this information is relevant, of course, is that recent weather is a context in which features derived from cross-row operations might add new information to our model. Change in barometric pressure or temperature over the last n hours, for instance, might be a more useful variable than the current pressure or temperature. (Particularly, when that our model is intended to predict commutes to take place later in the same day!)

The next step is to rerun our model. This time, our AUC is a little higher; we're scoring *0.534*. Looking at our confusion matrix, we're also seeing improvements:

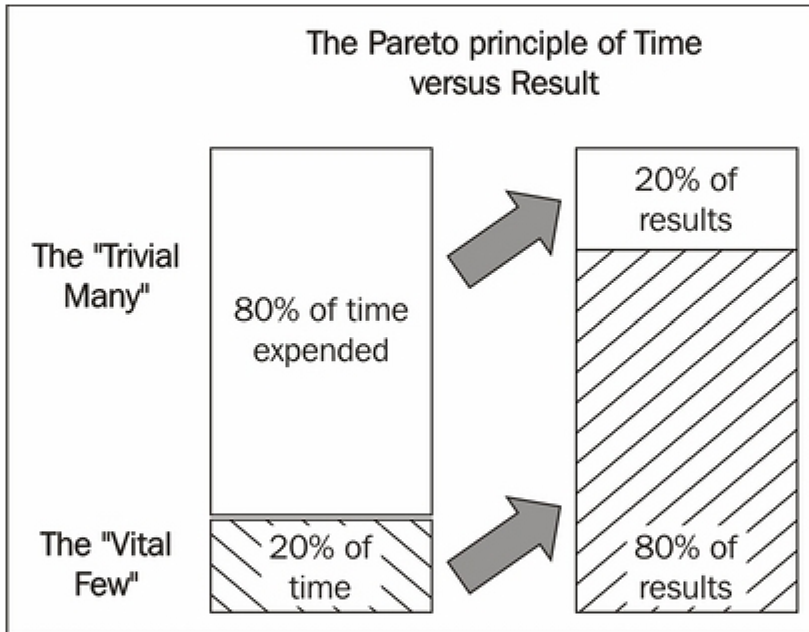
		Prediction	
		TRUE	FALSE
Actual Result	TRUE	3	7
	FALSE	22	132

If the issues are linked to weather factors, continuing to pull weather data is a good idea; setting this solution up to run over an extended period will gradually gather longitudinal inputs from each source, gradually giving us much more reliable predictions.

At this point, we're only a short distance away from our MVP target. We can continue to extend our input dataset, but the smart solution is to find another way to approach the problem. There are two actions that we can meaningfully take.

Note

Being human, data scientists tend to think in terms of simplifying assumptions. One of these that crops up quite frequently is basically an application of the Pareto principle to cost/benefit analysis decisions. Fundamentally, the Pareto principle states that for many events, roughly 80% of the value or effect comes from roughly 20% of the input effort, or cause, obeying what's referred to as a Pareto distribution. This concept is very popular in software engineering contexts among others, as it can guide efficiency improvements.



To apply this theory to the current case, we know that we could spend more time finessing our feature engineering. There are techniques that we haven't applied and other features that we could create. However, at the same time, we know that there are entire areas that we haven't touched: external data searches and model changes, particularly, which we could quickly try. It makes sense to explore these cheap but potentially impactful options on our next pass before digging into additional dataset preparation.

During our exploratory analysis, we noticed that some of our variables are quite sparse. It wasn't immediately clear how helpful they all were (particularly for stations where fewer incidents of a given type occurred).

Let's test out our variable set using some of the techniques that we worked with earlier in the chapter. In particular, let's apply `Lasso` to the problem of reducing our feature set to a performant subset:

```
fromsklearn.preprocessing import StandardScaler


scaler = StandardScaler()
```

```
X = scaler.fit_transform(DisruptionInformation["data"])
Y = DisruptionInformation["target"]
names = DisruptionInformation["feature_names"]

lasso = Lasso(alpha=.3)
lasso.fit(X, Y)

print "Lasso model: ", pretty_print_linear(lasso.coef_, names, sort
= True)
```

This output is immediately valuable. It's obvious that many of the weather features (either through not showing up sufficiently often or not telling us anything useful when they do) are adding nothing to our model and should be removed. In addition, we're not getting a lot of value from our traffic aggregates. While these can remain in for the moment (in the hope that gathering more data will improve their usefulness), for our next pass we'll rerun our model without the poorly-scoring features that our use of LASSO has revealed.

There is one fairly cheap additional change, which we ought to make: we should upgrade our model to one that can fit nonlinearly and thus can fit to approximate any function. This is worth doing because, as we observed, some of our features showed a range of skewed distributions indicative of a nonlinear underlying trend. Let's apply a random forest to this dataset:

```
fromsklearn.ensemble import RandomForestClassifier,
ExtraTreesClassifier
rf = RandomForestRegressor(n_jobs = 3, verbose = 3, n_estimators=20)
rf.fit(DisruptionInformation_train.targets,DisruptionInformation_trai
n.data)

r2 = r2_score(DisruptionInformation.data,
rf.predict(DisruptionInformation.targets))
mse = np.mean((DisruptionInformation.data -
rf.predict(DisruptionInformation.targets))**2)

pl.scatter(DisruptionInformation.data,
rf.predict(DisruptionInformation.targets))
pl.plot(np.arange(8, 15), np.arange(8, 15), label="r^2=" + str(r2),
c="r")
pl.legend(loc="lower right")
pl.title("RandomForest Regression with scikit-learn")
pl.show()
```

Let's return again to our confusion matrix:

	Prediction		
		TRUE	FALSE
Actual Result	TRUE	4	6
	FALSE	15	134

At this point, we're doing fairly well. A simple upgrade to our model has yielded significant improvements, with our model correctly identifying almost 40% of commute delay incidents (enough to start to be useful to us!), while misclassifying a small amount of cases.

Frustratingly, this model would still be getting us out of bed early incorrectly more times than it would correctly. The gold standard, of course, would be if it were predicting more commute delays than it was causing false (early) starts! We could reasonably hope to achieve this target if we continue to gather feature data over a sustained period; the main weakness of this model is that it has very few cases to sample from, given the rarity of commute disruption events.

We have, however, succeeded in gathering and marshaling a range of data from different sources in order to create a model from freely-available data that yields a recognizable, real-world benefit (reducing the amount of late arrivals at work by 40%). This is definitely an achievement to be happy with!

Further reading

My suggested go-to introduction to feature selection is Ando Sabaas' four-part exploration of a broad range of feature selection techniques. It's full of Python code snippets and informed commentary. Get started at http://blog.datadive.net/selecting-good-features-part-i-univariate-selection/.

For a discussion on feature selection and engineering that ranges across materials in chapters 6 and 7, consider Alexandre Bourhard-Côté's slides at http://people.eecs.berkeley.edu/~jordan/courses/294-fall09/lectures/feature/slides.pdf. Also consider reviewing Jeff Howbert's slides at http://courses.washington.edu/css490/2012.Winter/lecture_slides/05a_feature_creation_selection.pdf.

There is a shortage of thorough discussion of feature creation, with a lot of available material discussing either dimensionality reduction techniques or very specific feature creation as required in specific domains. One way to get a more general understanding of the range of possible transformations is to read code documentation. A decent place to build on your existing knowledge is Spark ML's feature-transformation algorithm documentation at https://spark.apache.org/docs/1.5.1/ml-features.html#feature-transformers, which describes a broad set of possible transformations on numerical and text features. Remember, though, that feature creation is often problem-specific, domain-specific, and a highly creative process. Once you've learned a range of technical options, the trick is in figuring out how to apply these techniques to the problem at hand!

For readers with an interest in hyperparameter optimization, I recommend that you read Alice Zheng's posts on Turi's blog as a great place to start: http://blog.turi.com/how-to-evaluate-machine-learning-models-part-4-hyperparameter-tuning.

I also find the scikit-learn documentation to be a useful reference for grid search specifically: http://scikit-learn.org/stable/modules/grid_search.html.

Summary

In this chapter, you learned and applied a set of techniques that enable us to effectively build and finesse datasets for machine learning, starting from very little initial data. These powerful techniques enable a data scientist to turn seemingly shallow datasets into opportunities. We demonstrated this power using a set of customer service tweets to create a travel disruption predictor.

In order to take that solution into production, though, we'd need to add some functionality. Removing some locations in the penultimate step was a questionable decision; if this solution is intended to identify journey disruption risk, then removing locations seems like a non-starter! This is particularly true given that we do not have year-round data and so cannot identify the effect of seasonal or longitudinal trends (like extended maintenance works or a scheduled station closure). We were a little hasty in removing these elements and a better solution would be to retain them for a longer period.

Following on from these concerns, we should recognize the need to start building some dynamism into our solution. When spring rolls around and our dataset starts to contain new climate conditions, it is entirely likely that our model will fail to adapt as effectively. In the next chapter, we will be looking at building more sophisticated model ensembles and discuss methods of building robustness into your model solutions.

Chapter 8. Ensemble Methods

As we progressed through the earlier chapters of this book, you learned how to apply a number of new techniques. We developed our use of several advanced machine learning algorithms and acquired a broad range of companion techniques used to enhance your use of learning techniques via more effective feature selection and preparation. This chapter seeks to enhance your existing technique set using ensemble methods: techniques that bind multiple different models together to solve a real-world problem.

Ensemble techniques have become a fundamental part of the data scientist's toolset. The use of ensembles has become common practice in competitive machine learning contexts, and ensembles are now considered an indispensable tool in many contexts. The techniques that we'll develop in this chapter give our models an edge in performance, while increasing their robustness to underlying data change.

We'll examine a series of ensembling options, discussing both the code and application of these techniques. We'll color this explanation with guidance and reference to real-world applications, including the models created by successful Kagglers.

The development of any of the models that we reviewed in this title allows us to solve a wide range of data problems, but applying our models to production contexts raises an additional set of problems. Our solutions are still vulnerable to changes in the underlying observations. Whether this is expressed in a different population of individuals, in temporal variations (for example, seasonal changes in the phenomenon being captured) or by other changes to the underlying conditions, the end result is often the same—the models that worked well in the conditions they were trained against are frequently unable to generalize and continue to perform well as time passes.

The final section of this chapter describes methodologies to transfer the techniques from this book to operational environments and the kinds of additional monitoring and support you should consider if your intended applications have to be resilient to change.

Introducing ensembles

"This is how you win ML competitions: you take other peoples' work and ensemble them together."

--Vitaly Kuznetsov NIPS2014

In the context of machine learning, an ensemble is a set of models that is used to solve a shared problem. An ensemble is made up of two components: a set of models and a set of decision rules that govern how the results of those models are combined into a single output.

Ensembles offer a data scientist the ability to construct multiple solutions for a given problem and then combine these into a single final result that draws from the best elements of each input solution. This provides robustness against noise, which is reflected in more effective training against an initial dataset (leading to lower levels of overfitting and reductions in training error) and against data change of the kinds discussed in the preceding section.

It is no exaggeration to say that ensembles are the most important recent development in machine learning.

In addition, ensembles enable greater flexibility in how one solves for a given problem, in that they enable the data scientist to test different parts of a solution and resolve issues specific to subsets of the input data or parts of the models in use, without completely retuning the whole model. As we'll see, this can make life easier!

Ensembles are typically considered as falling into one of several classes, based on the nature of the decision rules used. The key ensemble types are as follows:

- **Averaging methods**: They develop models in parallel and then use averaging or voting techniques to develop a combined estimator
- **Stacking (or Blending) methods**: They use the weighted output of multiple classifiers as inputs to a next-layer model
- **Boosting methods**: They involve building models in sequence where each added model aims to improve the score of the combined estimator

Given the importance and utility of both of these classes of the ensemble method, we'll treat each one in turn: discussing theory, algorithm options, and real-world examples.

Understanding averaging ensembles

Averaging ensembles have a long and rich history in the physical sciences and statistical modeling, seeing a common application in many contexts including molecular dynamics and audio signal processing. Such ensembles are typically seen as almost exactly replicated cases of a given system. The average (mean) values of and variance between cases in this system are key values for the system as a whole.

In a machine learning context, an averaging ensemble is a collection of models that train on the same dataset, whose results are aggregated in a range of ways. Depending on implementation goals, an averaging ensemble can bring several benefits.

Averaging ensembles can be used to reduce the variability of a model's performance. One common method involves creating multiple model configurations that take different parameter subsets as input. Techniques that take this approach are referred to collectively as bagging algorithms.

Using bagging algorithms

Different bagging implementations will operate differently but share the common property of taking random subsets of the feature space. There are four main types of the bagging approach. Pasting draws random subsets of the samples without replacement. When this is done with replacement, then the approach is simply called **bagging**. Pasting is typically computationally cheaper than bagging and can yield similar results in simpler applications.

When samples are taken feature-wise, the method is known as **random subspaces**. Random subspace methods provide a slightly different capability; they essentially reduce the need for extensive, highly optimized feature selection. Where such activities typically lead to a single model with optimized input,

random subspaces allow the use of multiple configurations in parallel, with a flattening of the variance of any one solution.

Note

While the use of an ensemble to reduce the variability in model performance may sound like a performance hit (the natural response might be but why not just pick the single best performing model in the ensemble?), there are big advantages to this approach.

Firstly, as discussed, averaging improves the ability of your model set to adapt to unfamiliar noise (that is, it reduces overfitting). Secondly, an ensemble can be used to target different elements of the input dataset to model effectively. This is a common approach in competitive machine learning contexts, where a data scientist will iteratively adjust the ensemble based on the results of classification and particular types of failure cases. In some cases, this is an exhaustive process involving the inspection of model results (commonly as part of a normal, iterative model development process) but many data scientists prefer techniques or a solution that they will implement first.

Random subspaces can be a very powerful approach, particularly if it's possible to use multiple subspace sizes and exhaustively check feature combinations. The cost of random subspace methods increases nonlinearly with the size of your dataset and, beyond a certain point, it will become costly to test every configuration of parameters for multiple subspace sizes.

Finally, an ensemble's estimators may be created from subsets drawn from both samples and features, in a method known as **random patches**. On a like-for-like case, the performance of random patches is usually around the same level as that of random subspace techniques with significantly reduced memory consumption.

As we've discussed the theory behind bagging ensembles, let's look at how we go about implementing one. The following code describes a random patches classifier implemented using sklearn's `BaggingClassifier` class:

```
from sklearn.cross_validation import cross_val_score
from sklearn.ensemble import BaggingClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.datasets import load_digits
from sklearn.preprocessing import scale

digits = load_digits()
data = scale(digits.data)
X = data
y = digits.target


bagging = BaggingClassifier(KNeighborsClassifier(), max_samples=0.5,
max_features=0.5)
scores = cross_val_score(bagging, X, y)
mean = scores.mean()
```

```
print(scores)
print(mean)
```

As with many sklearn classifiers, the core code needed is very straightforward; the classifier is initialized and used to score the dataset. Cross-validation (via `cross_val_score`) adds no meaningful complexity.

This bagging classifier used a **K-Nearest Neighbors** (**KNN**) classifier (`KNeighboursClassifier`) as a base, with feature-wise and case-wise sampling rates each set to 50%. This outputs very strong results against the digits dataset, correctly classifying a mean of 93% of cases after cross-validation:

```
[ 0.94019934  0.92320534  0.9295302 ]

0.930978293043
```

Using random forests

An alternative set of averaging ensemble techniques is referred to collectively as random forests. Perhaps the most successful ensemble technique used by competitive data scientists, random forests develop parallel sets of decision tree classifiers. By introducing two main sources of randomness to the classifier construction, the forest ends up containing diverse trees. The data that is used to build each tree is sampled with replacement from the training set, while the tree creation process no longer uses the best split from all features, instead choosing the best split from a random subset of the features.

Random forests can be easily called using the `RandomForestClassifier` class in `sklearn`. For a simple example, consider the following:

```
import numpy as np
from sklearn.ensemble import RandomForestClassifier
from sklearn.datasets import load_digits
from sklearn.preprocessing import scale

digits = load_digits()
data = scale(digits.data)

n_samples, n_features = data.shape
n_digits = len(np.unique(digits.target))
labels = digits.target

clf = RandomForestClassifier(n_estimators=10)
clf = clf.fit(data, labels)
scores = clf.score(data,labels)
print(scores)
```

The scores output by this ensemble, 0.999, are difficult to beat. Indeed, we haven't seen performance at this level from any of the individual models we employed in preceding chapters.

A variant of random forests, called **extremely randomized trees** (**ExtraTrees**), uses the same random subset of features method in selecting the best split at each branch in the tree. However, it also randomizes the discrimination threshold; where a decision tree normally chooses the most effective split between classes, ExtraTrees split at a random value.

Due to the relatively efficient training of decision trees, a random forest algorithm can potentially support a large number of varied trees with the effectiveness of the classifier improving as the number of nodes increases. The randomness introduced provides a degree of robustness to noise or data change; like the bagging algorithms we reviewed earlier, however, this gain typically comes at the cost of a slight drop in performance. In the case of ExtraTrees, the robustness may increase further while the performance measure improves (typically a bias value reduces).

The following code describes how ExtraTrees work in practice. As with our random subspace implementation, the code is very straightforward. In this case, we'll develop a set of models to compare how ExtraTrees shape up against tree and random forest approaches:

```
from sklearn.cross_validation import cross_val_score
from sklearn.ensemble import RandomForestClassifier
from sklearn.ensemble import ExtraTreesClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.datasets import load_digits
from sklearn.preprocessing import scale

digits = load_digits()
data = scale(digits.data)
X = data
y = digits.target

clf = DecisionTreeClassifier(max_depth=None, min_samples_split=1,
    random_state=0)
scores = cross_val_score(clf, X, y)
print(scores)

clf = RandomForestClassifier(n_estimators=10, max_depth=None,
    min_samples_split=1, random_state=0)
scores = cross_val_score(clf, X, y)
print(scores)

clf = ExtraTreesClassifier(n_estimators=10, max_depth=None,
    min_samples_split=1, random_state=0)
scores = cross_val_score(clf, X, y)
print(scores)
```

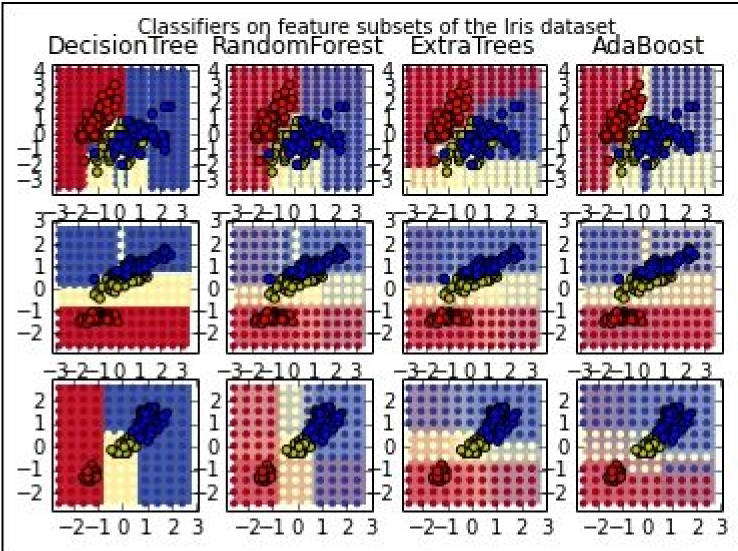
The scores, respectively, are as follows:

```
[ 0.74252492  0.82136895  0.75671141]
[ 0.88372093  0.9015025   0.8909396 ]
[ 0.91694352  0.93489149  0.91778523]
```

Given that we're working with entirely tree-based methods here, the score is simply the proportion of correctly-labeled cases. We can see here that there isn't much in it between the two forest methods, which both perform strongly with mean scores of *0.9*. In this example, random forest actually wins out marginally (on the order of an *0.002* increase) over ExtraTrees, while both techniques substantially outperform the basic decision tree, whose mean score sits at *0.77*.

One drawback when working with random forests (especially as the size of the forest increases) is that it can be hard to review the effectiveness of, or tune, a given implementation. While individual trees are extremely easy to work with, the sheer number of trees in a developed ensemble and the obfuscation created by random splitting can make it challenging to refine a random forest implementation. One option is to begin looking at the decision boundaries that individual models draw. By contrasting the models within one's ensemble, it becomes easier to identify where one model performs better at dividing classes than others.

In this example, for instance, we can easily see how our models perform at a high level without digging into specific details:



While it can be challenging to understand beyond a simple level (using high-level plots and summary scores) how a random forest implementation is performing, the hardship is worthwhile. Random forests perform very strongly with only a minimal cost in additional computation. They are very often a good technique to throw at a problem during the early stages, while one is still determining an angle of attack, because their ability to yield strong results fast can provide a useful benchmark. Once you know how a random forest implementation performs, you can begin to optimize and extend your ensemble.

To this end, we should continue exploring the different ensemble techniques so as to further build out our toolkit of ensembling options.

Applying boosting methods

Another approach to ensemble creation is to build boosting models. These models are characterized by their use of multiple models in sequence to iteratively "boost" or improve the performance of the ensemble.

Boosting models frequently use a series of weak learners, models that provide only marginal gain compared to random guessing. At each iteration, a new weak learner is trained on an adjusted dataset. Over multiple iterations, the ensemble is extended with one new tree (whichever tree optimized the ensemble performance score) at each iteration.

Perhaps the most well-known boosting method is **AdaBoost**, which adjusts the dataset at each iteration by performing the following actions:

- Selecting a decision stump (a shallow, often one-level decision tree, effectively the most significant decision boundary for the dataset in question)
- Increasing the weighting of cases that the decision stump labeled incorrectly, while reducing the weighting of correctly labeled cases

This iterative weight adjustment causes each new classifier in the ensemble to prioritize training the incorrectly labeled cases; the model adjusts by targeting highly-weighted data points. Eventually, the stumps are combined to form a final classifier.

AdaBoost can be used both in classification and regression contexts and achieves impressive results. The following example shows an AdaBoost implementation in action on the `heart` dataset:

```
import numpy as np

from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import AdaBoostClassifier
from sklearn.datasets.mldata import fetch_mldata
from sklearn.cross_validation import cross_val_score

n_estimators = 400
# A learning rate of 1. may not be optimal for both SAMME and SAMME.R
learning_rate = 1.

heart = fetch_mldata("heart")
X = heart.data
y = np.copy(heart.target)
y[y==-1]=0

X_test, y_test = X[189:], y[189:]
X_train, y_train = X[:189], y[:189]
```

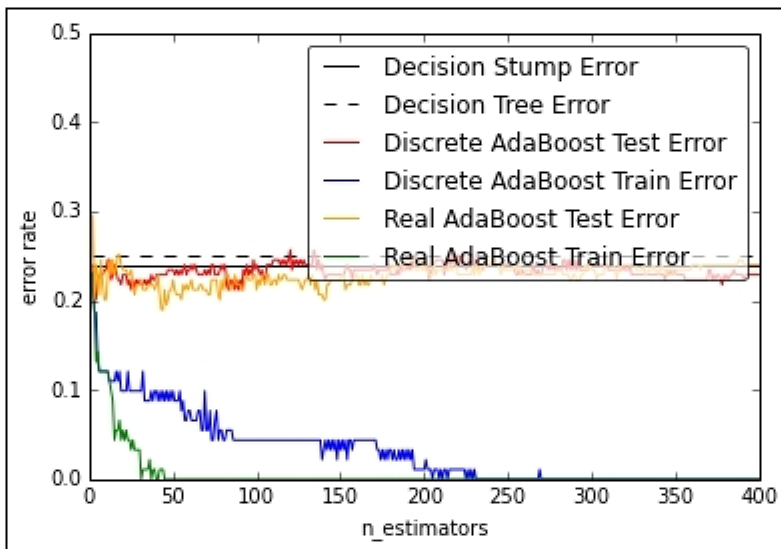
```
dt_stump = DecisionTreeClassifier(max_depth=1, min_samples_leaf=1)
dt_stump.fit(X_train, y_train)
dt_stump_err = 1.0 - dt_stump.score(X_test, y_test)

dt = DecisionTreeClassifier(max_depth=9, min_samples_leaf=1)
dt.fit(X_train, y_train)
dt_err = 1.0 - dt.score(X_test, y_test)

ada_discrete = AdaBoostClassifier(
    base_estimator=dt_stump,
    learning_rate=learning_rate,
    n_estimators=n_estimators,
    algorithm="SAMME")
ada_discrete.fit(X_train, y_train)

scores = cross_val_score(ada_discrete, X_test, y_test)
print(scores)
means = scores.mean()
print(means)
```

In this case, the `n_estimators` parameter dictates the number of weak learners used; in the case of averaging methods, adding estimators will always reduce the bias of your model, but will increase the probability that your model has overfit its training data. The `base_estimator` parameter can be used to define different weak learners; the default is decision trees (as training a weak tree is straightforward, one can use stumps, very shallow trees). When applied to the `heart` dataset, as in this example, AdaBoost achieved correct labeling in just over 79% of cases, a reasonably solid performance for a first pass:

```
[ 0.77777778  0.81481481  0.77777778]

0.79012345679
```

Boosting models provide a significant advantage over averaging models; they make it much easier to create an ensemble that identifies problem cases or types of problem cases and address them. A boosting model will usually target the easiest to predict cases first, with each added model fitting against a subset of the remaining incorrectly predicted cases.

One resulting risk is that a boosting model begins to overfit (in the most extreme case, you can imagine ensemble components that have fit to specific cases!) the training data. Managing the correct amount of ensemble components is a tricky problem but thankfully we can resort to a familiar technique to resolve it. In Chapter 1, *Unsupervised Machine Learning*, we discussed a visual heuristic called the **elbow method**. In that case, the plot was of K (the number of means), against a performance measure for the clustering implementation. In this case, we can employ an analogous process using the number of estimators (n) and the bias or error rate for the ensemble (which we'll call e). For a range of different boosting estimators, we can plot their outputs as follows:



By identifying a point at which the curve has begun to level off, we can reduce the risk that our model has overfit, which becomes increasingly likely as the curve begins to level off. This is true for the simple reason that as the curve levels, it necessarily means that the added gains from each new estimator are the correct classification of fewer and fewer cases!

Part of the appeal of a visual aid of this kind is that it enables us to get a feel for how likely our solution is to be overfitting. We can (and should!) be applying validation techniques wherever we can, but in some cases (for example, when aiming to hit a particular MVP target for a model implementation, whether that be informed by use cases or the distribution of scores on the Kaggle public leaderboard), we may be tempted to press forward with a performant implementation. Understanding exactly how attenuated the gains we're receiving are as we add each new estimator is critical to understanding the risk of overfitting.

Using XGBoost

In mid-2015, a new algorithm to solve structured machine learning problems, XGboost, has taken the competitive data science world by storm. **Extreme Gradient Boosting** (**XGBoost**) is a well-written, performant library that provides a generalized boosting algorithm (Gradient Boosting).

XGBoost works much like AdaBoost with one key difference—the means by which the model is improved is different.

At each iteration, XGBoost is seeking to improve the performance of the existing model set by reducing the residuals (the differences between targets and label predictions) of that ensemble. Every iteration, the model added is selected based on whether it is most able to reduce the existing ensemble's residuals. This is analogous to gradient descent (where a function is iteratively minimized by moving against a loss gradient); hence, the name Gradient Boosting.

Gradient Boosting has proven to be highly successful in recent Kaggle contests, where it has supported the winners of the CrowdFlower Competition and Microsoft Malware Classification Challenge, along with many other structured data competitions in the final half of 2015.

To apply XGBoost, let's grab the XGBoost library. The best way to get this is via `pip`, with the `pip install xgboost` command on the command line. For Windows users, `pip` installation is currently (late 2015) disabled on Windows. For your benefit, a cold copy of XGBoost is available in the `Chapter 8` folder of this book's GitHub repository.

Applying XGBoost is fairly straightforward. In this case, we'll apply the library to a multiclass classification task, using the UCI Dermatology dataset. This dataset contains an age variable and a large number of categorical variables. An example row of data looks like this:

```
3,2,0,2,0,0,0,0,0,0,0,0,1,2,0,2,1,1,1,0,0,0,1,0,0,0,0,0,0,0,0,1,0,10,
2
```

A small number of age values (penultimate feature) are missing, encoded by `?`. The objective in working with this dataset is to correctly classify one of six different skin conditions, per the following class distribution:

```
Database:  Dermatology

Class code:   Class:                  Number of instances:
1             psoriasis      112
2             seboreic dermatitis             61
3             lichen planus                   72
4             pityriasis rosea                49
5             cronic dermatitis               52
6             pityriasis rubra pilaris        20
```

We'll begin applying XGBoost to this problem by loading up the data and dividing it into test and train cases via a 70/30 split:

```
import numpy as np
import xgboost as xgb

data = np.loadtxt('./dermatology.data',
delimiter=',',converters={33: lambda x:int(x == '?'), 34: lambda
x:int(x)-1 } )
sz = data.shape

train = data[:int(sz[0] * 0.7), :]
test = data[int(sz[0] * 0.7):, :]


train_X = train[:,0:33]
train_Y = train[:, 34]
```

```
test_X = test[:,0:33]
test_Y = test[:, 34]
```

At this point, we initialize and parameterize our model. The `eta` parameter defines the step size shrinkage. In gradient descent algorithms, it's very common to use a shrinkage parameter to reduce the size of an update. Gradient descent algorithms have a tendency (especially close to convergence) to zigzag back and forth over the optimum; using a shrinkage parameter to downscale the size of a change makes the effect of gradient descent more precise. A common (and default) scaling value is `0.3`. In this example, `eta` has been set to `0.1` for even greater precision (at the possible cost of more iterations).

The `max_depth` parameter is intuitive; it defines the maximum depth of any tree in the example. Given six output classes, six is a reasonable value to begin with. The `num_round` parameter defines how many rounds of Gradient Boosting the algorithm will perform. Again, you typically require more rounds for a multiclass problem with more classes. The `nthread` parameter, meanwhile, defines how many CPU threads the code will run over.

The `DMatrix` structure used here is purely for the training speed and memory optimization. It's generally a good idea to use these while using XGBoost; they can be built from `numpy.arrays`. Using `DMatrix` enables the `watchlist` functionality, which unlocks some advanced features. In particular, `watchlist` allows us to monitor the evaluation results on all the data in the list provided:

```
xg_train = xgb.DMatrix( train_X, label=train_Y)
xg_test = xgb.DMatrix(test_X, label=test_Y)

param = {}

param['objective'] = 'multi:softmax'

param['eta'] = 0.1
param['max_depth'] = 6
param['nthread'] = 4
param['num_class'] = 6

watchlist = [ (xg_train,'train'), (xg_test, 'test') ]
num_round = 5
bst = xgb.train(param, xg_train, num_round, watchlist );
```

We train our model, `bst`, to generate an initial prediction. We then repeat the training process to generate a prediction with `softmax` enabled (via `multi:softprob`):

```
pred = bst.predict( xg_test );

print ('predicting, classification error=%f' % (sum( int(pred[i]) != 
test_Y[i] for i in range(len(test_Y))) / float(len(test_Y)) ))
```

```
param['objective'] = 'multi:softprob'
bst = xgb.train(param, xg_train, num_round, watchlist );

yprob = bst.predict( xg_test ).reshape( test_Y.shape[0], 6 )
ylabel = np.argmax(yprob, axis=1)

print ('predicting, classification error=%f' % (sum( int(ylabel[i])
!= test_Y[i] for i in range(len(test_Y))) / float(len(test_Y)) ))
```

Using stacking ensembles

The traditional ensembles that we saw earlier in this chapter all shared a common design philosophy: they involve multiple classifiers trained to fit a set of target labels and involve the models themselves being applied to generate some meta-function through strategies including model voting and boosting.

There is an alternative design philosophy as regards ensemble creation, known as stacking or, alternatively, as blending. Stacking involves multiple layers of models in a configuration where the output of one layer of models is used as training data for a model at the next layer. It's possible to blend hundreds of different models successfully.

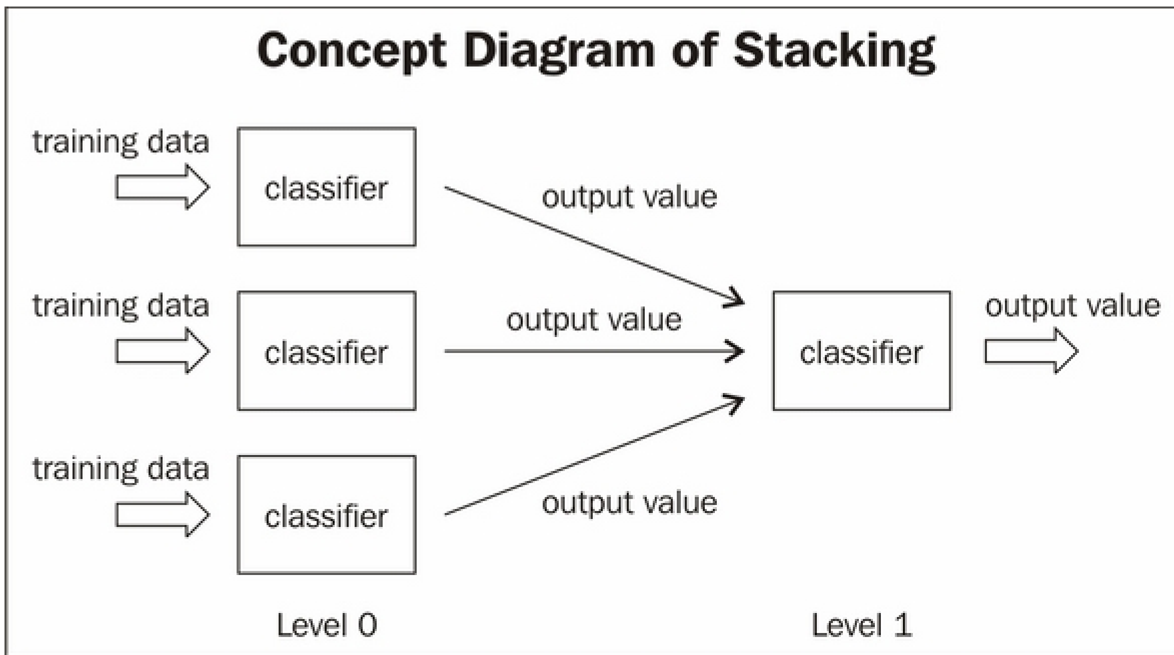
Stacking ensembles can also make up the blended set of features at a layer's output from multiple sub-blends (sometimes called **blend-of-blends**). To add to the fun, it's also possible to also extract particularly effective parameters from the models of a stacking ensemble and use them as meta-features, within blends or sub-blends at different levels.

All of this combines to make stacking ensembles a very powerful and extensible technique. The winners of the Kaggle Netflix prize (and associated $1 million award) used stacking ensembles over hundreds of features to great effect. They used several additional tricks to improve the effectiveness of their prediction:

- They trained and optimized their ensemble while holding out some data. They then retrained using the held-out data and again optimized before applying their model to the test dataset. This isn't an uncommon practice, but it yields good results and is worth keeping in mind.
- They trained using gradient descent and RMSE as the performance function. Crucially, they used the RMSE of the ensemble, rather than that of any of the models, as the relevant performance indicator (the measure of residuals). This should be considered a healthy practice whenever working with ensembles.
- They used model combinations that are known to improve on the residuals of other models. Neighborhood-based approaches, for instance, improve on the residuals of the RBM, which we examined earlier in this book. By getting to know the relative strengths and weaknesses of your machine learning algorithms, you can find ideal ensemble configurations.
- They calculated the residuals of their blend using k-fold cross-validation, another technique that we explored and applied earlier in this book. This helped overcome the fact that they'd trained their blend's constituent models using the same dataset as the resulting blend.

The main point to take away from the highly customized nature of the **Pragmatic Chaos** model used to win the Netflix prize is that a first-class model is usually the product of intensive iteration and some

creative network configuration changes. The other key takeaway is that the basic architectural pattern of a stacking ensemble is as follows:



Now that you've learned the fundamentals of how the stacking ensemble work, let's try applying them to solve data problems. To get us started, we'll use the `blend.py` code provided in the GitHub repository accompanying `Chapter 8`, . Versions of this blending code have been used by highly-scoring Kagglers across multiple contests.

To begin with, we'll examine how stacking ensembles can be applied to attack a real data science problem: the Kaggle contest *Predicting a Biological Response* aimed to build as effective a model as possible in order to predict the biological response of molecules given their chemical properties. We'll be looking at one particularly successful entry in this competition to understand how stacking ensembles can work in practice.

In this dataset, each row represents a molecule, while each of the 1,776 features describe characteristics of the molecule in question. The goal was to predict a binary response from the molecule in question, given these properties.

The code that we'll be applying comes from a competitor in that tournament who used a stacking ensemble to combine five classifiers: two differently configured random forest classifiers, two extra trees classifiers, and a gradient boosting classifier, which helps to yield slightly differentiated predictions from the other four components.

The duplicated classifiers were provided with different split criteria. One used the **Gini Impurity** (gini), a measure of how often a random record would be incorrectly labeled if it were randomly labeled according to the distribution of labels in the potential branch in question. The other tree used information gain (entropy), a measure of information content. The information content of a potential

branch can be measured by the number of bits that would be required to encode it. Using entropy as a measure to determine the appropriate split leads branches to become increasingly less diverse, but it's important to recognize that the entropy and `gini` criteria can yield quite different results:

```
if __name__ == '__main__':

    np.random.seed(0)

    n_folds = 10
    verbose = True
    shuffle = False

    X, y, X_submission = load_data.load()

    if shuffle:
        idx = np.random.permutation(y.size)
        X = X[idx]
        y = y[idx]

    skf = list(StratifiedKFold(y, n_folds))

    clfs = [RandomForestClassifier(n_estimators=100, n_jobs=-1,
criterion='gini'),
            RandomForestClassifier(n_estimators=100, n_jobs=-1,
criterion='entropy'),
            ExtraTreesClassifier(n_estimators=100, n_jobs=-1,
criterion='gini'),
            ExtraTreesClassifier(n_estimators=100, n_jobs=-1,
criterion='entropy'),
            GradientBoostingClassifier(learning_rate=0.05,
subsample=0.5, max_depth=6, n_estimators=50)]

    print "Creating train and test sets for blending."

    dataset_blend_train = np.zeros((X.shape[0], len(clfs)))
    dataset_blend_test = np.zeros((X_submission.shape[0], len(clfs)))

    for j, clf in enumerate(clfs):
        print j, clf
        dataset_blend_test_j = np.zeros((X_submission.shape[0],
len(skf)))
        for i, (train, test) in enumerate(skf):
            print "Fold", i
            X_train = X[train]
            y_train = y[train]
            X_test = X[test]
            y_test = y[test]
```

```
            clf.fit(X_train, y_train)
            y_submission = clf.predict_proba(X_test)[:,1]
            dataset_blend_train[test, j] = y_submission
            dataset_blend_test_j[:, i] =
clf.predict_proba(X_submission)[:,1]
        dataset_blend_test[:,j] = dataset_blend_test_j.mean(1)

    print
    print "Blending."
    clf = LogisticRegression()
    clf.fit(dataset_blend_train, y)
    y_submission = clf.predict_proba(dataset_blend_test)[:,1]

    print "Linear stretch of predictions to [0,1]"
    y_submission = (y_submission - y_submission.min()) /
(y_submission.max() - y_submission.min())

    print "Saving Results."
    np.savetxt(fname='test.csv', X=y_submission, fmt='%0.9f')
```

When we try running this submission on the private leaderboard, we find ourselves in a rather impressive 12th place (out of 699 competitors)! Naturally, we can't draw too many conclusions from a competition that we entered after completion, but, given the simplicity of the code, this is still a rather impressive result!

Applying ensembles in practice

One particularly important quality to be mindful of while applying ensemble methods is that your goal is to tune the performance of the ensemble rather than of the models that comprise it. Your approach should therefore be largely focused on building a strong ensemble performance score, rather than the strongest set of individual model performances.

The amount of attention that you pay to the models within your ensemble will vary. With an arrangement of differently configured or initialized models of a single type (for example, a random forest), it is sensible to focus almost entirely on the performance of the ensemble and metaparameters that shape it.

For more challenging problems, we frequently need to pay closer attention to the individual models within our ensemble. This is most obviously true when we're trying to create smaller ensembles for more challenging problems, but to build a truly excellent ensemble, it is often necessary to be considerate of the parameters and algorithms underlying the structure that you've built.

With this said, you'll always be looking at the performance of the ensemble as well as the performance of models within the set. You'll be inspecting the results of your models to try and work out *what each model did well*. You'll also be looking for the less obvious factors that affect ensemble performance, most notably the correlation of model predictions. It's generally recognized that a more effective ensemble will tend to contain performant but uncorrelated components.

To understand this claim, consider techniques such as correlation measures and PCA that we can use to measure the amount of information content present in dataset variables. In the same way, we can use Pearson's correlation coefficient against the predictions output by each of our models to understand the relationship between performance and correlation for each model.

Taking us back to stacking ensembles specifically, our ensemble's models are outputting metafeatures that are then used as inputs to a next-layer model. Just as we would vet the features used by a more conventional neural network, we want to ensure that the features output by our ensemble's components work well as a dataset. The calculation of the Pearson correlation coefficient across model outputs and use of the results in model selection is an excellent place to start in this regard.

When we deal with single-model problems, we almost always have to spend some time inspecting the problem and identifying an appropriate learning algorithm. If we're faced with a two-class classification problem with a moderate amount of features (*10's*) and labeled training cases, we might select a logistic regression, an SVM, or some other appropriate algorithm for the context. Different approaches will apply to different problems and through trial and error, parallel testing, and experience (both personal and posted online!), you will identify the appropriate approach for a specific objective given specific input data.

A similar logic applies to ensemble creation. Rather than identifying a single appropriate model, the challenge is to identify combinations of models that effectively describe different elements of an input dataset in such a way that the dataset as a whole is adequately described. By understanding the strengths and weaknesses of your component models as well as by exploring and visualizing your dataset, you'll be able to draw conclusions about how to develop your ensemble effectively through multiple iterations.

Ultimately, at this level, data science is a field with a great many techniques at hand. The best practitioners are able to apply their knowledge of their own algorithms and options to develop very effective solutions over many iterations.

These solutions involve the knowledge of algorithms and interaction of model combinations, model parameter adjustments, dataset translations, and ensemble manipulation. Just as importantly, they require an uninhibited and creative mindset.

One good example of this is the work of prominent Kaggle competitor, Alexander Guschin. Focusing on one specific example—the **Otto Product Classification** contest—can give us an idea as to the range of options available to a confident and creative data scientist.

Most model development processes begin with a period in which you throw different solutions at the problem, attempting to find the tricks underlying the data and figuring out what works. Settling on a stacking model, Alexander set about building metafeatures. While we looked at XGBoost as an ensemble in its own right, in this case it was used as a component to the stacking ensemble in order to generate some of the metafeatures to be used by the final model. Neural networks were used in addition to the gradient boosted trees as both algorithms tend to produce good results.

To add some contrast to the mixture, Alexander added a KNN implementation, specifically because the results (and therefore the metaparameters) generated by a KNN tend to differ significantly from the models already included. This approach of picking up components whose outputs tend to differ is crucial in creating an effective stacking ensemble (and to most ensemble types).

To further develop this model, Alexander added some custom elements to the second layer of his model. While combining the XGBoost and neural network predictions, he also added bagging at this layer. At this point, most of the techniques that we've discussed in this chapter have shown up in some part of this model. In addition to the model development, some feature engineering (in particular, the use of TF-IDF on half of the training and test data) and the use of plotting techniques to identify class differentiation were used throughout.

A truly mature model that can tackle the most significant data science challenges is one that combines the techniques we've seen throughout this book, created using a solid understanding of the underlying algorithms and the possibilities for how these techniques can interact with each other.

This book so far has taught many of the fundamentals—the base of practical knowledge—that a practitioner has to collect. It has used many examples and an increasing amount of real-world cases to demonstrate how a broad base of knowledge becomes increasingly powerful in letting you develop effective solutions to difficult problems.

What's required of you as a data scientist is to first apply this broad set of techniques to develop an experience of how they can perform and what they could do for you. Then it is up to you to develop that creativity and experimental mindset that distinguishes some of the best data scientists.

Using models in dynamic applications

We've spent this chapter discussing the use of techniques to manage model performance under conditions that might be seen as ideal; specifically, conditions in which all of the data is available ahead of time so that a model can be trained on all data. These assumptions are frequently valid in research contexts or when dealing with one-time problems, but in many contexts they are unsafe assumptions. The range of unsafe contexts goes beyond the cases where the data is simply unavailable, such as data science contests where a held-out dataset is used to establish the final leaderboard.

Returning to a subject from earlier in this chapter, you'll recall the Pragmatic Chaos algorithm, which won the Netflix prize? By the time Netflix came to assessing the algorithm for implementation, both the business context and requirements had shifted so dramatically that the minimal accuracy gains provided by that algorithm didn't justify implementation costs. The $1M algorithm was redundant and was never implemented in production! The point to take from this example is that in commercial contexts, it is critical for our models to have as much adaptability as we can provide.

The really challenging applications of machine learning algorithms, in which our existing run once methodologies become less valuable, are ones where real data changes occur across time (or other dimensions). In these contexts, one knows that a substantial data change will occur and that existing models cannot be easily trained to adapt to this data change. At that point, new techniques are needed as well as new information.

To adapt and gather this information, we need to become better able to predict the ways in which data change is liable to occur. With this information, our model building and the content of our ensembles can start to change in order to cover the most likely data change scenarios that we see ahead. This adaptation lets us pre-empt data change and reduce the adjustment time required. As we'll see later in this chapter, in real-world applications any reduction in the time it takes us to pivot based on data change is valuable.

In the next section, we'll be looking at tools that we can use to make our models more robust to changing data. We'll discuss the means by which we can maintain a broad set of model options, simultaneously accommodating one or multiple data change scenarios, without reducing the performance of our models.

Understanding model robustness

It's important to understand exactly what the problem is here and how and when it is presented. This involves defining two things; the first being robustness as it applies to machine learning algorithms. The second, of course, is data change. Some of the content in the first part of this section is at an introductory level, but experienced data scientists may still find value in reviewing the section!

In academic terms, the robustness of a machine learning algorithm is the property that characterizes how effective your algorithm is while being applied to a dataset other than the dataset on which it was trained.

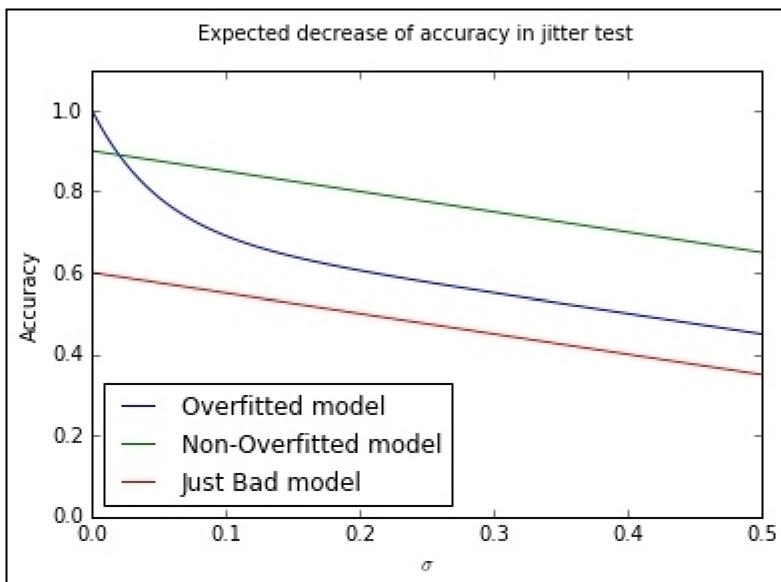
Robustness testing is a core part of machine learning methodology in any context. The importance of validation techniques such as k-fold cross-validation and the use of tests when developing models for even the simplest contexts is a consequence of machine learning algorithm vulnerability to data change.

Most datasets contain both a signal and noise. Noise may be predictable (and thus more easily managed) or it may be stochastic and difficult to treat. A dataset may contain more or less noise. Typically, datasets with more or less predictable noise are harder to train and test against the same datasets with this noise removed (which can be easily tested).

When one has trained a model on a given dataset, it is almost inevitable that this model has learned based on both the signal and noise. The concept of overfitting is generally used to describe a model that has fit so well to a given dataset that it has learned to predict based on both the signal and noise, rendering it less powerful against other samples than a model with a less exact fit.

Part of the goal of training a model is to reduce the impact of any local noise on learning as much as possible. The purpose of validation techniques that hold out a set of data to test is to ensure that any learning of noise during training happens only on noise that is local to the training set. The difference between training and test error can be used to understand the degree of overfitting between model implementations.

We've applied cross-validation in Chapter 1, *Unsupervised Machine Learning*. Another useful means of testing models for the overfitting is to directly add random noise in the form of jitter to the training dataset. This technique was introduced via a Kaggle notebook in October 2015 by Alexander Minushkin and offers a very interesting test. The concept is simple; by adding jitter and looking at the accuracy of prediction on the training data, we can distinguish an overfitted model (whose training error will increase more quickly as we add jitter) from a well- or poorly-fitted model:



In this case, we're able to plot the results of a jitter test to easily identify whether a model has overfit. From a very strong initial position, an overfit model will typically rapidly decline in performance as small amounts of jitter are added. For better-fitting models, the loss in performance with added jitter is much reduced, with the degree of overfitting in a model being particularly obvious at low levels of added jitter (where a well-fit model will tend to outperform an overfit counterpart).

Let's look at how we implement a jitter test for overfitting. We use a familiar score, `accuracy_score`, defined as the proportion of class labels predicted correctly, as the basis for test scoring. Jitter is defined by simply adding random noise to the data (using `np.random.normal`) with the amount of noise defined by the configurable `scale` parameter:

```
from sklearn.metrics import accuracy_score

def jitter(X, scale):
    if scale > 0:
        return X + np.random.normal(0, scale, X.shape)
    return X

def jitter_test(classifier, X, y, metric_FUNC = accuracy_score,
sigmas = np.linspace(0, 0.5, 30), averaging_N = 5):
    out = []

    for s in sigmas:
        averageAccuracy = 0.0
        for x in range(averaging_N):
            averageAccuracy += metric_FUNC( y,
classifier.predict(jitter(X, s)))

        out.append( averageAccuracy/averaging_N)

    return (out, sigmas, np.trapz(out, sigmas))

allJT = {}
```

The `jitter_test` itself is defined as a wrapper to normal sklearn classification, given a classifier, training data, and a set of target labels. The classifier is then called to predict against a version of the data that first has the `jitter` operation called against it.

At this point, we'll begin creating a number of datasets to run our jitter test over. We'll use sklearn's `make_moons` dataset, commonly used as a dataset to visualize clustering and classification algorithm performance. This dataset is comprised of two classes whose data points create interleaving half-circles. By adding varying amounts of noise to `make_moons` and using differing amounts of samples, we can create a range of example cases to run our jitter test against:

```
import sklearn
import sklearn.datasets

import warnings
warnings.filterwarnings("ignore", category=DeprecationWarning)

Xs = []
ys = []
```

```
#low noise, plenty of samples, should be easy
X0, y0 = sklearn.datasets.make_moons(n_samples=1000, noise=.05)
Xs.append(X0)
ys.append(y0)

#more noise, plenty of samples
X1, y1 = sklearn.datasets.make_moons(n_samples=1000, noise=.3)
Xs.append(X1)
ys.append(y1)

#less noise, few samples
X2, y2 = sklearn.datasets.make_moons(n_samples=200, noise=.05)
Xs.append(X2)
ys.append(y2)

#more noise, less samples, should be hard
X3, y3 = sklearn.datasets.make_moons(n_samples=200, noise=.3)
Xs.append(X3)
ys.append(y3)
```

This done, we then create a `plotter` object that we'll use to show our models' performance directly against the input data:

```
def plotter(model, X, Y, ax, npts=5000):

    xs = []
    ys = []
    cs = []
    for _ in range(npts):
        x0spr = max(X[:,0])-min(X[:,0])
        x1spr = max(X[:,1])-min(X[:,1])
        x = np.random.rand()*x0spr + min(X[:,0])
        y = np.random.rand()*x1spr + min(X[:,1])
        xs.append(x)
        ys.append(y)
        cs.append(model.predict([x,y]))
    ax.scatter(xs,ys,c=list(map(lambda x:'lightgrey' if x==0 else
'black', cs)), alpha=.35)
    ax.hold(True)
    ax.scatter(X[:,0],X[:,1],
               c=list(map(lambda x:'r' if x else 'lime',Y)),
               linewidth=0,s=25,alpha=1)
    ax.set_xlim([min(X[:,0]), max(X[:,0])])
    ax.set_ylim([min(X[:,1]), max(X[:,1])])
    return
```

We'll use an SVM classifier as the base model for our jitter tests:

```
import sklearn.svm
classifier = sklearn.svm.SVC()

allJT[str(classifier)] = list()

fig, axes = plt.subplots(nrows=2, ncols=2, figsize=(11,13))
i=0
for X,y in zip(Xs,ys):
    classifier.fit(X,y)
    plotter(classifier,X,y,ax=axes[i//2,i%2])
    allJT[str(classifier)].append (jitter_test(classifier, X, y))
    i += 1
plt.show()
```

The jitter test provides an effective means of assessing model overfitting and performs comparably to cross-validation; indeed, Minushkin provides evidence that it can outperform cross-validation as a tool to measure model fit quality.

Both of these tools to mitigate the overfitting work well in contexts where your algorithm is either run over data on a one-off basis or where underlying trends don't vary substantially. This is true for the majority of single-dataset problems (such as most academic or web repository datasets) or data problems where the underlying trends change slowly.

However, there are many contexts where the data involved in modeling might change over time in one or several dimensions. This can occur because of change in the methods by which data is captured, usually because new instruments or techniques come into use. For instance, video data captured by commonly-available devices has improved substantially in resolution over the decade since 2005 and the quality (and size!) of such data has increased. Whether you're using the video frames themselves or instead the file size as a parameter, you'll observe noticeable shifts in the nature, quality, and distributions of features.

Alternatively, changes in dataset variables might be caused by differences in underlying trends. The classic data schema concept of measures and dimensions comes back into play here, as we can better understand how data change is affected by considering what dimensions influence our measurement.

The key example is time. Depending on context, many variables are subject to day-of-week, month-of-year, or seasonal variations. In many cases, a helpful option might be to parameterize these variables, (as we discussed in the preceding chapter, techniques such as one-hot encoding can help our algorithms learn to parse such trends) particularly if we're dealing with periodic trends that are easily predicted (for example, the impact of month-of-year on scarf sales in a given location) and easily modeled.

A more problematic type of time series trend is non-periodic change. As in the preceding video camera example, some types of time series trends change irrevocably and in ways that might not be trivial to predict. Telemetry from software tends to be influenced by the quality and functionality of the software build live at the time the telemetry was emitted. As builds change over time, the values sent in telemetry and the variables created from those values can change radically overnight in hard-to-predict ways.

Human behavior, a hugely important factor in many datasets, helpfully changes both periodically and non-periodically. People shop more around seasonal holidays, but also change their shopping habits permanently based on new societal or technological developments.

Some of the added complexity here comes not just from the fact that single variables and their distributions are affected by time series trends, but also from how relationships between relevant factors and their associated variables will change. The relationships between variables may change in quantifiable terms. One example is how, for humans, height and weight are two variables whose relationship varies between times and locations. The BMI feature, which we might use to track this relationship, shows differing distributions when sampled across periods of time or between locations.

Furthermore, variables can change in another serious way; namely, their importance to a performant modeling algorithm may vary over time! Some variables whose values are highly relevant in some periods of time will be less relevant in others. As an example, consider how climate and weather variables affect agriculture markets. For some crops and the companies dealing in them, these variables are fairly unimportant for much of the year. At the time of crop growth and harvest, however, they become fundamentally important. To make this more complex, the strength of these factors' importance is also tied to location (and local climate).

The challenge for modeling is clear. For models that are trained once and run again on new data, managing data change can present serious challenges. For models that are dynamically recomputed based on new input data, data change can still create problems as variable distributions and relationships change and available variables become more or less valuable in generating an effective solution.

Part of the key to successfully managing data change in your application of ML is to recognize the dimensions (and there are common culprits) where change is probable and liable to affect the distributions of your features, relationships, and feature importance, which a model will attempt to pick up on.

Once you have an understanding as to what the factors in your data are that are likely to influence overfitting, you're better positioned to develop a solution that manages these factors effectively.

This said, it will still seem hugely challenging to build a single model that can resolve any potential issues. The simple response to this is that if one faces serious data change issues, the solution probably isn't to try to solve for them with a single model! In the next section, we'll be looking at ensemble methods to provide a better answer.

Identifying modeling risk factors

While it is in many cases quite straightforward to identify which elements present a risk to your model over time, it can help to employ a structured process for identification. This section briefly describes some of the heuristics and techniques you can employ to screen your models for the risk of data change.

Most data scientists keep a data dictionary for datasets that are intended for general use or automated applications. This is especially likely to happen if the data or applications are complex, but keeping a data dictionary is generally good practice. Some of the most effective work you can do in identifying risk factors is to run through these features and tag them based on different risk types.

Some of the tags that I tend to use include the following:

- **Longitudinally variant**: Is this parameter liable to change over a long time due to longitudinal trends that many not be fully visible in the span of the training data that you have available? The most obvious example is the ecological seasons, which affect many areas of human behavior as well as the many things that depend on some more fundamental climatic variables. Other longitudinal trends include the financial year and the working month, but extend to include many other longitudinal trends relevant to your area of investigation. The life cycle of new iPhone models or the population flux of voles might be an important longitudinal factor depending on the nature of your work.
- **Slowly changing**: Is this categorical parameter likely to gain new values over time? This concept is borrowed from data warehousing best practices. A slowly changing dimension in the classical sense will gain new parameter codes (for example, as a new store opens or a new case is identified). These can throw your model entirely if not managed properly or if they appear in sufficient number. Another impact of slowly changing data, which can be more problematic to handle, is that it can begin to affect the distribution of your features. This can have a substantial impact on the effectiveness of your model.
- **Key parameter**: A combination of data value monitoring and recalculation of decision boundaries/regression equations will often handle a certain amount of slowly changing data and seasonal variance well, but consider taking action should you see an unexpectedly large amount of new cases or case types, especially when they affect variables depended on heavily by your model. For this reason, also make sure that you know which variables are most relied upon by your solution!

The process of tagging in this way is helpful (not least as an export of your own memory) mostly because it helps you to do the following:

- Organize your expectations and develop a kind of checklist for your development of monitoring readiness. If you aren't able to keep track of at least your longitudinally variant and slowly changing parameter change, you are effectively blind to any output from your model besides changes in the parameters that it favors when recomputed and its (likely slowly declining) performance measure.
- Investigate mitigation (for example, improved normalization or extra parameters that codify those dimensions in which your data is variant). In many ways, mitigation and the addition of parameters is the best solution you can tap to handle data change.
- Set up robustness testing using constructed datasets, where your risk features are deliberately varied to simulate data change. Stress-test your model under these conditions and find out exactly how much variance it'll tolerate. With this information, you can easily set yourself up to use your monitoring values as an early alert system; once data change exceeds a certain safe threshold, you know how much degradation to expect in the model performance.

Strategies to managing model robustness

We've discussed a number of effective ensemble techniques that allow us to balance the twin needs for performant and robust models. However, throughout our exposition and use of these techniques, we had to decide how and when we would reduce our model's performance to improve robustness.

Indeed, a common theme in this chapter has been how to balance the conflicting objectives of creating an effective, performant model, without making this model too inflexible to respond to data change. Many of the solutions that we've seen so far have required that we trade-off one outcome against the other, which is less than ideal.

At this point, it's worth our taking a slightly wider view of our options and drawing from complimentary techniques. The need for robust, performant statistical models within evolving business landscapes is neither new nor untreated; fields such as credit risk modeling have a long history of applied statistical modeling in changing domains and have developed effective decision management methodologies in order to succeed. Data scientists can turn some of these established techniques to our own benefit via using them to help organize our own models.

One effective methodology is **Champion/Challenger**, a test-centric approach that involves running multiple, parallel model configurations. In addition to the model whose outputs are applied (to direct business activities or inform reporting), champion/challenger approaches training one or more alternative model configurations.

By maintaining and monitoring multiple models, one can arrange to substitute the current model as and when an alternative outperforms it. This is usually done by maintaining a performance scoring process for all models and observing the results so that a manual decision call can be made about whether and when to switch to a challenger.

While the simplest implementation may involve switching to a challenger as soon as it outperforms the main model, this is rarely done as there are risks around specific challenger models being exposed to local minima (for example, the day-of-week or month-of-year local trends). It is normal to spend a significant period assessing a challenger model, particularly ahead of sensitive applications. In complex real cases, one may even want to do additional testing by providing a sample of treatment cases to a promising challenger to determine whether it generates significant lift over the champion.

There is scope for some creativity beyond simple, "replace the challenger" succession rules. Voting-based approaches are quite common, where a top subset of the trained ensembles provides scores on a case-by-case basis and those scores treated as (weighted or unweighted) votes. Another approach involves using a **Borda count**, a voting system where each voter ranks the candidate solutions in order of preference. In the context of ensembling, one would typically assign each individual model's prediction a point value equal to its inverse rank (keeping each model separate!). Then one can combine these votes (usually experimenting with a range of different weightings) to generate a result.

Voting can perform fairly well with a larger number of models but is dependent on the specific modeling context and factors like the similarity of the different voters. As we discussed earlier in this chapter, it's critical to use tests such as Pearson's correlation coefficient to ensure that your model set is both performant and uncorrelated.

One may find that particular classes of input data (users, say, with specific segmentation tags) are more effectively treated by a given challenger and may implement a case routing system where multiple champions deal with different user subgroups. This approach overlaps somewhat with the benefits of boosting ensembles, but can help in production circumstances by separating concerns. However,

maintaining multiple champions will increase the monitoring and oversight burden for your data team, so this option is best avoided if not entirely necessary.

A major concern to address is how we go about scoring our models, not least because there are immediate practical challenges. In particular, it is hard to compare multiple models in real contexts, given that class labels (to guide correctness) typically aren't available. In predictive contexts, this problem is compounded by the fact that the champion model's predictions are typically used to take actions that alter predicted events. This activity makes it very difficult to make assertions about how a challenger model's predictions would've performed; by taking action based on our champion's predictions, we're unable to confirm the results of our models!

The most common implementation process is to provide each challenger model with a statistically viable sample of the input data and then compare the lift from each approach. This approach inherently limits the number of challengers that one can support for some modeling problems. Another option is to leave just one statistically viable sample out of any treatment activity and use it to create a single regression test. This test is applied to the entire set of champion and challenger models, providing a meaningful basis for comparison.

The downside to this approach is that the change to a more effective model will always trail the data change by however long it takes to generate correct class labels for the test cases. While in many cases this isn't crippling (the champion model remains in place for the period it takes to generate accurate models), it can present problems in contexts where underlying conditions change rapidly compared to the training time for models.

Note

It's worth making one brief comment on the relationship between model training time and data change frequency. It isn't always clearly stated as such, but the typical goal in applied machine learning contexts is to reduce the factor of training time to data change frequency to the smallest value possible. To take the worst case, if the length of time it takes to train a model is longer than the length of time that model will be accurate for (and the ratio is equal to or greater than one), your model will never generate current results that can directly drive current actions. In general, a high ratio should prompt review and adjustment activities (either an investigation into whether faster score delivery at lower confidence delivers more value or adjustment to the rate at which controllable environment variables change).

The smaller this ratio becomes, the more leeway your team has to apply your model's outputs to drive actions and generate value. Depending on how variant and quantifiable this ratio is for your modeling context, it can be a useful concept to promote within your organization as a health measure for your automated modeling solution.

These alternative models may simply be the next best-performing ensemble configurations; they may be older models, kept around for observation. In sophisticated operations, some challengers are configured to handle different *what-if* scenarios (for example, *what if the temperature in this region is 2 C below expectations* or *what if sales are significantly below expectations*). These models may have been trained on the same data as the main model or on deliberately skewed or prepared data that simulates the what-if scenario.

More challengers tend to be better (providing improved robustness and performance), provided that the challengers are not all minute variations on the same theme. Challenger models also provide a safe venue for innovation and testing, while observing effective challengers can provide useful insights into how robust your champion ensemble is likely to be to a range of possible environmental changes.

The techniques that you've learned to apply in this section have provided us with the tools to apply our existing toolkit of models to real applications in evolving environments. This chapter also discussed complications that can arise when applying ML models to production; data change, between samples or across dimensions, will cause our models to become increasingly ineffective. By thoroughly unpacking the concept of data change, we became better able to characterize this risk and recognize where and how it might present itself.

The remainder of the chapter was dedicated to techniques that provide improved model robustness. We discussed how to identify model degradation risk by looking at the underlying data and discussed some helpful heuristics to this end. We drew from existing decision management methods to learn about and use Champion/Challenger, a well-regarded process with a long history in contexts including applied machine learning. Champion/Challenger helps us organize and test multiple models in healthy competition. In conjunction with effective performance monitoring, a proactive tactical plan for model substitution will give you faster and more controllable management of the model life cycle and quality, all the while providing a wealth of valuable operational insights.

Further reading

Perhaps the most wide-ranging and informative tour of Ensembles and ensemble types is provided by the Kaggle competitor, Triskelion, at http://mlwave.com/kaggle-ensembling-guide/.

For discussion of the Netflix Prize-winning model, Pragmatic Chaos, refer to http://www.stat.osu.edu/~dmsl/GrandPrize2009_BPC_BellKor.pdf. For an explanation by Netflix on how changing business contexts rendered that $1M-model redundant, refer to the Netflix Tech blog at http://techblog.netflix.com/2012/04/netflix-recommendations-beyond-5-stars.html.

For a walkthrough on applying random forest ensembles to commercial contexts, with plenty of space given to all-important diagnostic charts and reasoning, consider Arshavir Blackwell's blog at https://citizennet.com/blog/2012/11/10/random-forests-ensembles-and-performance-metrics/.

For further information on random forests specifically, I find the scikit-learn documentation helpful: http://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html.

A great introduction to gradient-boosted trees is provided within the XGBoost documentation at http://xgboost.readthedocs.io/en/latest/model.html.

For a write-up of Alexander Guschin's entry to the Otto Product Classification challenge, refer to the No Free Hunch blog: http://blog.kaggle.com/2015/06/09/otto-product-classification-winners-interview-2nd-place-alexander-guschin/.

Alexander Minushkin's Jitter test for overfitting is described at https://www.kaggle.com/miniushkin/introducing-kaggle-scripts/jitter-test-for-overfitting-notebook.

Summary

In this chapter, we covered a lot of ground. We began by introducing ensembles, some of the most powerful and popular techniques in competitive machine learning contexts. We covered both the theory and code needed to apply ensembles to our machine learning projects, using a combination of expert knowledge and practical examples.

In addition, this chapter also dedicates a section to discussing the unique considerations that arise when you run models for weeks and months at a time. We discussed what data change can mean, how to identify it, and how to think about guarding against it. We gave specific consideration to the question of how to create sets of models running in parallel, which you can switch between based on seasonal change or performance drift in your model set.

During our review of these techniques, we spent significant time with real-world examples with the specific aim of learning more about the creative mindset and broad range of knowledge required of the best data scientists.

The techniques throughout this book have led up to a point that, armed with technical knowledge, code to reapply, and an understanding of the possibilities, you are truly able to take on any data modeling challenge.

Chapter 9. Additional Python Machine Learning Tools

Over the course of the eight preceding chapters, we have examined and applied a range of techniques that help us enrich and model data for many applications.

We approached the content in these chapters using a combination of Python libraries, particularly NumPy and Theano, while the other libraries were drawn upon as and when we needed to access specific algorithms. We did not spend a great deal of time discussing what other options existed in terms of tools, what the unique differentiators of these tools were, or why we might be interested.

The primary goal of this final chapter is to highlight some other key libraries and frameworks that are available to you to use. These tools streamline and simplify the process of creating and applying models. This chapter presents these tools, demonstrates their application, and provides extensive advice regarding *Further reading*.

A major contributor to succeed in solving data science challenges and being successful as a data scientist is having a good understanding of the latest developments in algorithms and libraries. As professionals, data scientists tend to be highly dependent on the quality of the data they use, but it is also very important to have the best tools available.

In this chapter, we will review some of the best in the recent tools available to data scientists, identifying the benefits they offer, and discussing how to apply them alongside tools and techniques discussed earlier in this book within a consistent working process.

Alternative development tools

Over the last couple of years, a number of new machine learning frameworks have emerged that offer advantages in terms of workflow. Usually these frameworks are highly focused on a specific use case or objective. This makes them very useful, perhaps even must-have tools, but it also means that you may need to use multiple workflow improvement libraries.

With an ever-growing set of new Python ML projects being lit up to address specific workflow challenges, it's worth discussing two libraries that add to our existing workflow and which accelerate or improve the work we've done in the preceding chapters. In this chapter, we'll be introducing **Lasagne** and **TensorFlow**, discussing the code and capabilities of each library and identifying why each framework is worth considering as a part of your toolset.

Introduction to Lasagne

Let's face it; sometimes creating models in Python takes longer than we'd like. However, they can be efficient for models that are more complex and offer big benefits (such as GPU acceleration and configurability) libraries similar to Theano can be relatively complex to use when working on simple cases. This is unfortunate because we often want to work with simple models, for instance, when we're setting up benchmarks.

Lasagne is a library developed by a team of deep learning and music data mining researchers to work as an interface to Theano. It is designed specifically to nail a particular goal—to allow for fast and efficient prototyping of new models.

This focus dictated how Lasagne was created, to call Theano functions and return Theano expressions or `numpy` data types, in a much less complex and more easily understood manner than the same operations written in native Theano code.

In this section, we'll take a look at the conceptual model underlying Lasagne, apply some Lasagne code, and understand what the library adds to our existing practices.

Getting to know Lasagne

Lasagne operates using the concept of layers, a familiar concept in machine learning. A layer is a set of neurons and operating rules that will take an input and generate a score, label, or other transformations. Neural networks generally function as a set of layers that feed input data in at one end and push output values out at the other (though the ways in which this gets done vary broadly).

It has become very popular in deep learning contexts to start treating individual layers as first class citizens. Traditionally, in machine learning work, a network would be established from layers using only a few parameter specifications (such as node count, bias, and weight values).

In recent years, data scientists seeking that extra edge have begun to take increasing interest in the configuration of individual layers. Nowadays it is not unusual in advanced machine learning environments to see layers that contain sub-models and transformed inputs. Even features, nowadays, might skip layers as needed and new features may be added to layers partway through a model. As an example of some of this refinement, consider the convolutional neural network architectures employed by Google to solve image recognition challenges. These networks are extensively refined at a layer level to generate performance improvements.

It therefore makes sense that Lasagne treats layers as its basic model component. What Lasagne adds to the model creation process is the ability to stack different layers into a model quickly and intuitively. One may simply call a class within `lasagne.layers` to stack a class onto your model. The code for this is highly efficient and looks as follows:

```
l0 = lasagne.layers.InputLayer(shape=X.shape)

l1 = lasagne.layers.DenseLayer(
l0, num_units=10, nonlinearity=lasagne.nonlinearities.tanh)

l2 = lasagne.layers.DenseLayer(l1, num_units=N_CLASSES,
nonlinearity=lasagne.nonlinearities.softmax)
```

In three simple statements, we have created the basic structure of a network using simple and configurable functions.

This code creates a model using three layers. The layer `l0` calls the `InputLayer` class, acting as an input layer for our model. This layer translates our input dataset into a Theano tensor, based on the expected shape of the input (defined using the `shape` parameter).

The next layers, `l1` and `l2` are each fully connected (dense) layers. Layer `l2` is defined as an output layer, with a number of units equal to the number of classes, while `l1` uses the same `DenseLayer` class to create a hidden layer of `10` units.

In addition to configuration of the standard parameters (weights, biases, unit count and nonlinearity type) available to the `DenseLayer` class, it is possible to employ entirely different network types using different classes. Lasagne provides classes for a broad set of familiar layers, including dense, convolutional and pooling layers, recurrent layers, normalisation and noise layers, amongst others. There is, furthermore, a special-purpose layer class, which provides a range of additional functionality.

If something more bespoke than what these classes provide is needed, of course, the user can resort to defining their own layer type easily and use it in conjunction with other Lasagne classes. However, for a majority of prototyping and fast, iterative development contexts, this is a great amount of pre-prepared capability.

Lasagne provides a similarly succinct interface to define the loss calculation for a network:

```
true_output = T.ivector('true_output')
objective = lasagne.objectives.Objective(l2,
loss_function=lasagne.objectives.categorical_crossentropy)

loss = objective.get_loss(target=true_output)
```

The `loss` function defined here is one of the many available functions, including squared error, hinge loss for binary and multi-class cases, and `crossentropy` functions. An accuracy scoring function for validation is also provided.

With these two components, a `loss` function and a network architecture, we again have everything we need to train a network. To do this, we need to write a little more code:

```
all_params = lasagne.layers.get_all_params(l2)
updates = lasagne.updates.sgd(loss, all_params, learning_rate=1)
train = theano.function([l0.input_var, true_output], loss,
updates=updates)

get_output = theano.function([l0.input_var], net_output)

for n in xrange(100):
 train(X, y)
```

This code leverages the `theano` functionality to train our example network, using our `loss` function, to iteratively train to classify a given set of input data.

Introduction to TensorFlow

When we reviewed Google's take on the **convolutional neural network** (**CNN**) in Chapter 4, *Convolutional Neural Networks*, we found a convoluted, many-layered beast. The question of how to create and monitor such networks only became more important as the network scales in layer count and complexity to attack challenges that are more complex.

To address this challenge, the Machine Intelligence research organisation at Google developed and distributed a library named TensorFlow, which exists to enable easier refinement and modeling of very involved machine learning models.

TensorFlow does this by providing two main benefits; a clear and simple programming interface (in this case, a Python API) onto familiar structures (such as NumPy objects), and powerful diagnostic and graph visualisation tools, such as **TensorBoard**, to enable informed tuning of a data architecture.

Getting to know TensorFlow

TensorFlow enables a data scientist to design data transformation operations as a flow across a computation graph. This graph can be extended and modified, while individual nodes can be tuned extensively, enabling detailed refinements of individual layers or model components. The TensorFlow workflow typically involves two phases. The first of these is referred to as the construction phase, during which a graph is assembled.

During the construction phase, we can write code using the Python API for Tensorflow. Like Lasagne, TensorFlow offers a relatively simple interface to writing network layers, requiring simply that we specify weights and bias before creating our layers. The following example shows initial setting of weight and bias variables, before creating (using one line of code each) a convolutional layer and a simple max-pooling layer. Additionally, we use `tf.placeholder` to generate placeholder variables for our input data.

```
x = tf.placeholder(tf.float32, shape=[None, 784])
y_ = tf.placeholder(tf.float32, shape=[None, 10])

W = tf.Variable(tf.zeros([5, 5, 1, 32]))
b = tf.Variable(tf.zeros([32]))


h_conv = tf.nn.relu(conv2d(x_image, W) + b)
h_pool = max_pool_2x2(h_conv)
```

This structure can be extended to include a `softmax` output layer, just as we did with Lasagne.

```
W_out = tf.Variable(tf.zeros([1024,10]))
B_out = tf.Variable(tf.zeros([10]))

y = tf.nn.softmax(tf.matmul(h_conv, W_out) + b_out)
```

Again, we can see significant improvements in the iteration time over writing directly in Theano and Python libraries. Being written in C++, TensorFlow also provides performance gains over Python, providing advantages in execution time.

Next up, we need to train and evaluate our model. Here, we'll need to write a little code to define our `loss` function for training (cross entropy, in this case), an `accuracy` function for validation and an optimisation method (in this case, steepest gradient descent).

```
cross_entropy = tf.reduce_mean(-tf.reduce_sum(y_ * tf.log(y),
reduction_indices=[1]))

train_step =
tf.train.GradientDescentOptimizer(0.5).minimize(cross_entropy)

correct_prediction = tf.equal(tf.argmax(y_,1), tf.argmax(y_,1))

accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
```

Following this, we can simply begin running our model iteratively. This is all succinct and very straightforward:

```
sess.run(tf.initialize_all_variables())
for i in range(20000):
  batch = mnist.train.next_batch(50)
  if i%100 == 0:
    train_accuracy = accuracy.eval(feed_dict={
        x:batch[0], y_: batch[1], keep_prob: 1.0})
    print("step %d, training accuracy %g"%(i, train_accuracy))
  train_step.run(feed_dict={x: batch[0], y_: batch[1], keep_prob:
0.5})

print("test accuracy %g"%accuracy.eval(feed_dict={
    x: mnist.test.images, y_: mnist.test.labels, keep_prob: 1.0}))
```

Using TensorFlow to iteratively improve our models

Even from the single example in the preceding section, we should be able to recognise what TensorFlow brings to the table. It offers a simple interface for the task of developing complex architectures and training methods, giving us easier access to the algorithms we've learnt about earlier in this book.

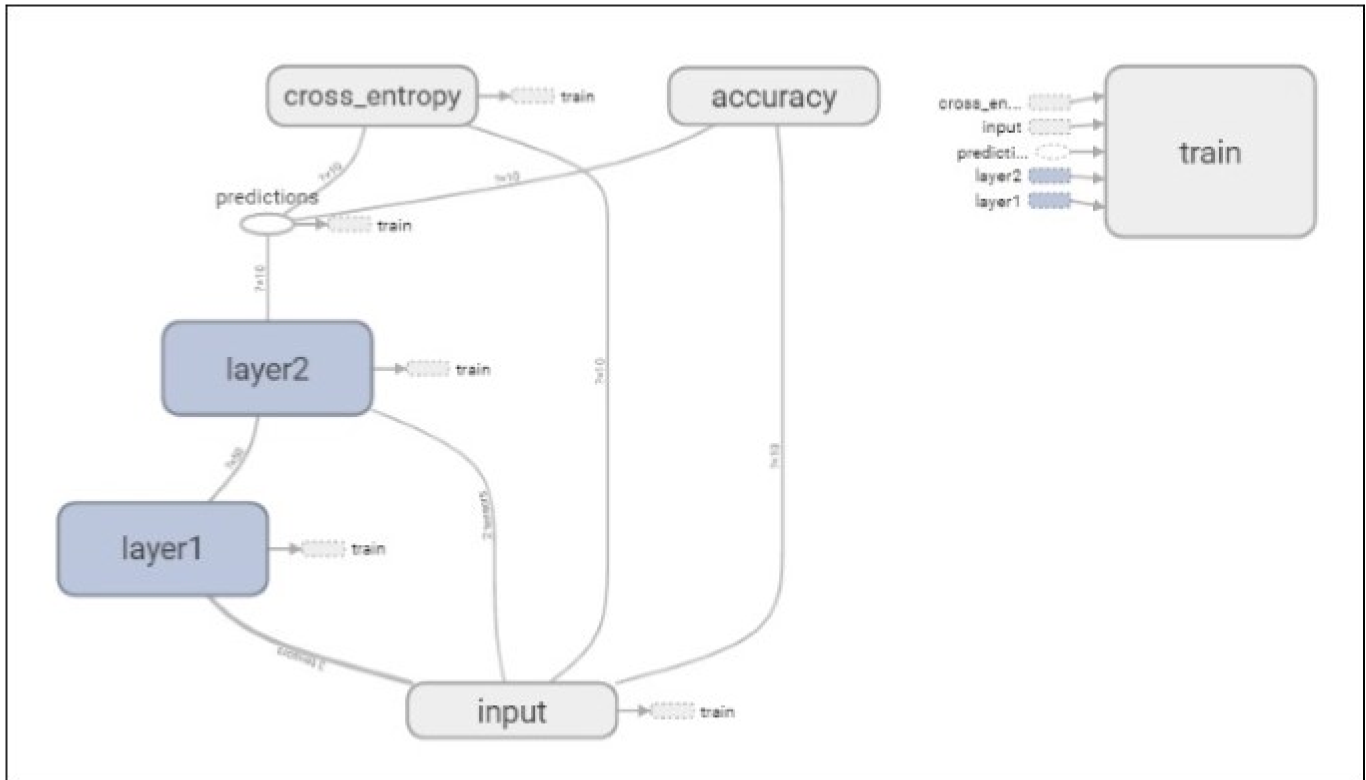
As we know, however, developing an initial model is only a small part of the model development process. We usually need to test and dissect our models repeatedly to improve their performance. However, this tends to be an area where our tools are less unified in a single library or technique, and the tests and monitoring solutions less consistent across models.

TensorFlow looks to solve the problem of how to get good insight into our models during iteration, in what it calls the execution phase of model development. During the execution phase, we can make use of tools provided by the TensorFlow team to explore and improve our models.

Perhaps the most important of these tools is TensorBoard, which provides an explorable, visual representation of the model we've built. TensorBoard provides several capabilities, including dashboards that show both basic model information (including performance measurements during each iteration for test and/or training).

In addition, TensorBoard dashboards provide lower-level information including plots of the range of values for weights, biases and activation values at every model layer; tremendously useful diagnostic information during iteration. The process of accessing this data is hassle-free and it is immediately useful.

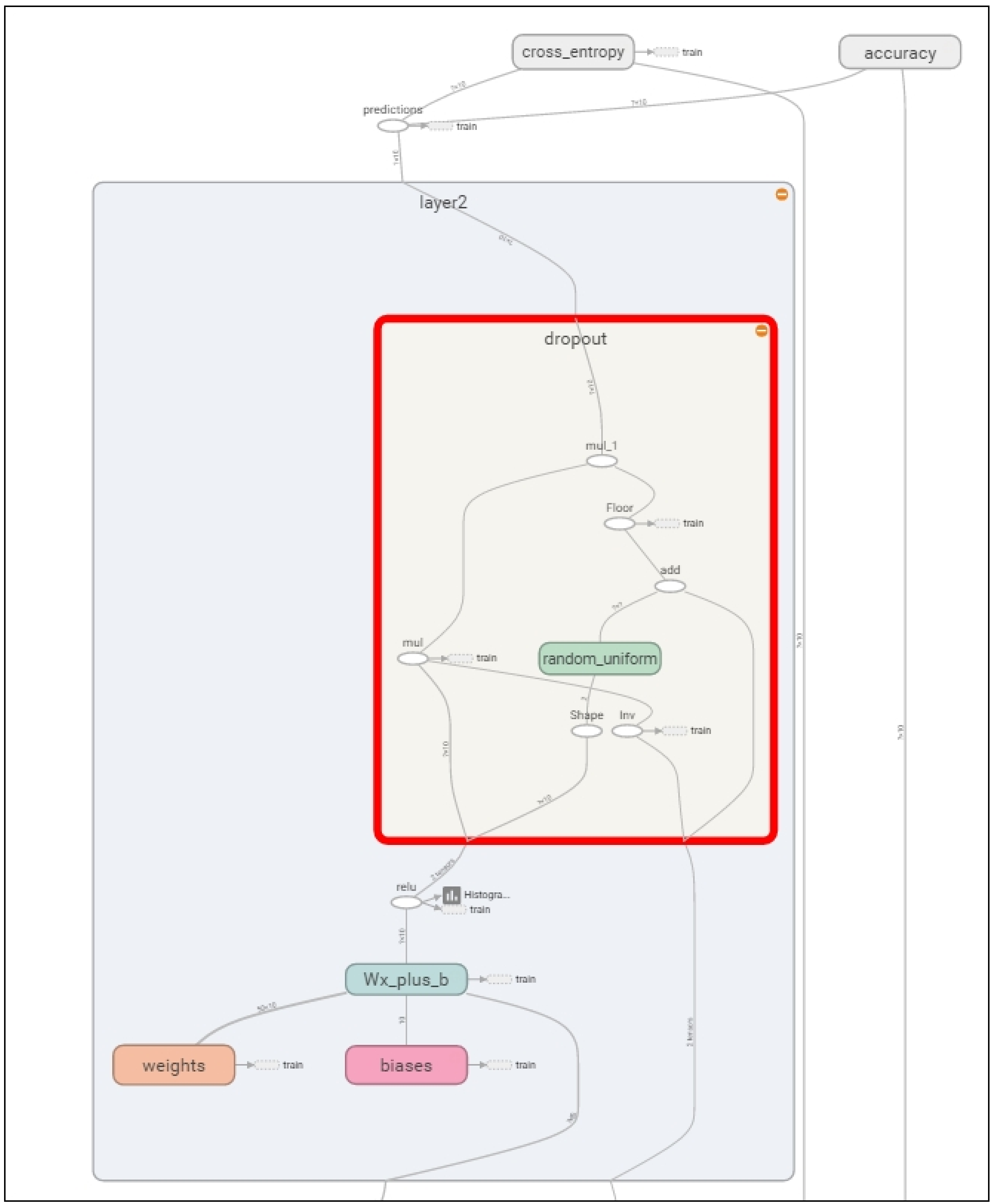
Further to this, TensorBoard provides a detailed graph of the tensor flow for a given model. The tensor is an n-dimensional array of data (in this case, of n-many features); it's what we tend to think of when we use the term *the input dataset*. The series of operations that is applied to a tensor is described as the tensor flow and in TensorFlow it's a fundamental concept, for a simple and compelling reason. When refining and debugging a machine learning model, what matters is having information about the model and its operations at even a low level.



TensorBoard graphs show the structure of a model in variable detail. From this initial view, it is possible to dig into each component of the model and into successive sub-elements. In this case, we are able to view the specific operations that take place within the dropout function of our second network layer. We can see what happens and identify what to tweak for our next iteration.

This level of transparency is unusual and can be very helpful when we want to tweak model components, especially when a model element or layer is underperforming (as we might see, for instance, from TensorBoard graphs showing layer metaparameter values or from network performance as a whole).

TensorBoards can be created from event logs and generated when TensorFlow is run. This makes the benefits of TensorBoards easily obtained during the course of everyday development using TensorFlow.



cross_entropy
train
accuracy
predictions
train
layer2
dropout
mul_1
Floor
train
add
mul
train
random_uniform
Shape
Inv
train
relu
Histogra...
train
Wx_plus_b
train
weights
train
biases
train

As of late April 2016, the DeepMind team joined the Google Brain team and a broad set of other researchers and developers in using TensorFlow. By making TensorFlow open source and freely available, Google is committing to continue supporting TensorFlow as a powerful tool for model development and refinement.

Knowing when to use these libraries

At one or two points in this chapter, we probably ran into the question of *Okay, so, why didn't you just teach us about this library to begin with?* It's fair to ask why we spent time digging around in Theano functions and other low-level information when this chapter presents perfectly good interfaces that make life easier.

Naturally, I advocate using the best tools available, especially for prototyping tasks where the value of the work is more in understanding the general ballpark you're in, or in identifying specific problem classes. It's worth recognising the three reasons for not presenting content earlier in this book using either of these libraries.

The first reason is that these tools will only get you so far. They can do a lot, agreed, so depending on the domain and the nature of that domain's problems, some data scientists may be able to rely on them for the majority of deep learning needs. Beyond a certain level of performance and problem complexity, of course, you need to understand what is needed to construct a model in Theano, create your own scoring function from scratch or leverage the other techniques described in this book.

Another part of the decision to focus on teaching lower-level implementation is about the developing maturity of the technologies involved. At this point, Lasagne and TensorFlow are definitely worth discussing and recommending to you. Prior to this, when the majority of the book was written, the risk around discussing the libraries in this chapter was greater. There are many projects based on Theano (some of the more prominent frameworks which weren't discussed in this chapter are **Keras**, **Blocks** and **Pylearn2**)

Even now, it's entirely possible that different libraries and tools will be the subject of discussion or the default working environment in a year or two years' time. This field moves extremely fast, largely due to the influence of key companies and research groups who have to keep building new tools as the old ones reach their useful limits… or it just becomes clear how to do things better.

The other reason to dig in at a lower level, honestly, is that this is an involved book. It sets theory alongside code and uses the code to teach the theory. Abstracting away how the algorithms work and simply discussing how to apply them to crack a particular example can be tempting. The tools discussed in this chapter enable practitioners to get very good scores on some problems without ever understanding the functions that are being called. My opinion is that this is not a very good way to train a data scientist.

If you're going to operate on subtle and difficult data problems, you need to be able to modify and define your own algorithm. You need to understand how to choose an appropriate solution. To do these things, you need the details provided in this book and even more very specific information that I haven't provided, due to the limitations of (page) space and time. At that point, you can apply deep learning algorithms flexibly and knowledgeably.

Similarly, it's important to recognise what these tools do well, or less well. At present, Lasagne fits very well within that use-case where a new model is being developed for benchmarking or early passes, where the priority should be on iteration speed and getting results.

TensorFlow, meanwhile, fits later into the development lifespan of a model. When the easy gains disappear and it's necessary to spend a lot of time debugging and improving a model, the relatively quick iterations of TensorFlow are a definite plus, but it's the diagnostic tools provided by TensorBoard that present an overwhelming value-add.

There is, therefore, a place for both libraries in your toolset. Depending on the nature of the problem at hand, these libraries and more will prove to be valuable assets.

Further reading

The Lasagne User Guide is thorough and worth reading. Find it at http://lasagne.readthedocs.io/en/latest/index.html.

Similarly, find the TensorFlow tutorials at https://www.tensorflow.org/versions/r0.9/get_started/index.html.

Summary

In this final chapter, we moved some distance from our previous discussions of algorithms, configuration and diagnosis to consider tools that improve our experience when implementing deep learning algorithms.

We discovered the advantages to using Lasagne, an interface to Theano designed to accelerate and simplify early prototyping of our models. Meanwhile, we examined TensorFlow, the library developed by Google to aid Deep Learning model adjustment and optimization. TensorFlow offers us a remarkable amount of visibility of model performance, at minimal effort, and makes the task of diagnosing and debugging a complex, deep model structure much less challenging.

Both tools have their own place in our processes, with each being appropriate for a particular set of problems.

Over the course of this book as a whole, we have walked through and reviewed a broad set of advanced machine learning techniques. We went from a position where we understood some fundamental algorithms and concepts, to having confident use of a very current, powerful and sought-after toolset.

Beyond the techniques, though, this book attempts to teach one further concept, one that's much harder to teach and to learn, but which underpins the best performance in machine learning.

The field of machine learning is moving very fast. This pace is visible in new and improved scores that are posted almost every week in academic journals or industry white papers. It's visible in how training examples like MNIST have moved quickly from being seen as meaningful challenges to being *toy problems*, the deep learning version of the Iris dataset. Meanwhile, the field moves on to the next big challenge; CIFAR-10, CIFAR-100.

At the same time, the field moves cyclically. Concepts introduced by academics like Yann LeCun in the 80's are in resurgence as computing architectures and resource growth make their use more viable over real data at scale. To use many of the most current techniques at their best limits, it's necessary to understand concepts that were defined decades ago, themselves defined on the back of other concepts defined still longer ago.

This book tries to balance these concerns. Understanding the cutting edge and the techniques that exist there is critical; understanding the concepts that'll define the new techniques or adjustments made in two or three years' time is equally important.

Most important of all, however, is that this book gives you an appreciation of how malleable these architectures and approaches can be. A concept consistently seen at the top end of data science practice is that the best solution to a specific problem is a problem-specific solution.

This is why top Kaggle contest winners perform extensive feature preparation and tweak their architectures. It's why TensorFlow was written to allow clear vision of granular properties of ones' architectures. Having the knowledge and the skills to tweak implementations or combine algorithms fluently is what it takes to have true mastery of machine learning techniques.

Through the many techniques and examples reviewed within this book, it is my hope that the ways of thinking about data problems and a confidence in manipulating and configuring these algorithms has been passed on to you as a practicing data scientist. The many recommended *Further reading* examples in this book are largely intended to further extend that knowledge and help you develop the skills taught in this book.

Beyond that, I wish you all the best of luck in your model building and configuration. I hope that you learn for yourself just how enjoyable and rewarding this field can be!

Appendix A. Chapter Code Requirements

This book's content leverages openly available data and code, including open source Python libraries and frameworks. While each chapter's example code is accompanied by a `README` file documenting all the libraries required to run the code provided in that chapter's accompanying scripts, the content of these files is collated here for your convenience.

It is recommended that you already have some libraries that are required for the earlier chapters when working with code from any later chapter. These requirements are identified using keywords. It is particularly important to set up the libraries mentioned in Chapter 1, *Unsupervised Machine Learning*, for any content provided later in the book. The requirements for every chapter are given in the following table:

Chapter Number	Requirements
1	• Python 3 (3.4 recommended) • sklearn (NumPy, SciPy) • matplotlib
2-4	• theano
5	• Semisup-learn
6	• **Natural Language Toolkit** (**NLTK**) • BeautifulSoup
7	• Twitter API account
8	• XGBoost
9	• Lasagne • TensorFlow

Part III. Module 3

Large Scale Machine Learning with Python

Learn to build powerful machine learning models quickly and deploy large-scale predictive applications

Chapter 1. First Steps to Scalability

Welcome to this book on scalable machine learning with Python.

In this chapter, we will discuss how to learn effectively from big data with Python and how it can be possible using your single machine or a cluster of other machines, which you can get, for instance, from **Amazon Web Services** (**AWS**) or the Google Cloud Platform.

In the book, we will be using Python's implementation of machine learning algorithms that are scalable. This means that they can work with a large amount of data and do not crash because of memory constraints. They also take a reasonable amount of time, which is something manageable for a data science prototype and also deployment in production. Chapters are organized around solutions (such as streaming data), algorithms (such as neural networks or ensemble of trees), and frameworks (such as Hadoop or Spark). We will also provide you with some basic reminders about the machine learning algorithms and explain how to make them scalable and suitable to problems with massive datasets.

Given such premises as a start, you'll need to learn the basics (so as to figure out the perspective under which this book has been written) and set up all your basic tools to start reading the chapters immediately.

In this chapter, we will introduce you to the following topics:

- What scalability actually means
- What bottlenecks you should pay attention to when dealing with data
- What kind of problems this book will help you solve
- How to use Python to analyze datasets at scale effectively
- How to set up your machine quickly to execute the examples presented in this book

Let's start this journey together around scalable solutions with Python!

Explaining scalability in detail

Even if the hype now is about big data, large datasets existed long before the term itself had been coined. Large collections of texts, DNA sequences, and vast amounts of data from radio telescopes have always represented a challenge for scientists and data analysts. As most machine learning algorithms have a computational complexity of $O(n^2)$ or even $O(n^3)$, where *n* is the number of training instances, the challenge from massive datasets has been previously faced by data scientists and analysts by resorting to data algorithms that could be more efficient. A machine learning algorithm is deemed scalable when it can work after an appropriate setup, in case of large datasets. A dataset can be large because of a large number of cases or variables, or because of both, but a scalable algorithm can deal with it in an efficient way as its running time increases almost linearly accordingly to the size of the problem. Therefore, it is just a matter of exchanging 1:1 more time (or more computational power) with more data. Instead, a machine learning algorithm doesn't scale if it's faced with large amounts of data; it simply stops working or operates with a running time that increases in a nonlinear way, for instance, exponentially, thus making learning unfeasible.

The introduction of cheap data storage, a large RAM, and multiprocessor CPU dramatically changed everything, increasing the ability of single laptops to analyze large amounts of data. Another big game changer arrived on the scene in the past years, shifting the attention from single powerful machines to clusters of commodity computers (cheaper, easily available machines). This big change has been the introduction of **MapReduce** and the open source framework Apache Hadoop with its **Hadoop Distributed File System** (**HDFS**) and, in general, of parallel computation on networks of computers.

In order to figure out how both of these changes deeply and positively affected your capabilities of solving your large scale problems, we should first start from what actually prevented you (and still prevents, depending on how massive is your problem) from analyzing large datasets.

No matter what your problem is, you will eventually find out that you cannot analyze your data because of any of these limits:

- Computing affecting the time taken to execute the analysis
- I/O affecting how much of your data you can take from storage to memory in a time unit
- Memory affecting how much large data you can process at a time

Your computer has limitations that will determine if you can learn from your data and how long it will take before you hit a wall. Computing limitations occur in many intensive calculations, I/O problems will bottleneck your prompt access to data, and finally memory limitations can constraint you to take on only a part of your data, thus limiting the kind of matrix computations that you may have access to or the precision or even exactness of your estimations.

Each of these hardware limitations will also affect you differently in severity with regard to the data you are analyzing:

- Tall data, which is characterized by a large number of cases
- Wide data, which is characterized by a large number of features
- Tall and wide data, which has a large number of both cases and features
- Sparse data, which is characterized by a large number of zero entries or entries that could be transformed into zeros (that is, the data matrix may be tall and/or wide but informative, but not all the matrix entries have informative value)

Finally, it comes down to the algorithm that you are going to use in order to learn from the data. Each algorithm has its own characteristics, being able to map data using a solution differently affected by bias or variance. Therefore, with respect to your problem that, so far, you solved by machine learning, you considered, based on experience or empirical tests, that certain algorithms may work better than others did. With large scale problems, you have to add other and different considerations when deciding on the algorithm:

- How complex your algorithm is; that is, if the number of rows and columns in your data affects the number of computations in a linear or nonlinear way. Most machine learning solutions are based on algorithms of quadratic or cubic complexity, thus strongly limiting their applicability to big data.
- How many parameters your model has; here, it's not just a problem of variance of the estimates (overfitting), but of the time it may take to compute them all.

- If the optimization processes are parallelizable; that is, can you easily split the computations across multiple nodes or CPU cores, or do you have to rely on a single, sequential, optimization process?
- Should the algorithm learn from all the data at once or can you use single examples or small batches of data instead?

If you cross-evaluate hardware limitations with data characteristics and these kind of algorithms, you'll get a host of possible problematic combinations that can prevent you from getting results from large scale analysis. From a practical point of view, all the problematic combinations can be solved by three approaches:

- Scaling up, that is, improving performances on a single machine by software or hardware modifications (more memory, faster CPU, faster storage disk, and using GPUs)
- Scaling out, that is, distributing the computation (and the performances) across multiple machines leveraging outside resources, namely other storage disks and other CPUs (or GPUs)
- Scaling up and out, that is, taking the best of the scaling up and out solutions together

Making large scale examples

Some motivating examples may make things clearer and more memorable for you. Let's take two simple examples:

- Being able to predict the **click-through rate** (**CTR**) can help you earn quite a lot these days when Internet advertising is so widespread, diffused, and eating large shares of traditional media communication
- Being able to propose the right information to your customers, when they are searching the products and services offered by your site, could really enhance your chances to sell if you can guess what to put at the top of their results

In both cases, we have quite large datasets as they are produced by users' interactions on the Internet.

Depending on the business that we have in mind (we can imagine some big players here), we are clearly talking of millions of data points per day in both our examples. In the advertising case, data is certainly tall, being a continuous stream of information as the most recent data, more representative of markets and consumers, replaces the older one. In the search engine case, data is wide, being enriched by the feature provided by the results you offered to your customers: for instance, if you are in the travels business, you will have quite a lot of features about hotels, locations, and services offered.

Clearly, scalability is an issue for both these problems:

- You have to learn from data that is growing every day and you have to learn fast because as you are learning, new data keeps arriving. Yet, you have to deal with data that clearly cannot fit in memory because the matrix is too tall or too large.
- You frequently need to update your machine learning model in order to accommodate new data. You need an algorithm that can process the information in a timely manner. O(n2) or O(n3) complexities could be impossible for you to handle because of the data quantity; you need some algorithm that can work with lower complexity (such as O(n)) or by dividing the data so that *n* will be much, much smaller.

- You have to be able to predict fast because the predictions have to be delivered only to new customers. Again, the complexity of your algorithm does matter.

The scalability problem can be solved in one or multiple ways:

- Scaling up by reducing the dimensionality of the problem; for instance, in the case of the search engine, by effectively selecting the relevant features to be used
- Scaling up using the right algorithm; for instance, in the case of advertising data, there are appropriate algorithms to learn effectively from streams
- Scaling out the learning process by leveraging multiple machines
- Scaling up the deployment process using multiprocessing and vectorization on a single server effectively

In this book, we will point out for you what kind of practical problems can be solved by each one of the solutions or algorithms proposed. It will become automatic for you to connect a particular constraint in time and execution (CPU, memory, or I/O) to the most suitable solution among the ones that we propose.

Introducing Python

As our treatise will depend on Python—our open source language of choice for this book—we have to stop for a brief moment and present the language before clarifying how Python can easily help you scale up and out with your massive data problem.

Created in 1991 as a general-purpose, interpreted, object-oriented language, Python has slowly and steadily conquered the scientific community and grown into a mature ecosystem of specialized packages for data processing and analysis. It allows you to have uncountable and fast experimentations, easy theory developments, and prompt deployments of scientific applications.

As a machine learning practitioner, you will find using Python interesting for various reasons:

- It offers a large, mature system of packages for data analysis and machine learning. It guarantees that you will get all that you may need in the course of a data analysis, and sometimes even more.
- It is very versatile. No matter what your programming background or style is (object-oriented or procedural), you will enjoy programming with Python.
- If you don't know it yet but you know other languages such as C/C++ or Java well, then it is very simple to learn and use. After you grasp the basics, there's no other better way to learn more than by immediately starting with the coding.
- It is cross-platform; your solutions will work perfectly and smoothly on Windows, Linux, and macOS systems. You won't have to worry about portability.
- Although interpreted, it is undoubtedly fast compared to other mainstream data analysis languages such as R and MATLAB (though it is not comparable to C, Java, and the newly emerged Julia language).
- It can work with in-memory big data because of its minimal memory footprint and excellent memory management. The memory garbage collector will often save the day when you load, transform, dice, slice, save, or discard data using the various iterations and reiterations of data wrangling.

Tip

If you are not already an expert (and actually we require some basic knowledge of Python in order to be able to make the most out of this book), you can read everything about the language and find the basic installations files directly from the Python foundations at https://www.python.org/.

Scale up with Python

Python is an interpreted language; it runs the reading of your script from memory and executes it during runtime, thus accessing the necessary resources (files, objects in memory, and so on). Apart from being interpreted, another important aspect to take into consideration when using Python for data analysis and machine learning is that Python is single-threaded. Being single-threaded means that any Python program is executed sequentially from the start to the end of the script and that Python cannot take advantage of the extra processing power offered by the multiple threads and processors likely present in your computer (most computers nowadays are multicore).

Given such a situation, scaling up using Python can be achieved by different strategies:

- Compiling Python scripts in order to achieve more speed of execution. Though easily possible using, for instance, **PyPy**—a **Just-in-Time** (**JIT**) compiler that can be found at http://pypy.org/, we actually didn't resort to such a solution in our book because it requires writing algorithms in Python from scratch.
- Using Python as a wrapping language; thus putting together the operations executed by Python with the execution of external libraries and programs, some capable of multicore processing. In our book, you will find many examples of this when we call specialized libraries such as the **Library for Support Vector Machines** (**LIBSVM**) or programs such as **Vowpal Wabbit** (**VW**), **XGBoost**, or **H2O** in order to execute machine learning activities.
- Effectively using vectorization techniques, that is, special libraries for matrix computations. This can be achieved using **NumPy** or **pandas**, both using computations from GPUs. GPUs are just like multicore CPUs, each one with their own memory and ability to process calculations in parallel (you can figure out that they have multiple tiny cores). Especially when working with neural networks, vectorization techniques based on GPUs can speed up computations incredibly. However, GPUs have their own limitations; first of all, their available memory has a certain I/O in passing your data to their memory and getting the results back to your CPU, and they require parallel programming via a special API, such as **CUDA** for NVIDIA-manufactured GPUs (so you have to install the appropriate drivers and programs).
- Reducing a large problem into chunks and solving each chunk one at a time in-memory (divide and conquer algorithms). This leads to the partitioning or subsampling of data from memory or disk and managing approximate solutions of your machine learning problem, which is quite effective. It is important to notice that both partitioning and subsampling can operate for cases and features (and both). If the original data is kept on a disk storage, I/O constraints will become quite determinant of the resulting performances.
- Effectively leveraging both multiprocessing and multithreading, depending on the learning algorithm that you will be using. Some algorithms will naturally be able to split their operations into parallel ones. In such cases, the only constraint will be your CPU's and your memory (as your data will have to be replicated for every parallel worker that you will be using). Some other

algorithms will instead take advantage of multithreading, thus managing more operations at the same time on the same memory blocks.

Scale out with Python

Scaling out solutions simply involve connecting together multiple machines into a cluster. As you connect the machines (scaling out), you can also scale up each one of them using configurations that are more powerful (thus augmenting CPU, memory, and I/O), applying the techniques we mentioned in the previous paragraph and enhancing their performances.

By connecting multiple machines, you can leverage their computational power in a parallel fashion. Your data will be distributed across multiple storage disks/memory, limiting I/O transfers by having each machine work only on its available data (that is, its own storage disk or RAM memory).

In our book, this translates into using outside resources effectively by means of the following:

- The H2O framework
- The Hadoop framework and its components, such as HDFS, MapReduce, and **Yet Another Resource Negotiator** (**YARN**)
- The Spark framework on top of Hadoop

Each of these frameworks will be controlled by Python (for instance, Spark by its Python interface named pySpark).

Python for large scale machine learning

Given the availability of many useful packages for machine learning and the fact that it is a programming language quite popular among data scientists, Python is our language of choice for all the code presented in this book.

In this book, when necessary, we will provide further instructions in order to install any further necessary library or tool. Here, we will instead start installing the basics, that is, the Python language and the most frequently used packages for computations and machine learning.

Choosing between Python 2 and Python 3

Before starting, it is important to know that there are two main branches of Python: versions 2 and 3. As many core functionalities have changed, scripts built for one version are sometimes incompatible with the other one (they won't work without raising errors and warnings). Although the third version is the newest, the older one is still the most used version in the scientific area and the default version for many operative systems (mainly for compatibility in upgrades). When version 3 was released (in 2008), most scientific packages weren't ready so the scientific community stuck with the previous version. Fortunately, since then, almost all packages have been updated leaving just a few (see http://py3readiness.org for a compatibility overview) as orphans of Python 3 compatibility.

In spite of the recent growth in popularity of Python 3 (which, we shouldn't forget, is the future of Python), Python 2 is still widely used among data scientists and data analysts. Moreover, for a long time Python 2 has been the default Python installation (for instance, on Ubuntu), so it is the most likely version that most of the readers should have ready at hand. For all these reasons, we will adopt Python 2 for this book. It is not merely *love for the old technologies*, it is just a practical choice in order to make *Large Scale Machine Learning with Python* accessible to the largest audience:

- The Python 2 code will immediately address the existing audience of data experts.
- Python 3 users will find it very easy to convert our scripts in order to work under their favored Python version because the code we wrote is easily convertible and we will provide a Python 3 version of all our scripts and notebooks, freely downloadable from the Packt website.

Tip

In case you need to understand the differences between Python 2 and Python 3 in depth, we suggest reading this web page about writing Python 2-3 compatible code:

http://python-future.org/compatible_idioms.html

From **Python-Future**, you may also find reading about how to convert Python 2 code to Python 3 useful:

http://python-future.org/automatic_conversion.html

Package upgrades

More often than not, you will find yourself in a situation where you have to upgrade a package because the new version is either required by a dependency or has additional features that you would like to use. To do so, first check the version of the library that you have installed by glancing at the `__version__` attribute, as shown in the following example using the NumPy package:

```
>>> import numpy
>>> numpy.__version__ # 2 underscores before and after
'1.9.0'
```

Now, if you want to update it to a newer release, say precisely the 1.9.2 version, you can run the following command from the command line:

```
$ pip install -U numpy==1.9.2
```

Alternatively (but we do not recommend it unless it proves necessary), you can also use the following command:

```
$ easy_install --upgrade numpy==1.9.2
```

Finally, if you're just interested in upgrading it to the latest available version, simply run the following command:

```
$ pip install -U numpy
```

You can also run the `easy_install` alternative:

```
$ easy_install --upgrade numpy
```

Scientific distributions

As you've read so far, creating a working environment is a time-consuming operation for a data scientist. You first need to install Python and then, one by one, you can install all the libraries that you will need. (Sometimes, the installation procedures may not go as smoothly as you'd hoped for earlier.)

If you want to save time and effort and want to ensure that you have a fully working Python environment that is ready to use, you can just download, install, and use the scientific Python distribution. Apart from Python, they also include a variety of preinstalled packages, and sometimes they even have additional tools and an IDE setup for your usage. A few of them are very well-known among data scientists, and in the sections that follow, you will find some of the key features for two of these packages that we found most useful and practical.

To immediately focus on the contents of the book, we suggest that you first promptly download and install a scientific distribution, such as **Anaconda** (which is the most complete one around, in our opinion), and decide to fully uninstall the distribution and set up Python alone after practicing the examples in the book, which can be accompanied by just the packages you need for your projects.

Again, if possible, download and install the version containing Python 3.

The first package that we would recommend you to try is Anaconda (https://www.continuum.io/downloads), which is a Python distribution offered by Continuum Analytics that includes nearly 200 packages, including NumPy, SciPy, pandas, IPython, matplotlib, Scikit-learn, and StatsModels. It's a cross-platform distribution that can be installed on machines with other existing Python distributions and versions, and its base version is free. Additional add-ons that contain advanced features are charged separately. Anaconda introduces **conda**, a binary package manager, as a command-line tool to manage your package installations. As stated on its website, Anaconda's goal is to provide enterprise-ready Python distribution for large-scale processing, predictive analytics and scientific computing. As for Python version 2.7, we recommend the Anaconda distribution 4.0.0. (In order to have a look at the packages installed with Anaconda, you can have a look at the list at https://docs.continuum.io/anaconda/pkg-docs.)

As a second suggestion, if you are working on Windows and you desire a portable distribution, **WinPython** (http://winpython.sourceforge.net/) could be a quite interesting alternative (sorry, no Linux or MacOS versions). WinPython is also a free, open source Python distribution maintained by the community. It is also designed with scientists in mind, and it includes many essential packages such as NumPy, SciPy, matplotlib, and IPython (basically the same as Anaconda's). It also includes **Spyder** as an IDE, which can be helpful if you have experience using the MATLAB language and interface. Its crucial advantage is that it is portable (you can put it in any directory or even in a USB flash drive), so you can have different versions present on your computer, move a version from a Windows computer to another, and you can easily replace an older version with a newer one just by replacing its directory. When you run WinPython or its shell, it will automatically set all the environment variables necessary to run Python as if it were regularly installed and registered on your system.

Tip

At the time of writing, Python 2.7 was the most recent distribution prepared on October 2015 with the release 2.7.10; since then, WinPython has published only updates of the Python 3 version of the distribution. After installing the distribution on your system, you may need to update some of the key packages necessary for the examples present in this book.

Introducing Jupyter/IPython

IPython was initiated in 2001 as a free project by Fernando Perez, addressing a lack in the Python stack for scientific investigations using a user-programming interface that could incorporate the scientific approach (mainly experimenting and interactively discovering) in the process of software development.

A scientific approach implies the fast experimentation of different hypotheses in a reproducible fashion (as does the data exploration and analysis task in data science), and when using IPython, you will be able to implement an explorative, iterative, and trial-and-error research strategy more naturally during your code writing.

Recently, a large part of the IPython project has moved to a new one called **Jupyter**. This new project extends the potential usability of the original IPython interface to a wide range of programming languages. (For a complete list, visit https://github.com/ipython/ipython/wiki/IPython-kernels-for-other-languages.)

Thanks to the powerful idea of kernels, programs that run the user's code are communicated by the frontend interface and provide feedback on the results of the executed code to the interface itself; you can use the same interface and interactive programming style, no matter what language you are developing in.

Jupyter (IPython is the zero kernel, the original starting one) can be simply described as a tool for interactive tasks operable by a console or web-based notebook, which offers special commands that help developers better understand and build the code that is being currently written.

Contrary to an IDE, which is built around the idea of writing a script, running it afterward and evaluating its results, Jupyter lets you write your code in chunks named **cells**, run each of them sequentially, and evaluate the results of each one separately, examining both textual and graphic outputs. Besides graphical integration, it provides you with further help, thanks to customizable commands, a rich history (in the JSON format), and computational parallelism for an enhanced performance when dealing with heavy numeric computations.

Such an approach is also particularly fruitful for the tasks involving developing code based on data as it automatically accomplishes the often neglected duty of documenting and illustrating how data analysis has been done, its premises and assumptions, and its intermediate and final results. If a part of your job is to also present your work and persuade internal or external stakeholders to the project, Jupyter can really do the magic of storytelling for you with few additional efforts. There are many examples on https://github.com/ipython/ipython/wiki/A-gallery-of-interesting-IPython-Notebooks, some of which you may find inspiring for your work as we did.

Actually, we have to confess that keeping a clean, up-to-date Jupyter Notebook has saved us uncountable times when meetings with managers/stakeholders have suddenly popped up, requiring us to hastily present the state of our work.

In short, Jupyter offers you the following features:

- Seeing intermediate (debugging) results for each step of the analysis
- Running only some sections (or cells) of the code
- Storing intermediate results in the JSON format and having the ability to do version control on them
- Presenting your work (this will be a combination of text, code, and images), sharing it via the Jupyter Notebook Viewer service (http://nbviewer.jupyter.org/), and easily exporting it to HTML, PDF, or even slideshows

Jupyter is our favored choice throughout this book, and it is used to clearly and effectively illustrate storytelling operations with scripts and data and their consequent results.

Though we strongly recommend using Jupyter, if you are using an REPL or IDE, you can use the same instructions and expect identical results (except for print formats and extensions of the returned results).

If you do not have Jupyter installed on your system, you can promptly set it up using the following command:

```
$ pip install jupyter
```

Tip

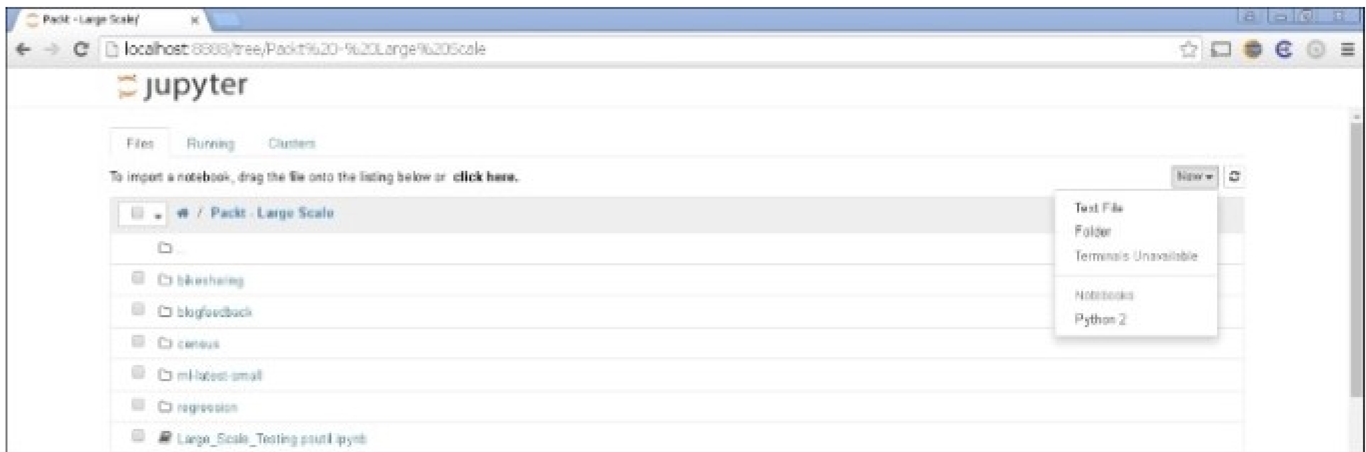
You can find complete instructions about the Jupyter installation (covering different operating systems) at http://jupyter.readthedocs.io/en/latest/install.html.

If you already have Jupyter installed, it should be upgraded to at least version 4.1.

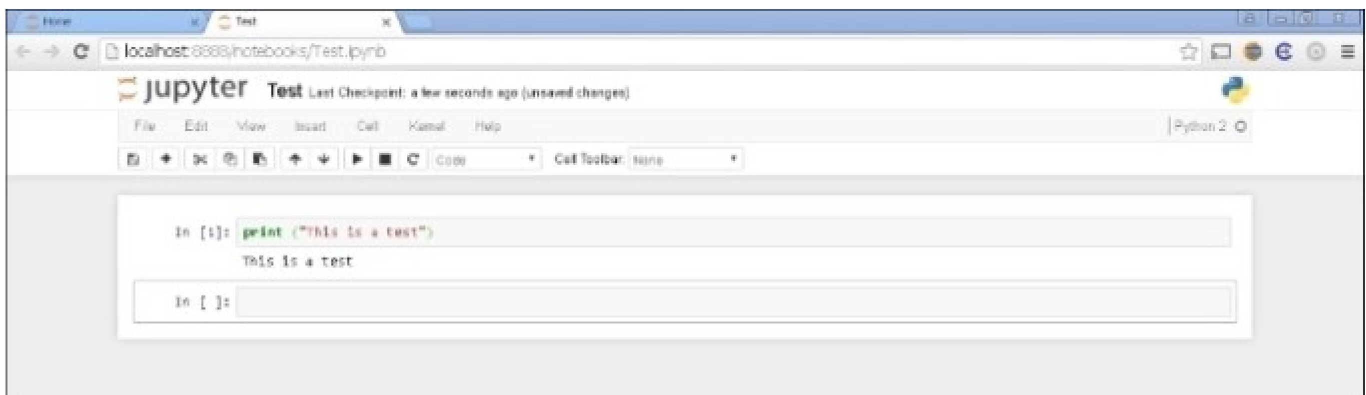
After installation, you can immediately start using Jupyter, calling it from the command line:

```
$ jupyter notebook
```

Once the Jupyter instance has opened in the browser, click on the **New** button, and in the Notebooks section, choose **Python 2** (other kernels may be present in the section, depending on what you installed):



At this point, your new empty notebook will look like the following screenshot and you can start entering the commands in the cells:



For instance, you may start typing the following in the cell:

```
In: print ("This is a test")
```

After writing in cells, you just press the play button (below the **Cell** tab) to run it and obtain an output. Then, another cell will appear for your input. As you are writing in a cell, if you press the plus button on the above menu bar, you will get a new cell, and you can move from a cell to another using the arrows on the menu.

Most of the other functions are quite intuitive and we invite you to try them. In order to know better how Jupyter works, you may use a quick-start guide such as http://jupyter-notebook-beginner-guide.readthedocs.io/en/latest/ or you can get a book specialized in Jupyter functionalities.

Note

For a complete treatise of the full range of Jupyter functionalities when running the IPython kernel, refer to the following two Packt Publishing books:

- *IPython Interactive Computing and Visualization Cookbook* by *Cyrille Rossant*, *Packt Publishing*, September 25, 2014
- *Learning IPython for Interactive Computing and Data Visualization* by *Cyrille Rossant*, *Packt Publishing*, April 25, 2013

For our illustrative purposes, just consider that every Jupyter block of instructions has a numbered input statement and an output one, so you will find the code presented in this book structured in to two blocks—at least when the output is not trivial at all—otherwise, just expect only the input part:

```
In:  <the code you have to enter>
Out: <the output you should get>
```

As a rule, you just have to type the code after **In:** in your cells and run it. You can then compare your output with the output that we provide using **Out:** followed by the output that we actually obtained on our computers when we tested the code.

Python packages

The packages that we are going to introduce in the present paragraph will be frequently used in the book. If you are not using a scientific distribution, we offer you a walkthrough on what versions you should decide on and how to install them quickly and successfully.

NumPy

NumPy, which is Travis Oliphant's creation, is at the core of every analytical solution in the Python language. It provides the user with multidimensional arrays along with a large set of functions to operate multiple mathematical operations on these arrays. Arrays are blocks of data arranged along multiple dimensions, which implement mathematical vectors and matrices. Arrays are useful not just to store data, but also for fast matrix operations (vectorization), which are indispensable when you wish to solve ad hoc data science problems.

- Website: http://www.numpy.org/
- Version at the time of writing: 1.11.1
- Suggested install command:

```
$ pip install numpy
```

Tip

As a convention that is largely adopted by the Python community, when importing NumPy, it is suggested that you alias it as `np`:

```
import numpy as np
```

SciPy

An original project by Travis Oliphant, Pearu Peterson, and Eric Jones, SciPy completes NumPy's functionalities, offering a larger variety of scientific algorithms for linear algebra, sparse matrices, signal and image processing, optimization, fast Fourier transformation, and much more.

- Website: http://www.scipy.org/
- Version at the time of writing: 0.17.1
- Suggested install command:

```
$ pip install scipy
```

Pandas

Pandas deals with everything that NumPy and SciPy cannot do. In particular, thanks to its specific object data structures, DataFrames, and Series, it allows the handling of complex tables of data of different types (something that NumPy's arrays cannot) and time series. Thanks to Wes McKinney's creation, you will be able to easily and smoothly load data from a variety of sources, and then slice, dice, handle missing elements, add, rename, aggregate, reshape, and finally visualize it at your will.

- Website: http://pandas.pydata.org/
- Version at the time of writing: 0.18.0
- Suggested install command:

```
$ pip install pandas
```

Tip

Conventionally, pandas is imported as `pd`:

```
import pandas as pd
```

Scikit-learn

Started as part of SciKits (SciPy Toolkits), Scikit-learn is the core of data science operations in Python. It offers all that you may need in terms of data preprocessing, supervised and unsupervised learning, model selection, validation, and error metrics. Expect us to talk at length about this package throughout the book.

Scikit-learn started in 2007 as a Google Summer of Code project by David Cournapeau. Since 2013, it has been taken over by the researchers at Inria (French Institute for Research in Computer Science and Automation).

Scikit-learn offers modules for data processing (`sklearn.preprocessing` and `sklearn.feature_extraction`), model selection and validation (`sklearn.cross_validation`, `sklearn.grid_search`, and `sklearn.metrics`), and a complete set of methods (`sklearn.linear_model`) in which the target value, being a number or probability, is expected to be a linear combination of the input variables.

- Website: http://scikit-learn.org/stable/
- Version at the time of writing: 0.17.1
- Suggested install command:

```
$ pip install scikit-learn
```

Tip

Note that the imported module is named `sklearn`.

The matplotlib package

Originally developed by John Hunter, matplotlib is the library containing all the building blocks to create quality plots from arrays and visualize them interactively.

You can find all the MATLAB-like plotting frameworks inside the PyLab module.

- Website: http://matplotlib.org/
- Version at the time of writing: 1.5.1
- Suggested install command:

```
$ pip install matplotlib
```

You can simply import just what you need for your visualization purposes:

```
import matplotlib as mpl
from matplotlib import pyplot as plt
```

Gensim

Gensim, programmed by Radim Řehůřek, is an open source package suitable to analyze large textual collections by the usage of parallel distributable online algorithms. Among advanced functionalities, it implements **Latent Semantic Analysis** (**LSA**), topic modeling by **Latent Dirichlet Allocation** (**LDA**), and Google's **word2vec**, a powerful algorithm to transform texts into vector features to be used in supervised and unsupervised machine learning.

- Website: http://radimrehurek.com/gensim/
- Version at the time of writing: 0.13.1
- Suggested install command:

```
$ pip install gensim
```

H2O

H2O is an open source framework for big data analysis created by the start-up **H2O.ai** (previously named as 0xdata). It is usable by R, Python, Scala, and Java programming languages. H2O easily allows using a standalone machine (leveraging multiprocessing) or Hadoop cluster (for example, a cluster in an AWS environment), thus helping you scale up and out.

- Website: http://www.h2o.ai
- Version at the time of writing: 3.8.3.3

In order to install the package, you first have to download and install Java on your system, (You need to have **Java Development Kit** (**JDK**) 1.8 installed as H2O is Java-based.) then you can refer to the online instructions provided at http://www.h2o.ai/download/h2o/python.

We can overview all the installation steps together in the following lines.

You can install both H2O and its Python API, as we have been using in our book, by the following instructions:

```
$ pip install -U requests
$ pip install -U tabulate
$ pip install -U future
$ pip install -U six
```

These steps will install the required packages, and then we can install the framework, taking care to remove any previous installation:

```
$ pip uninstall h2o
$ pip install h2o
```

In order to have installed the same version as we have in our book, you can change the last `pip install` command with the following:

```
$ pip install http://h2o-release.s3.amazonaws.com/h2o/rel-turin/3/Python/h2o-3.8.3.3-py2.py3-none-any.whl
```

If you run into problems, please visit the H2O Google groups page, where you can get help with your problems:

https://groups.google.com/forum/#!forum/h2ostream

XGBoost

XGBoost is a scalable, portable, and distributed gradient boosting library (a tree ensemble machine learning algorithm). It is available for Python, R, Java, Scala, Julia, and C++ and it can work on a single machine (leveraging multithreading), both in Hadoop and Spark clusters.

- Website: https://xgboost.readthedocs.io/en/latest/
- Version at the time of writing: 0.4

Detailed instructions to install XGBoost on your system can be found at https://github.com/dmlc/xgboost/blob/master/doc/build.md.

The installation of XGBoost on both Linux and Mac OS is quite straightforward, whereas it is a little bit trickier for Windows users. For this reason, we provide specific installations steps to have XGBoost working on Windows:

1. First of all, download and install **Git for Windows** (https://git-for-windows.github.io/).
2. Then you need a **Minimalist GNU for Windows** (**MinGW**) compiler present on your system. You can download it from http://www.mingw.org/ according to the characteristics of your system.
3. From the command line, execute the following:

   ```
   $ git clone --recursive https://github.com/dmlc/xgboost
   $ cd xgboost
   $ git submodule init
   $ git submodule update
   ```

4. Then, from the command line, copy the configuration for 64-bit systems to be the default one:

   ```
   $ copy make\mingw64.mk config.mk
   ```

   Alternatively, you can copy the plain 32-bit version:

   ```
   $ copy make\mingw.mk config.mk
   ```

5. After copying the configuration file, you can run the compiler, setting it to use four threads in order to speed up the compiling procedure:

   ```
   $ make -j4
   ```

6. Finally, if the compiler completed its work without errors, you can install the package in your Python by executing the following commands:

   ```
   $ cd python-package
   $ python setup.py install
   ```

Theano

Theano is a Python library that allows you to define, optimize, and evaluate mathematical expressions involving multidimensional arrays efficiently. Basically, it provides you with all the building blocks that you need to create deep neural networks.

- Website: http://deeplearning.net/software/theano/
- Release at the time of writing: 0.8.2

The installation of Theano should be straightforward as it is now a package on PyPI:

```
$ pip install Theano
```

If you want the most updated version of the package, you can get them with GitHub cloning:

```
$ git clone git://github.com/Theano/Theano.git
```

Then you can proceed with the direct Python installation:

```
$ cd Theano
$ python setup.py install
```

To test your installation, you can run the following from the shell/CMD and verify the reports:

```
$ pip install nose
$ pip install nose-parameterized
$ nosetests theano
```

If you are working on a Windows OS and the previous instructions don't work, you can try these steps:

1. Install TDM-GCC x64 (http://tdm-gcc.tdragon.net/).
2. Open the Anaconda command prompt and execute the following:

   ```
   $ conda update conda
   $ conda update -all
   $ conda install mingw libpython
   $ pip install git+git://github.com/Theano/Theano.git
   ```

Tip

Theano needs libpython, which isn't compatible yet with version 3.5, so if your Windows installation is not working, that could be the likely cause.

In addition, Theano's website provides some information to Windows users that could support you when everything else fails:

http://deeplearning.net/software/theano/install_windows.html

An important requirement for Theano to scale out on GPUs is to install NVIDIA **CUDA** drivers and SDK for code generation and execution on GPU. If you do not know too much about the CUDA Toolkit, you can actually start from this web page in order to understand more about the technology being used:

https://developer.nvidia.com/cuda-toolkit

Therefore, if your computer owns an NVIDIA GPU, you can find all the necessary instructions in order to install CUDA using this tutorial page from NVIDIA itself:

http://docs.nvidia.com/cuda/cuda-quick-start-guide/index.html#axzz4A8augxYy

TensorFlow

Just like Theano, **TensorFlow** is another open source software library for numerical computation using data flow graphs instead of just arrays. Nodes in such a graph represent mathematical operations, whereas the graph edges represent the multidimensional data arrays (the so-called tensors) moved between the nodes. Originally, Google researchers, being part of the Google Brain Team, developed TensorFlow and recently they made it open source for the public.

- Website: https://github.com/tensorflow/tensorflow
- Release at the time of writing: 0.8.0

For the installation of TensorFlow on your computer, follow the instructions found at the following link:

https://github.com/tensorflow/tensorflow/blob/master/tensorflow/g3doc/get_started/os_setup.md

Windows support is not present at the moment but it is in the current roadmap:

https://github.com/tensorflow/tensorflow/blob/master/tensorflow/g3doc/resources/roadmap.md

For Windows users, a good compromise could be to run the package on a Linux-based virtual machine or Docker machine. (The preceding OS set-up page offers directions to do so.)

The sknn library

The **sknn** library (for extensions, **scikit-neuralnetwork**) is a wrapper for Pylearn2, helping you to implement deep neural networks without requiring you to become an expert on Theano. As a bonus, the library is compatible with the Scikit-learn API.

- Website: https://scikit-neuralnetwork.readthedocs.io/en/latest/
- Release at the time of publication: 0.7
- To install the library, just use the following command:

```
$ pip install scikit-neuralnetwork
```

Optionally, if you want to take advantage of the most advanced features such as convolution, pooling, or upscaling, you have to complete the installation as follows:

```
$ pip install -r https://raw.githubusercontent.com/aigamedev/
scikit-neuralnetwork/master/requirements.txt
```

After installation, you also have to execute the following:

```
$ git clone https://github.com/aigamedev/scikit-neuralnetwork.git
$ cd scikit-neuralnetwork
$ python setup.py develop
```

As seen for XGBoost, this will make the `sknn` package available in your Python installation.

Theanets

The **theanets** package is a deep learning and neural network toolkit written in Python and uses Theano to accelerate computations. Just as with sknn, it tries to make it easier to interface with Theano functionalities in order to create deep learning models.

- Website: https://github.com/lmjohns3/theanets
- Version at the time of writing: 0.7.3
- Suggested installation procedure:

```
$ pip install theanets
```

You can also download the current version from GitHub and install the package directly in Python:

```
$ git clone https://github.com/lmjohns3/theanets
$ cd theanets
$ python setup.py develop
```

Keras

Keras is a minimalist, highly modular neural networks library written in Python and capable of running on top of either TensorFlow or Theano.

- Website: http://keras.io/
- Version at the time of writing: 1.0.5
- Suggested installation from PyPI:

```
$ pip install keras
```

You can also install the latest available version (advisable as the package is in continuous development) using the following command:

```
$ pip install git+git://github.com/fchollet/keras.git
```

Other useful packages to install on your system

Concluding this long tour of the many packages that you will see in action among the pages of this book, we close with three simple, yet quite useful, packages, that need little presentation but need to be installed on your system: **memory profiler**, **climate**, and **NeuroLab**.

Memory profiler is a package monitoring memory usage by a process. It also helps dissecting memory consumption by a specific Python script, line by line. It can be installed as follows:

```
$ pip install -U memory_profiler
```

Climate just consists of some basic command-line utilities for Python. It can be promptly installed as follows:

```
$ pip install climate
```

Finally, NeuroLab is a very basic neural network package loosely based on the **Neural Network Toolbox** (**NNT**) in MATLAB. It is based on NumPy and SciPy, not Theano; consequently, do not expect astonishing performances but know that it is a good learning toolbox. It can be easily installed as follows:

```
$ pip install neurolab
```

Summary

In this introductory chapter, we have illustrated the different ways in which we can make machine learning algorithms scalable using Python (scale up and scale out techniques). We also proposed some motivating examples and set the stage for the book by illustrating how to install Python on your machine. In particular, we introduced you to Jupyter and covered all the most important packages that will be used in this book.

In the next chapter, we will dive into discussing how stochastic gradient descent can help you deal with massive datasets by leveraging I/O on a single machine. Basically, we will cover different ways of streaming data from large files or data repositories and feed it into a basic learning algorithm. You will be amazed at how simple solutions can be effective, and you will discover that even your desktop computer can easily crunch big data.

Chapter 2. Scalable Learning in Scikit-learn

Loading a dataset into memory, preparing a data matrix, training a machine learning algorithm, and testing its generalization capabilities using out-of-sample observations are often not such a big deal given the quite powerful and yet affordable computers of this day and age. However, more and more frequently, the scale of the data to be elaborated is so huge that loading it into the core memory of your computer is not possible and, even if manageable, the result is intractable both in terms of data management and machine learning.

Alternative viable strategies beyond the core memory processing are possible: splitting the data into samples, using parallelism, and finally learning in small batches or by single instances. The present chapter will focus on the out-of-the-box solution that the Scikit-learn package offers: the streaming of mini batches of instances (our observations) from data storage and the incremental learning based on them. Such a solution is called out-of-core learning.

To treat the data by working on manageable chunks and learning incrementally is a great idea. However, when you try to implement it, it can also prove challenging because of the limitations in the available learning algorithms and streaming data in a flow will require you to think differently in terms of data management and feature extraction. Beyond presenting the Scikit-learn functionalities for out-of-core learning, we will also strive to present you with Python solutions for apparently daunting problems you can face when forced to observe only small portions of your data at a time.

In this chapter, we will cover the following topics:

- The way out-of-core learning is implemented in Scikit-learn
- Effectively managing streams of data using the hashing trick
- The nuts and bolts of stochastic learning
- Implementing data science with online learning
- Unsupervised transformations of streams of data

Out-of-core learning

Out-of-core learning refers to a set of algorithms working with data that cannot fit into the memory of a single computer, but that can easily fit into some data storage such as a local hard disk or web repository. Your available RAM, the core memory on your single machine, may indeed range from a few gigabytes (sometimes 2 GB, more commonly 4 GB, but we assume that you have 2 GB at maximum) up to 256 GB on large server machines. Large servers are like the ones you can get on cloud computing services such as Amazon **Elastic Compute Cloud** (**EC2**), whereas your storage capabilities can easily exceed terabytes of capacity using just an external drive (most likely about 1 TB but it can reach up to 4 TB).

As machine learning is based on globally reducing a cost function, many algorithms initially have been thought to work using all the available data and having access to it at each iteration of the optimization process. This is particularly true for all algorithms based on statistical learning that exploit matrix calculus, for instance, inverting matrices, but also algorithms based on greedy search need to have an evaluation on as much data as is possible before taking the next step. Therefore, the most common out-of-the-box regression-like algorithms (weighted linear combinations of features) update their coefficients trying to minimize the pooled error of the entire dataset. In a similar way, being so sensible

to the noise present in the dataset, decision trees have to decide on the best splits based on all the data available in order to find an optimum solution.

If data cannot fit in the core memory of the computer in such a situation, you don't have many possible solutions. You can increase the available memory (depending on the limitations of the motherboard; after that, you will have to turn to distributed systems such as Hadoop and Spark, a solution we'll mention in the last chapters of the book) or simply reduce your dataset in order to have it fit the memory.

If your data is sparse, that is, you have many zero values in your dataset, you can transform your dense matrix into a sparse one. This is typical with textual data with many columns because each one is a word but with few values representing word counts because single documents usually display a limited selection of words. Sometimes, using sparse matrices can solve the problem allowing you to both load and process other quite large datasets, but this not a panacea (sorry, no free lunch, that is, there is no solution that can fit all problems) because some data matrices, though sparse, can have daunting sizes.

In such a situation, you can always try to reduce your dataset by cutting the number of instances or limiting the number of features, thus reducing the dimensions of the dataset matrix and its occupied area in-memory. Reducing the size of the dataset, by picking only a part of the observations, is a solution called subsampling (or simply sampling). Subsampling is not wrong per se but it has serious drawbacks and it is necessary to keep them in mind before deciding the course of analysis.

Subsampling as a viable option

When you subsample, you are actually discarding part of your informational richness and you cannot be so sure that you are only discarding redundant, not so useful observations. Actually, some hidden gems can be found only by considering all the data. Though computationally appealing—because subsampling just requires a random generator to tell you if you should pick an instance or not—by picking a subsampled dataset, you really risk limiting the capabilities of your algorithm to learn the rules and associations in your data in a complete way. In the bias-variance tradeoff, subsampling causes variance inflation of the predictions because estimates will be more uncertain due to random noise or outlying observations in your data.

In a world of big data, the algorithm with more quality data wins because it can learn more ways to relate predictions to predictors than other models with less (or more noisy) data. Consequently, subsampling, though acceptable as a solution, can impose a limit on the results of your machine learning activities because of less precise predictions and more variance of the estimates.

Subsampling limitations can be somehow overcome by learning multiple models on multiple subsamples of the data and then finally ensembling all the solutions or stacking the results of all the models together, thus creating a reduced data matrix for further training. This procedure is known as Bagging. (You actually compress the features in this way, thus reducing the space of your data in memory.) We will explore ensembling and stacking in a later chapter and discover how they can actually reduce the variance of estimates inflated by subsampling.

As an alternative, instead of cutting the instances, we could cut the features, but again, we will incur the problem that we need to build a model from the data in order to test what features we can select, so we still have to build a model with data that cannot fit in-memory.

Optimizing one instance at a time

Having realized that subsampling, though always viable, is not an optimal solution, we have to evaluate a different approach and out-of-core actually doesn't require you to give up observations or features. It just takes a bit longer to train a model, requiring more iterations and data transfer from your storage to your computer memory. We immediately provide a first intuition of how an out-of-core learning process works.

Let's start from the learning, which is a process where we try to map the unknown function expressing a response (a number or outcome that is a regression or classification problem) with respect to the available data. Learning is possible by fitting the internal coefficients of the learning algorithm trying to achieve the best fit on the data available that is minimizing a cost function, a measure that tells us how good our approximation is. Boiled down to basics, we are talking of an optimization process.

Different optimization algorithms, just like gradient descent, are processes able to work on any volume of data. They work at deriving a gradient for optimization (a direction in the optimization process) and they have the learning algorithm adapt its parameters in order to follow the gradient.

In the specific case of gradient descent, after a certain number of iterations, if the problem can be solved and there are no other problems such as a too high learning rate, the gradient should become so small that we can stop the optimization process. At the end of the process, we can be confident to have found a solution that is the optimum one (because it is a global optimum though sometimes it may be a local minimum, if the function to approximate is not convex).

As the directionality, dictated by the gradient, can be taken based on any volume of examples, it can also be taken on a single instance. Taking the gradient on a single instance requires a small learning rate, but in the end, the process can arrive at the same optimization as a gradient descent taken on the full data. In the end, all our algorithm needs is a direction to orientate correctly the learning process on its fitting the data available. Learning such a direction from a single case randomly taken from the data is therefore perfectly doable:

- We can obtain the same results as if we were working on all our data at one time, though the optimization path may turn a bit rough; if the majority of your observations point to an optimum direction, the algorithm will take that one. The only issue will be to correctly tune the correct parameters of the learning process and pass over the data multiple times in order to be sure for the optimization to complete as this learning procedure is much slower than working with all the data available.
- We don't have any particular issue in managing to keep a single instance in our core memory, leaving the bulk of the data out of it. Other issues may arise from moving the data by single examples from its repository to our core memory. Scalability is assured because the time it takes to process the data is linear; the time cost of using an instance more is always the same, no matter the total number of instances we have to process.

The approach of fitting a learning algorithm on a single instance or a subset of data fitting to memory at a time is called online learning and the gradient descent taken based on such single observations is called stochastic gradient descent. As previously suggested, online learning is an out-of-core technique and adopted by many learning algorithms in Scikit-learn.

Building an out-of-core learning system

We will illustrate the inner workings of a stochastic gradient descent in the next few paragraphs, offering more details and reasoning about it. Now knowing how it is possible to learn out-of-core (thanks to the stochastic gradient descent) allows us to depict with higher clarity what we should do to make it work on our computer.

You can partition your activity into different tasks:

1. Prepare your data repository access to stream the data instance by instance. This activity may require you to randomize the order of the data rows before fetching data to your computer in order to remove any information that ordering may bring about.
2. Do some data surveying first, maybe on a portion of all the data (for instance, the first ten thousand rows), trying to figure out if the arriving instances are consistent in their number of features, type of data, presence or lack of data values, minimum and maximum values for each variable, and mean and median. Find out the range or class of the target variable.
3. Prepare each received data row into a fixed format that can be accepted by the learning algorithm (a dense or sparse vector). At this stage, you can perform any basic transformation, turning categorical features into numeric ones, for instance, or having numeric features interact by themselves by a cross-product of the features themselves.
4. After randomizing the order of examples (as mentioned by the first point), establish a validation procedure using a systematic holdout or a holdout after a certain number of observations.
5. Tune hyperparameters by repeatedly streaming the data or working on small samples of it. This is also the right time to do some feature engineering (using unsupervised learning and special transformation functions such as kernel approximations) and leverage regularization and feature selection.
6. Build your final model using the data that you reserved for the training and ideally test the efficacy of the model on completely new data.

As a first step, we will discuss how to prepare your data and then easily create a stream suitable for online learning, leveraging useful functions from Python packages such as pandas and Scikit-learn.

Streaming data from sources

Some data is really streaming through your computer when you have a generative process that transmits data, which you can process on the fly or just discard, but not recall afterward unless you have stored it away in some data archival repository somewhere. It is like dragging water from a flowing river—the river keeps on flowing but you can filter and process all the water as it goes. It's a completely different strategy from processing all the data at once, which is more like putting all the water in a dam (an analogy for working with all the data in-memory).

As an example of streaming, we could quote the data flow produced instant by instant by a sensor or, even more simply, a Twitter streamline of tweets. Generally, the main sources of data streams are as follows:

- Environment sensors measuring temperature, pressure, and humidity
- GPS tracking sensors recording the location (latitude/longitude)
- Satellites recording image data
- Surveillance videos and sound records
- Web traffic

However, you won't often work on real streams of data but on static records left stored in a repository or file. In such cases, a stream can be recreated according to certain criteria, for example, extracting sequentially or randomly a single record at a time. If, for instance, our data is contained in a TXT or CSV file, all we need to do is fetch a single row of the file at a time and pass it to the learning algorithm.

For the examples in the present and following chapter, we will be working on files stored on your local hard disk and prepare the Python code necessary for its extraction as a stream. We won't use a toy dataset but we won't clutter your local hard drive with too much data for tests and demonstrations.

Datasets to try the real thing yourself

Since 1987, at **University of California, Irvine** (**UCI**), the **UCI Machine Learning Repository** has been hosted, which is a large repository of datasets for the empirical testing of machine learning algorithms by the machine learning community. At the time of writing this, the repository contains about 350 datasets from very different domains and purposes, from supervised regression and classification to unsupervised tasks. You can have a look at the available dataset at https://archive.ics.uci.edu/ml/.

From our side, we have selected a few datasets that will turn useful throughout the book, proposing challenging problems to you with an unusual, but still manageable, 2 GB RAM computer and a high number of rows or columns:

Dataset name	Dataset URL	Type of problem	Rows and columns
Bike-sharing dataset	https://archive.ics.uci.edu/ml/datasets/Bike+Sharing+Dataset	Regression	17389, 16

Dataset name	Dataset URL	Type of problem	Rows and columns
BlogFeedback dataset	https://archive.ics.uci.edu/ml/datasets/BlogFeedback	Regression	60021, 281
Buzz in social media dataset	https://archive.ics.uci.edu/ml/datasets/Buzz+in+social+media+	Regression and classification	140000, 77
Census-Income (KDD) dataset	https://archive.ics.uci.edu/ml/datasets/Census-Income+%28KDD%29	Classification with missing data	299285, 40
Covertype dataset	https://archive.ics.uci.edu/ml/datasets/Covertype	Classification	581012, 54
KDD Cup 1999 dataset	https://archive.ics.uci.edu/ml/datasets/KDD+Cup+1999+Data	Classification	4000000, 42

In order to download and use the dataset from the UCI repository, you have to go to the page dedicated to the dataset and follow the link under the title: **Download: Data Folder**. We have prepared some scripts for automatic downloading of the data that will be placed exactly in the directory that you are working with in Python, thus rendering the data access easier.

Here are some functions that we have prepared and will recall throughout the chapters when we need to download any of the datasets from UCI:

```
In: import urllib2 # import urllib.request as urllib2 in Python3
import requests, io, os, StringIO
import numpy as np
import tarfile, zipfile, gzip

def unzip_from_UCI(UCI_url, dest=''):
    """
    Downloads and unpacks datasets from UCI in zip format
    """
    response = requests.get(UCI_url)
    compressed_file = io.BytesIO(response.content)
    z = zipfile.ZipFile(compressed_file)
    print ('Extracting in %s' %  os.getcwd()+'\\'+dest)
    for name in z.namelist():
        if '.csv' in name:
            print ('\tunzipping %s' %name)
```

```
            z.extract(name, path=os.getcwd()+'\\'+dest)

def gzip_from_UCI(UCI_url, dest=''):
    """
    Downloads and unpacks datasets from UCI in gzip format
    """
    response = urllib2.urlopen(UCI_url)
    compressed_file = io.BytesIO(response.read())
    decompressed_file = gzip.GzipFile(fileobj=compressed_file)
    filename = UCI_url.split('/')[-1][:-3]
    with open(os.getcwd()+'\\'+filename, 'wb') as outfile:
        outfile.write(decompressed_file.read())
    print ('File %s decompressed' % filename)

def targzip_from_UCI(UCI_url, dest='.'):
    """
    Downloads and unpacks datasets from UCI in tar.gz format
    """
    response = urllib2.urlopen(UCI_url)
    compressed_file = StringIO.StringIO(response.read())
    tar = tarfile.open(mode="r:gz", fileobj = compressed_file)
    tar.extractall(path=dest)
    datasets = tar.getnames()
    for dataset in datasets:
        size = os.path.getsize(dest+'\\'+dataset)
        print ('File %s is %i bytes' % (dataset,size))
    tar.close()

def load_matrix(UCI_url):
    """
    Downloads datasets from UCI in matrix form
    """
    return np.loadtxt(urllib2.urlopen(UCI_url))
```

Tip

Downloading the example code

Detailed steps to download the code bundle are mentioned in the Preface of this book. Please have a look.

The code bundle for the book is also hosted on GitHub at https://github.com/PacktPublishing/Large-Scale-Machine-Learning-With-Python. We also have other code bundles from our rich catalog of books and videos available at https://github.com/PacktPublishing/. Check them out!

The functions are just convenient wrappers built around various packages working with compressed data such as `tarfile`, `zipfile`, and `gzip`. The file is opened using the `urllib2` module, which generates a handle to the remote system and allows the sequential transmission of data and being stored

in memory as a string (`StringIO`) or in binary mode (`BytesIO`) from the `io` module—a module devoted to stream handling (https://docs.python.org/2/library/io.html). After being stored in memory, it is recalled just as a file would be from functions specialized in deflating the compressed files from disk.

The four provided functions should conveniently help you download the datasets quickly, no matter if they are zipped, tarred, gzipped, or just plain text in matrix form, avoiding the hassle of manual downloading and extraction operations.

The first example – streaming the bike-sharing dataset

As the first example, we will be working with the bike-sharing dataset. The dataset comprises of two CSV files containing the hourly and daily count of bikes rented in the years between 2011 and 2012 within the Capital Bike-share system in Washington D.C., USA. The data features the corresponding weather and seasonal information regarding the day of rental. The dataset is connected with a publication by *Fanaee-T, Hadi, and Gama, Joao, Event labeling combining ensemble detectors and background knowledge, Progress in Artificial Intelligence (2013): pp. 1-15, Springer Berlin Heidelberg*.

Our first target will be to save the dataset on the local hard disk using the convenient wrapper functions defined just a few paragraphs earlier:

```
In: UCI_url = 'https://archive.ics.uci.edu/ml/
machine-learning-databases/00275/Bike-Sharing-Dataset.zip'
unzip_from_UCI(UCI_url, dest='bikesharing')

Out: Extracting in C:\scisoft\WinPython-64bit-2.7.9.4\notebooks\
bikesharing
    unzipping day.csv
    unzipping hour.csv
```

If run successfully, the code will indicate in what directory the CSV files have been saved and print the names of both the unzipped files.

At this point, having saved the information on a physical device, we will write a script constituting the core of our out-of-core learning system, providing the data streaming from the file. We will first use the `csv` library, offering us a double choice: to recover the data as a list or Python dictionary. We will start with a list:

```
In: import os, csv
local_path = os.getcwd()
source = 'bikesharing\\hour.csv'
SEP = ',' # We define this for being able to easily change it as
required by the file
with open(local_path+'\\'+source, 'rb') as R:
    iterator = csv.reader(R, delimiter=SEP)
    for n, row in enumerate(iterator):
        if n==0:
            header = row
```

```
        else:
            # DATA PROCESSING placeholder
            # MACHINE LEARNING placeholder
            pass
    print ('Total rows: %i' % (n+1))
    print ('Header: %s' % ', '.join(header))
    print ('Sample values: %s' % ', '.join(row))

Out: Total rows: 17380
Header: instant, dteday, season, yr, mnth, hr, holiday, weekday,
workingday, weathersit, temp, atemp, hum, windspeed, casual,
registered, cnt
Sample values: 17379, 2012-12-31, 1, 1, 12, 23, 0, 1, 1, 1, 0.26,
0.2727, 0.65, 0.1343, 12, 37, 49
```

The output will report to us how many rows have been read, the content of the header—the first row of the CSV file (stored in a list)—and the content of a row (for convenience, we printed the last seen one). The `csv.reader` function creates an `iterator` that, thanks to a `for` loop, will release each row of the file one by one. Note that we have placed two remarks internally in the code snippet, pointing out where, throughout the chapter, we will place the other code to handle data preprocessing and machine learning.

Features in this case have to be handled using a positional approach, which is indexing the position of the label in the header. This can be a slight nuisance if you have to manipulate your features extensively. A solution could be to use `csv.DictReader` that produces a Python dictionary as an output (which is unordered but the features may be easily recalled by their labels):

```
In: with open(local_path+'\\'+source, 'rb') as R:
       iterator = csv.DictReader(R, delimiter=SEP)
       for n, row in enumerate(iterator):
           # DATA PROCESSING placeholder
           # MACHINE LEARNING placeholder
           pass
       print ('Total rows: %i' % (n+1))
       print ('Sample values: %s' % str(row))

Out: Total rows: 17379
Sample values: {'mnth': '12', 'cnt': '49', 'holiday': '0',
'instant': '17379', 'temp': '0.26', 'dteday': '2012-12-31', 'hr':
'23', 'season': '1', 'registered': '37', 'windspeed': '0.1343',
'atemp': '0.2727', 'workingday': '1', 'weathersit': '1', 'weekday':
'1', 'hum': '0.65', 'yr': '1', 'casual': '12'}
```

Using pandas I/O tools

As an alternative to the `csv` module, we can use pandas' `read_csv` function. Such a function, specialized in uploading CSV files, is part of quite a large range of functions devoted to input/output on

different file formats, as specified by the pandas documentation at http://pandas.pydata.org/pandas-docs/stable/io.html.

The great advantages of using pandas I/O functions are as follows:

- You can keep your code consistent if you change your source type, that is, you need to redefine just the streaming iterator
- You can access a large number of different formats such as CSV, plain TXT, HDF, JSON, and SQL query for a specific database
- The data is streamed into chunks of the desired size as DataFrame data structures so that you can access the features in a positional way or by recalling their label, thanks to `.loc`, `.iloc`, `.ix` methods typical of slicing and dicing in a pandas dataframe

Here is an example using the same approach as before, this time built around pandas' `read_csv` function:

```
In: import pandas as pd
CHUNK_SIZE = 1000
with open(local_path+'\\'+source, 'rb') as R:
    iterator = pd.read_csv(R, chunksize=CHUNK_SIZE)
    for n, data_chunk in enumerate(iterator):
        print ('Size of uploaded chunk: %i instances, %i features' %
(data_chunk.shape))
        # DATA PROCESSING placeholder
        # MACHINE LEARNING placeholder
        pass
    print ('Sample values: \n%s' % str(data_chunk.iloc[0]))


Out:
Size of uploaded chunk: 2379 instances, 17 features
Size of uploaded chunk: 2379 instances, 17 features
Size of uploaded chunk: 2379 instances, 17 features
Size of uploaded chunk: 2379 instances, 17 features
Size of uploaded chunk: 2379 instances, 17 features
Size of uploaded chunk: 2379 instances, 17 features
Size of uploaded chunk: 2379 instances, 17 features
Sample values:
instant            15001
dteday        2012-09-22
season                 3
yr                     1
mnth                   9
hr                     5
holiday                0
weekday                6
workingday             0
weathersit             1
```

```
temp                   0.56
atemp                0.5303
hum                    0.83
windspeed            0.3284
casual                    2
registered               15
cnt                      17
Name: 0, dtype: object
```

Here, it is very important to notice that the iterator is instantiated by specifying a chunk size, that is, the number of rows the iterator has to return at every iteration. The `chunksize` parameter can assume values from 1 to any value, though clearly the size of the mini-batch (the chunk retrieved) is strictly connected to your available memory to store and manipulate it in the following preprocessing phase.

Bringing larger chunks into memory offers an advantage only in terms of disk access. Smaller chunks require multiple access to the disk and, depending on the characteristics of your physical storage, a longer time to pass through the data. Nevertheless, from a machine learning point of view, smaller or larger chunks make little difference for Scikit-learn out-of-core functions as they learn taking into account only one instance at a time, making them truly linear in computational cost.

Working with databases

As an example of the flexibility of the pandas I/O tools, we will provide a further example using a SQLite3 database where data is streamed from a simple query, chunk by chunk. The example is not proposed for just a didactical use. Working with a large data store in databases can indeed bring advantages from the disk space and processing time point of view.

Data arranged into tables in a SQL database can be normalized, thus removing redundancies and repetitions and saving disk storage. Database normalization is a way to arrange columns and tables in a database in a way to reduce their dimensions without losing any information. Often, this is accomplished by splitting tables and recoding repeated data into keys. Moreover, a relational database, being optimized on memory and operations and multiprocessing, can speed up and anticipate part of those preprocessing activities otherwise dealt within the Python scripting.

Using Python, SQLite (http://www.sqlite.org) is a good default choice because of the following reasons:

- It is open source
- It can handle large amounts of data (theoretically up to 140 TB per database, though it is unlikely to see any SQLite application dealing with such amounts of data)
- It operates on macOS and both Linux and Windows 32- and 64-bit environments
- It does not require any server infrastructure or particular installation (zero configuration) as all the data is stored in a single file on disk
- It can be easily extended using Python code to be turned into a stored procedure

Moreover, the Python standard library includes a `sqlite3` module providing all the functions to create a database from scratch and work with it.

In our example, we will first upload the CSV file containing the bike-sharing dataset on both a daily and hourly basis to a SQLite database and then we will stream from it as we did from a CSV file. The database uploading code that we provide can be reusable throughout the book and for your own applications, not being tied to the specific example we provide (you just have to change the input and output parameters, that's all):

```
In : import os, sys
import sqlite3, csv,glob

SEP = ','

def define_field(s):
    try:
        int(s)
        return 'integer'
    except ValueError:
        try:
            float(s)
            return 'real'
        except:
            return 'text'

def create_sqlite_db(db='database.sqlite', file_pattern=''):
    conn = sqlite3.connect(db)
    conn.text_factory = str  # allows utf-8 data to be stored

    c = conn.cursor()

    # traverse the directory and process each .csv file useful for
building the db
    target_files = glob.glob(file_pattern)

    print ('Creating %i table(s) into %s from file(s): %s' %
(len(target_files), db, ', '.join(target_files)))

    for k,csvfile in enumerate(target_files):
        # remove the path and extension and use what's left as a
table name
        tablename = os.path.splitext(os.path.basename(csvfile))[0]

        with open(csvfile, "rb") as f:
            reader = csv.reader(f, delimiter=SEP)

            f.seek(0)
            for n,row in enumerate(reader):
                if n==11:
                    types = map(define_field,row)
```

```
                else:
                    if n>11:
                        break

            f.seek(0)
            for n,row in enumerate(reader):
                if n==0:

                    sql = "DROP TABLE IF EXISTS %s" % tablename
                    c.execute(sql)
                    sql = "CREATE TABLE %s (%s)" % (tablename,\
                                ", ".join([ "%s %s" % (col, ct) \
for col, ct  in zip(row, types)]))
                    print ('%i) %s' % (k+1,sql))
                    c.execute(sql)

                    # Creating indexes for faster joins on long
strings
                    for column in row:
                        if column.endswith("_ID_hash"):
                            index = "%s__%s" % \
( tablename, column )
                            sql = "CREATE INDEX %s on %s (%s)" % \
( index, tablename, column )
                            c.execute(sql)

                    insertsql = "INSERT INTO %s VALUES (%s)" %
(tablename,
                                ", ".join([ "?" for column in row ]))

                    rowlen = len(row)
                else:
                    # raise an error if there are rows that don't
have the right number of fields
                    if len(row) == rowlen:
                        c.execute(insertsql, row)
                    else:
                        print ('Error at line %i in file %s') %
(n,csvfile)
                        raise ValueError('Houston, we\'ve had a
problem at row %i' % n)

            conn.commit()
            print ('* Inserted %i rows' % n)

    c.close()
    conn.close()
```

The script provides a valid database name and pattern to locate the files that you want to import (wildcards such as `*` are accepted) and creates from scratch a new database and table that you need, filling them afterwards with all the data available:

```
In: create_sqlite_db(db='bikesharing.sqlite',
file_pattern='bikesharing\\*.csv')

Out: Creating 2 table(s) into bikesharing.sqlite from file(s):
bikesharing\day.csv, bikesharing\hour.csv
1) CREATE TABLE day (instant integer, dteday text, season integer,
yr integer, mnth integer, holiday integer, weekday integer,
workingday integer, weathersit integer, temp real, atemp real, hum
real, windspeed real, casual integer, registered integer, cnt
integer)
* Inserted 731 rows
2) CREATE TABLE hour (instant integer, dteday text, season integer,
yr integer, mnth integer, hr integer, holiday integer, weekday
integer, workingday integer, weathersit integer, temp real, atemp
real, hum real, windspeed real, casual integer, registered integer,
cnt integer)
* Inserted 17379 rows
```

The script also reports the data types for the created fields and number of rows, so it is quite easy to verify that everything has gone smoothly during the importation. Now it is easy to stream from the database. In our example, we will create an inner join between the hour and day tables and extract data on an hourly base with information about the total rentals of the day:

```
In: import os, sqlite3
import pandas as pd

DB_NAME = 'bikesharing.sqlite'
DIR_PATH = os.getcwd()
CHUNK_SIZE = 2500

conn = sqlite3.connect(DIR_PATH+'\\'+DB_NAME)
conn.text_factory = str  # allows utf-8 data to be stored
sql = "SELECT H.*, D.cnt AS day_cnt FROM hour AS H INNER JOIN day as
D ON (H.dteday = D.dteday)"
DB_stream = pd.io.sql.read_sql(sql, conn, chunksize=CHUNK_SIZE)
for j,data_chunk in enumerate(DB_stream):
    print ('Chunk %i -' % (j+1)),
    print ('Size of uploaded chunk: %i instances, %i features' %
(data_chunk.shape))
    # DATA PROCESSING placeholder
    # MACHINE LEARNING placeholder

Out:
```

```
Chunk 1 - Size of uploaded chunk: 2500 instances, 18 features
Chunk 2 - Size of uploaded chunk: 2500 instances, 18 features
Chunk 3 - Size of uploaded chunk: 2500 instances, 18 features
Chunk 4 - Size of uploaded chunk: 2500 instances, 18 features
Chunk 5 - Size of uploaded chunk: 2500 instances, 18 features
Chunk 6 - Size of uploaded chunk: 2500 instances, 18 features
Chunk 7 - Size of uploaded chunk: 2379 instances, 18 features
```

If you need to speed up the streaming, you just have to optimize the database, first of all building the right indexes for the relational query that you intend to use.

Tip

`conn.text_factory = str` is a very important part of the script; it allows UTF-8 data to be stored. If such a command is ignored, you may experience strange errors when inputting data.

Paying attention to the ordering of instances

As a concluding remark for the streaming data topic, we have to warn you about the fact that, when streaming, you are actually including hidden information in your learning process because of the order of the examples you are basing your learning on.

In fact, online learners optimize their parameters based on each instance that they evaluate. Each instance will lead the learner toward a certain direction in the optimization process. Globally, the learner should take the right optimization direction, given a large enough number of evaluated instances. However, if the learner is instead trained by biased observations (for instance, observations ordered by time or grouped in a meaningful way), the algorithm will also learn the bias. Something can be done during training in order to not remember previously seen instances, but some bias will be introduced anyway. If you are learning time series—the response to the flow of time often being part of the model—such a bias is quite useful, but in most other cases, it acts as some kind of overfitting and translates into a certain lack of generalization in the final model.

If your data has some kind of ordering which you don't want to be learned by the machine learning algorithm (such as an ID order), as a cautionary measure, you can shuffle its rows before streaming the data and obtain a random order more suitable for online stochastic learning.

The fastest way, and the one occupying less space on disk, is to stream the dataset in memory and shrink it by compression. In most cases, but not all, this will work thanks to the compression algorithm applied and the relative sparsity and redundancy of the data that you are using for the training. In the cases where it doesn't work, you have to shuffle the data directly on the disk implying more disk space consumption.

Here, we first present a fast way to shuffle in-memory, thanks to the `zlib` package that can rapidly compress the rows into memory, and the `shuffle` function from the `random` module:

```
In: import zlib
from random import shuffle
```

```
def ram_shuffle(filename_in, filename_out, header=True):
    with open(filename_in, 'rb') as f:
        zlines = [zlib.compress(line, 9) for line in f]
        if header:
            first_row = zlines.pop(0)
    shuffle(zlines)
    with open(filename_out, 'wb') as f:
        if header:
            f.write(zlib.decompress(first_row))
        for zline in zlines:
            f.write(zlib.decompress(zline))

import os

local_path = os.getcwd()
source = 'bikesharing\\hour.csv'
ram_shuffle(filename_in=local_path+'\\'+source, \

filename_out=local_path+'\\bikesharing\\shuffled_hour.csv',
header=True)
```

Tip

For Unix users, the `sort` command, which can be easily used with a single invocation (the `-R` parameter), shuffles huge amounts of text files very easily and much more efficiently than any Python implementation. It can be combined with decompression and compression steps using pipes.

So something like the following command should do the trick:

```
zcat sorted.gz | sort -R | gzip - > shuffled.gz
```

In case the RAM is not enough to store all the compressed data, the only viable solution is to operate on the file as it is on the disk itself. The following snippet of code defines a function that will repeatedly split your file into increasingly smaller files, shuffle them internally, and arrange them again randomly in a larger file. The result is not a perfect random rearrangement, but the rows are scattered around enough to destroy any previous order that could influence online learning:

```
In: from random import shuffle
import pandas as pd
import numpy as np
import os

def disk_shuffle(filename_in, filename_out, header=True, iterations
= 3, CHUNK_SIZE = 2500, SEP=','):
    for i in range(iterations):
        with open(filename_in, 'rb') as R:
            iterator = pd.read_csv(R, chunksize=CHUNK_SIZE)
            for n, df in enumerate(iterator):
```

```
                if n==0 and header:
                    header_cols =SEP.join(df.columns)+'\n'

df.iloc[np.random.permutation(len(df))].to_csv(str(n)+'_chunk.csv',
index=False, header=False, sep=SEP)
        ordering = list(range(0,n+1))
        shuffle(ordering)
        with open(filename_out, 'wb') as W:
            if header:
                W.write(header_cols)
            for f in ordering:
                with open(str(f)+'_chunk.csv', 'r') as R:
                    for line in R:
                        W.write(line)
                os.remove(str(f)+'_chunk.csv')
        filename_in = filename_out
        CHUNK_SIZE = int(CHUNK_SIZE / 2)

import os

local_path = os.getcwd()
source = 'bikesharing\\hour.csv'
disk_shuffle(filename_in=local_path+'\\'+source, \

filename_out=local_path+'\\bikesharing\\shuffled_hour.csv',
header=True)
```

Stochastic learning

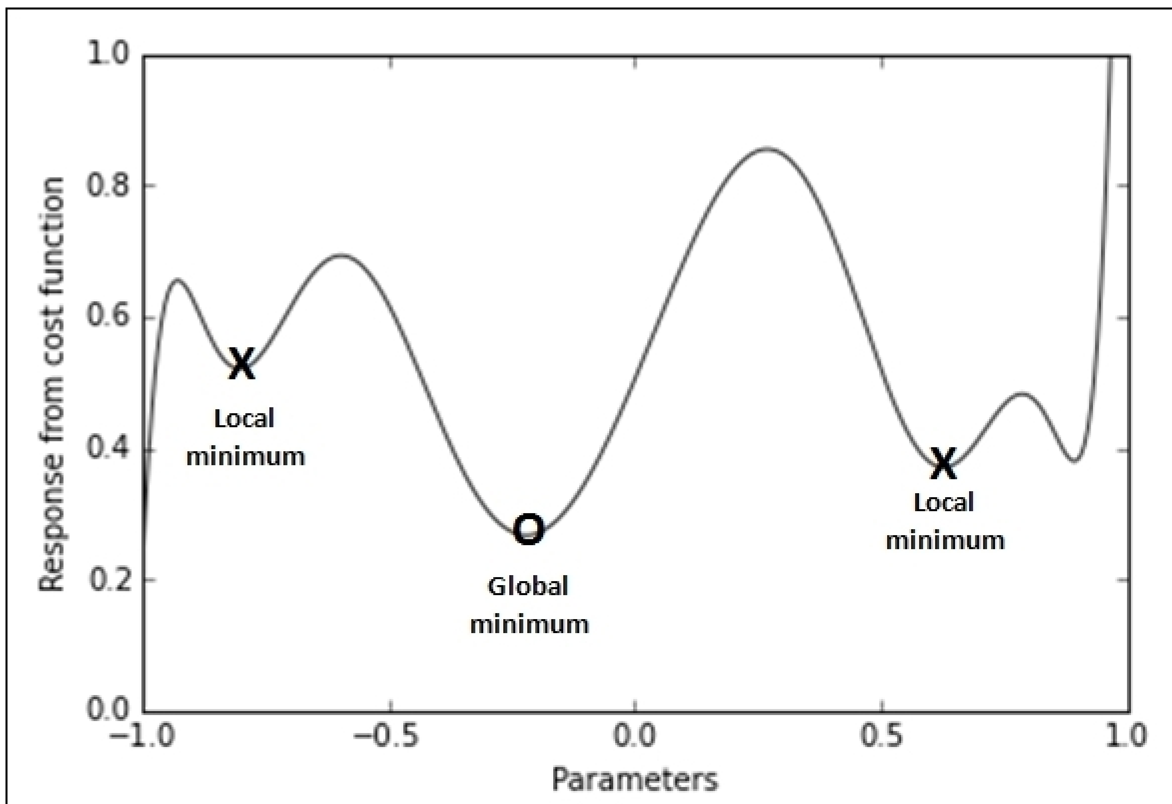
Having defined the streaming process, it is now time to glance at the learning process as it is the learning and its specific needs that determine the best way to handle data and transform it in the preprocessing phase.

Online learning, contrary to batch learning, works with a larger number of iterations and gets directions from each single instance at a time, thus allowing a more erratic learning procedure than an optimization made on a batch, which immediately tends to get the right direction expressed from the data as a whole.

Batch gradient descent

The core algorithm for machine learning, gradient descent, is therefore revisited in order to adapt to online learning. When working on batch data, gradient descent can minimize the cost function of a linear regression analysis using much less computations than statistical algorithms. The complexity of gradient descent ranks in the order $O(n*p)$, making learning regression coefficients feasible even in the occurrence of a large n (which stands for the number of observations) and large p (number of variables). It also works perfectly when highly correlated or even identical features are present in the training data.

Everything is based on a simple optimization method: the set of parameters is changed through multiple iterations in a way that it gradually converges to the optimal solution starting from a random one. Gradient descent is a theoretically well-understood optimization method with known convergence guarantees for certain problems such as regression ones. Nevertheless, let's start with the following image representing a complex mapping (typical of neural networks) between the values that the parameters can take (representing the hypothesis space) and result in terms of minimization of the cost function:

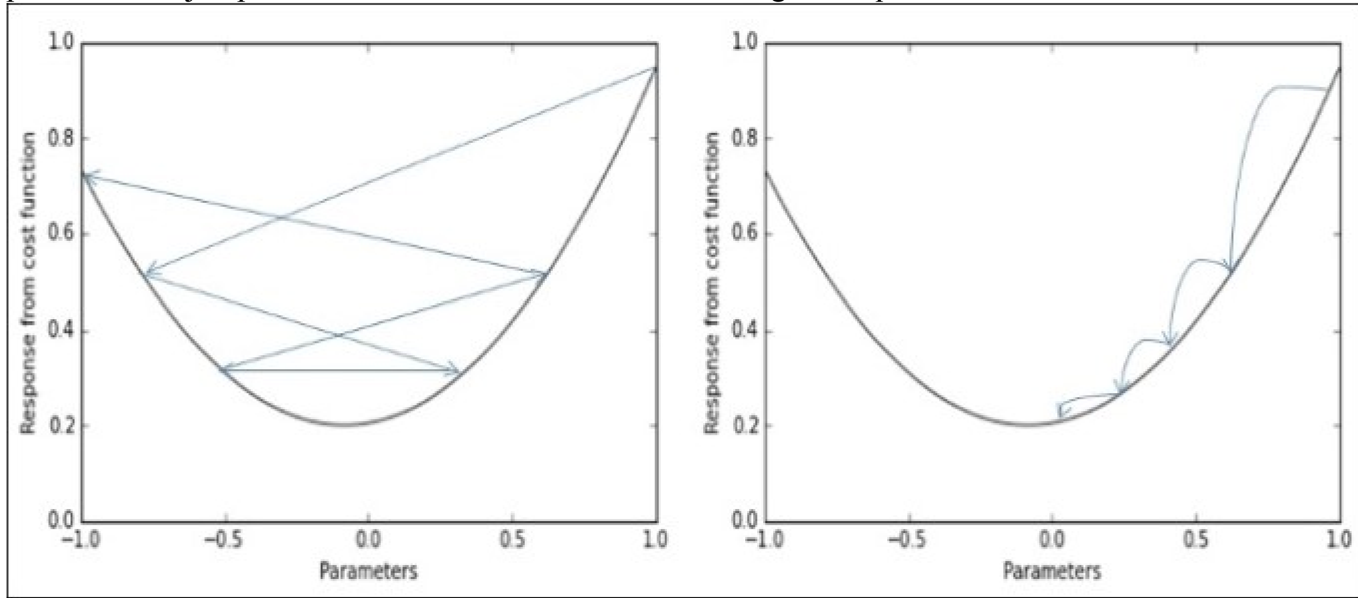


Using a figurative example, gradient descent resembles walking blindfolded around mountains. If you want to descend to the lowest valley without being able to see the path, you can just proceed by taking the direction that you feel is going downhill; try it for a while, then stop, feel the terrain again, and then proceed toward where you feel it is going downhill, and so on, again and again. If you keep on heading toward where the surface descends, you will finally arrive at a point when you cannot descend anymore because the terrain is flat. Hopefully, at that point, you should have reached your destination.

Using such a method, you need to perform the following actions:

- Decide the starting point. This is usually achieved by an initial random guess of the parameters of your function (multiple restarts will ensure that the initialization won't cause the algorithm to reach a local optimum because of an unlucky initial setting).
- Be able to feel the terrain, that is, be able to tell when it goes down. In mathematical terms, this means that you should be able to take the derivative of your actual parameterized function with respect to your target variable, that is, the partial derivative of the cost function that you are optimizing. Note that the gradient descent works on all of your data, trying to optimize the predictions from all your instances at once.
- Decide how long you should follow the direction dictated by the derivative. In mathematical terms, this corresponds to a weight (usually called alpha) to decide how much you should change your parameters at every step of the optimization. This aspect can be considered as the learning factor because it points out how much you should learn from the data at each optimization step. As with any other hyperparameter, the best value of alpha can be determined by performance evaluation on a validation set.

- Determine when to stop, given a too marginal improvement of the cost function with respect to the previous step. In such a sense, you also should be able to notice when something goes wrong and you are not going in the right direction maybe because you are using too large an alpha for the learning. This is actually a matter of *momentum*, that is, the speed at which the algorithm converges toward the optimum. It is just like throwing a ball down a mountainside: it just rolls over small dents in the surface, but if its speed is too high, it won't stop at the right point. Thus, if alpha is set correctly, the momentum will naturally slow down as the algorithm is approaching the optimum as shown in the following image in the right panel. However, if it is not set properly, it will just jump over the global optimum and report further errors to be minimized, as depicted in the following image on the right panel, when the optimization process causes parameters to jump across different values without achieving the required error minimization:



In order to better depict what happens with gradient descent, let's take the example of a linear regression whose parameters are optimized by such a procedure.

We start from the cost function *J* given the vector of weights *w*:

$$J(w) = \frac{1}{2n}\sum(Xw - y)^2$$

The matrix-vector multiplication *Xw* between the training data matrix *X* and the vector of coefficients *w* represents the predictions from the linear model, whose deviance from the response *y* is squared, then summed, and finally divided by two times *n*, the number of instances.

Initially, the vector *w* could be instantiated using random numbers taken from the standardized normal distribution whose mean is zero and standard deviation is the unit. (Actually, initialization can be done

in a lot of different ways, all working equally well to approximate linear regression whose cost function is bowl-shaped and has a unique minimum.) This allows our algorithm to start somewhere along the optimization path and could effectively speed up the convergence of the process. As we are optimizing a linear regression, initialization shouldn't cause much trouble to the algorithm (at worst, a wrong start will just slow it down). Instead, when we are using gradient descent in order to optimize different machine learning algorithms such as neural networks, we risk being stuck because of a wrong initialization. This will happen if, for instance, the initial *w* is just filled with zero values (the risk is getting stuck on a perfectly symmetric hilltop, where no directionality can immediately bring an optimization better than any other). This can happen with optimization processes that have multiple local minima, too.

Given the starting random coefficients vector *w*, we can immediately calculate the cost function *J(w)* and determine the initial direction for each single coefficient by subtracting from each a portion alpha (α, the learning rate) of the partial derivative of the cost function, as explicated by the following formula:

$$w_j = w_j - \alpha \frac{\partial}{\partial w} J(w)$$

This can be better conveyed after solving the partial derivative, as follows:

$$w_j = w_j - \alpha \frac{1}{n} \sum (Xw - y) x_j$$

Noticeably, the update is done on each singular coefficient (*wj*) given its feature vector *xj*, but based on all the predictions at once (hence the summation).

After iterating over all the coefficients in *w*, the coefficients' update will be completed and the optimization may restart again by calculating the partial derivative and updating the *w* vector.

An interesting characteristic of the process is that the update will be less and less as the *w* vector approaches the optimal configuration. Therefore, the process could stop when the change induced in w, with respect to the previous operation, is small. Anyway, it is true that we have decreasing updates when alpha, the learning rate, is set to the right size. In fact, if its value is too large, it may cause the optimization to detour and fail, causing—in some cases—a complete divergence of the process and the impossibility to converge finally to a solution. In fact, the optimization will tend to overshoot the target and actually get farther away from it.

At the other end, too small an alpha value will not only move the optimization process toward its target too slowly, but it may also be easily stuck somewhere in a local minima. This is especially true with regard to more complex algorithms, just like neural networks. As for linear regression and its

classification counterpart, logistic regression, because the optimization curve is bowl-shaped, just like a concave curve, it features a single minimum and no local minima at all.

In the implementation that we illustrated, alpha is a fixed constant (a fixed learning rate gradient descent). As alpha plays such an important role in converging to an optimal solution, different strategies have been devised for it to start larger and shrink as the optimization goes on. We will discuss such different approaches when examining the Scikit-learn implementation.

Stochastic gradient descent

The version of the gradient descent seen so far is known as full batch gradient descent and works by optimizing the error of the entire dataset, and thus needs it in-memory. The out-of-core versions are the **stochastic gradient descent** (**SGD**) and mini batch gradient descent.

Here, the formulation stays exactly the same, but for the update; the update is done for a single instance at a time, thus allowing us to leave the core data in its storage and take just a single observation in-memory:

$$w_j = w_j - \alpha(x_j w - y)x_j$$

The core idea is that, if the instances are picked randomly without particular biases, the optimization will move on average toward the target cost minimization. That explains why we discussed how to remove any ordering from a stream and making it as random as possible. For instance, in the bike-sharing example, if you have stochastic gradient descent learn the patterns of the early season first, then focus on the summer, then on the fall, and so on, depending on the season when the optimization is stopped, the model will be tuned to predict one season better than the others because most of the recent examples are from that season. In a stochastic gradient descent optimization, when data is **independent and identically distributed** (**i.i.d.**), convergence to the global minimum is guaranteed. Practically, i.i.d. means that your examples should have no sequential order or distribution but should be proposed to the algorithm as if picked randomly from your available ones.

The Scikit-learn SGD implementation

A good number of online learning algorithms can be found in the Scikit-learn package. Not all machine learning algorithms have an online counterpart, but the list is interesting and steadily growing. As for supervised learning, we can divide available learners into classifiers and regressors and enumerate them.

As classifiers, we can mention the following:

- `sklearn.naive_bayes.MultinomialNB`
- `sklearn.naive_bayes.BernoulliNB`
- `sklearn.linear_model.Perceptron`
- `sklearn.linear_model.PassiveAggressiveClassifier`
- `sklearn.linear_model.SGDClassifier`

As regressors, we have two options:

- `sklearn.linear_model.PassiveAggressiveRegressor`
- `sklearn.linear_model.SGDRegressor`

They all can learn incrementally, updating themselves instance by instance; though only `SGDClassifier` and `SGDRegressor` are based on the stochastic gradient descent optimization that we previously described, and they are the main topics of this chapter. The SGD learners are optimal for all large-scale problems as their complexity is bound to $O(k*n*p)$, where k is the number of passes over the data, n is the number of instances, and p is the number of features (naturally non-zero features if we are working with sparse matrices): a perfectly linear time learner, taking more time exactly in proportion to the number of examples shown.

Other online algorithms will be used as a comparative benchmark. Moreover, all algorithms have the usage of the same API in common, based on the `partial_fit` method for online learning and mini-batch (when you stream larger chunks rather than a single instance). Sharing the same API makes all these learning techniques interchangeable in your learning frame.

Contrary to the fit method, which uses all the available data for its immediate optimization, `partial_fit` operates a partial optimization based on each of the single instances passed. Even if a dataset is passed to the `partial_fit` method, the algorithm won't process the entire batch but for its single elements, making the complexity of the learning operations indeed linear. Moreover, a learner, after `partial_fit`, can be perpetually updated by subsequent `partial_fit` calls, making it perfect for online learning from continuous streams of data.

When classifying, the only caveat is that at the first initialization, it is necessary to know how many classes we are going to learn and how they are labeled. This can be done using the classes parameter, pointing out a list of the numeric values labels. This requires to be explored beforehand, streaming through the data in order to record the labels of the problem and also taking notice of their distribution in case they are unbalanced—a class is numerically too large or too small with respect to the others (but the Scikit-learn implementation offers a way to automatically handle the problem). If the target variable is numeric, it is still useful to know its distribution, but this is not necessary to successfully run the learner.

In Scikit-learn, we have two implementations—one for classification problems (`SGDClassifier`) and one for regression ones (`SGDRegressor`). The classification implementation can handle multiclass problems using the **one-vs-all** (**OVA**) strategy. This strategy implies that, given k classes, k models are built, one for each class against all the instances of other classes, therefore creating k binary classifications. This will produce k sets of coefficients and k vectors of predictions and their probability. In the end, based on the emitted probability of each class compared against the other, the classification is assigned to the class with the highest probability. If we need to give actual probabilities for the multinomial distribution, we can simply normalize the results by dividing by their sum. (This is what is happening in a softmax layer in a neural network, which we will see in the following chapters.)

Both classification and regression SGD implementations in Scikit-learn feature different loss functions (the cost function, the core of the stochastic gradient descent optimization).

For classification, expressed by the `loss` parameter, we can rely on the following:

- `loss='log'`: Classical logistic regression
- `loss='hinge'`: Softmargin, that is, a linear support vector machine
- `loss='modified_huber'`: A smoothed hinge loss

For regression, we have three loss functions:

- `loss='squared_loss'`: **Ordinary least squares** (**OLS**) for linear regression
- `loss='huber'`: Huber loss for robust regression against outliers
- `loss='epsilon_insensitive'`: A linear support vector regression

We will present some examples using the classical statistical loss functions, which are logistic loss and OLS. Hinge loss and **support vector machines** (**SVMs**) will be discussed in the next chapter, a detailed introduction about their functioning being necessary.

As a reminder (so that you won't have to go and consult any other supplementary machine learning book), if we define the regression function as h and its predictions are given by *h(X)* because *X* is the matrix of features, then the following is the suitable formulation:

$$y \approx h(X) = \beta X + \beta_0$$

Consequently, the OLS cost function to be minimized is as follows:

$$\frac{1}{2n} * \sum (h(X) - y)^2$$

In logistic regression, having a transformation of the binary outcome *0*/*1* into an odds ratio, πy being the probability of a positive outcome, the formula is as follows:

$$y \approx h(X) = \frac{1}{1 + e^{\beta X + \beta_0}}$$

The logistic cost function, consequently, is defined as follows:

$$-\frac{1}{n} * \sum \left[y * \ln\left(h(X)\right) + (1 - y) * \ln\left(1 - h(X)\right) \right]$$

Defining SGD learning parameters

To define SGD parameters in Scikit-learn, both in classification and regression problems (so that they are valid for both `SGDClassifier` and `SGDRegressor`), we have to make clear how to deal with some important parameters necessary for a correct learning when you cannot evaluate all the data at once.

The first one is `n_iter`, which defines the number of iterations through the data. Initially set to *5*, it has been empirically shown that it should be tuned in order for the learner, given the other default parameters, to see around `10^6` examples; therefore a good solution to set it would be `n_iter = np.ceil(10**6 / n)`, where *n* is the number of instances. Noticeably, `n_iter` only works with in-memory datasets, so it acts only when you operate by the fit method but not with `partial_fit`. In reality, `partial_fit` will reiterate over the same data just if you restream it in your procedure and the right number of iterations of restreams is something to be tested along the learning procedure itself, being influenced by the type of data. In the next chapter, we will illustrate hyperparameter optimization and the right number of passes will be discussed.

Tip

It might make sense to reshuffle the data after each complete pass over all of the data when doing mini-batch learning.

`shuffle` is a parameter required if you want to shuffle your data. It refers to the mini-batch present in-memory and not to out-of-core data ordering. It also works with `partial_fit` but its effect in such a case is very limited. Always set it to True, but for data to be passed in chunks, shuffle your data out-of-core, as we described before.

`warm_start` is another parameter that works with the fit method because it remembers the previous fit coefficients (but not the learning rate if it has been dynamically modified). If you are using the `partial_fit` method, the algorithm will remember the previously learned coefficients and the state of the learning rate schedule.

The `average` parameter triggers a computational trick that, at a certain instance, starts averaging new coefficients with older ones allowing a faster convergence. It can be set to `True` or an integer value indicating from what instance it will start averaging.

Last, but not least, we have `learning_rate` and its related parameters, `eta0` and `power_t`. The `learning_rate` parameter implies how each observed instance impacts on the optimization process. When presenting SGD from a theoretical point of view, we presented constant rate learning, which can be replicated setting `learning_rate='constant'`.

However, other options are present, letting the eta η (called the learning rate in Scikit-learn and defined at time *t*) gradually decrease. In classification, the solution proposed is `learning_rate='optimal'`, given by the following formulation:

$$\eta^t = \frac{1}{\alpha_{t0} + \alpha_t}$$

Here, *t* is the time steps, given by the number of instances multiplied by iterations, and *t0* is a value heuristically chosen because of the studies by Léon Bottou, whose version of the *Stochastic Gradient SVM* has heavily influenced the SGD Scikit-learn implementation (http://leon.bottou.org/projects/sgd). The clear advantage of such a learning strategy is that learning decreases as more examples are seen, avoiding sudden perturbations of the optimization given by unusual values. Clearly, this strategy is also out-of-the-box, meaning that you don't have much to do with it.

In regression, the suggested learning fading is given by this formulation, corresponding to `learning_rate= 'invscaling'`:

$$\eta^t = \frac{eta_0}{t^{power_t}}$$

Here, `eta0` and `power_t` are hyperparameters to be optimized by an optimization search (they are initially set to `0` and `0.5`). Noticeably, using the `invscaling` learning rate, SGD will start with a lower learning rate, less than the optimal rate one, and it will decrease more slowly, being a bit more adaptable during learning.

Feature management with data streams

Data streams pose the problem that you cannot evaluate as you would do when working on a complete in-memory dataset. For a correct and optimal approach to feed your SGD out-of-core algorithm, you first have to survey the data (by taking a chuck of the initial instances of the file, for example) and find out the type of data you have at hand.

We distinguish among the following types of data:

- Quantitative values
- Categorical values encoded with integer numbers
- Unstructured categorical values expressed in textual form

When data is quantitative, it could just be fed to the SGD learner but for the fact that the algorithm is quite sensitive to feature scaling; that is, you have to bring all the quantitative features into the same range of values or the learning process won't converge easily and correctly. Possible scaling strategies are converting all the values in the range [0,1], [-1,1] or standardizing the variable by centering its mean to zero and converting its variance to the unit. We don't have particular suggestions for the choice of the scaling strategy, but converting in the range [0,1] works particularly well if you are dealing with a sparse matrix and most of your values are zero.

As for in-memory learning, when transforming variables on the training set, you have to take notice of the values that you used (Basically, you need to get minimum, maximum, mean, and standard deviation of each feature.) and reuse them in the test set in order to achieve consistent results.

Given the fact that you are streaming data and it isn't possible to upload all the data in-memory, you have to calculate them by passing through all the data or at least a part of it (sampling is always a viable solution). The situation of working with an ephemeral stream (a stream you cannot replicate) poses even more challenging problems; in fact, you have to constantly keep trace of the values that you keep on receiving.

If sampling just requires you to calculate your statistics on a chunk of *n* instances (under the assumption that your stream has no particular order), calculating statistics on the fly requires you to record the right measures.

For minimum and maximum, you need to store a variable each for every quantitative feature. Starting from the very first value, which you will store as your initial minimum and maximum, for each new value that you will receive from the stream you will have to confront it with the previously recorded minimum and maximum values. If the new instance is out of the previous range of values, you just update your variable accordingly.

In addition, the average doesn't pose any particular problems because you just need to keep a sum of the values seen and a count of the instances. As for variance, you need to recall that the textbook formulation is as follows:

$$\sigma^2 = \frac{1}{n}\sum(x-\mu)^2$$

Noticeably, you need to know the mean μ, which you are also just learning incrementally from the stream. However, the formulation can be explicated as follows:

$$\sigma^2 = \frac{1}{n}\left(\sum x^2 - \frac{(\sum x)^2}{n}\right)$$

As you are just recording the number *n* of instances and a summation of *x* values, you just need to store another variable, which is the summation of values of *x* squared, and you will have all the ingredients for the recipe.

As an example, using the bike-sharing dataset, we can calculate the running mean, standard deviation, and range reporting the final result and plot how such stats changed as data was streamed from disk:

```
In: import os, csv
local_path = os.getcwd()
source = 'bikesharing\\hour.csv'
SEP=','
running_mean = list()
running_std = list()
with open(local_path+'\\'+source, 'rb') as R:
    iterator = csv.DictReader(R, delimiter=SEP)
    x = 0.0
    x_squared = 0.0
    for n, row in enumerate(iterator):
        temp = float(row['temp'])
        if n == 0:
            max_x, min_x = temp, temp
        else:
            max_x, min_x = max(temp, max_x),min(temp, min_x)
        x += temp
        x_squared += temp**2
        running_mean.append(x / (n+1))
        running_std.append(((x_squared - (x**2)/(n+1))/(n+1))**0.5)
        # DATA PROCESSING placeholder
        # MACHINE LEARNING placeholder
        pass
    print ('Total rows: %i' % (n+1))
```

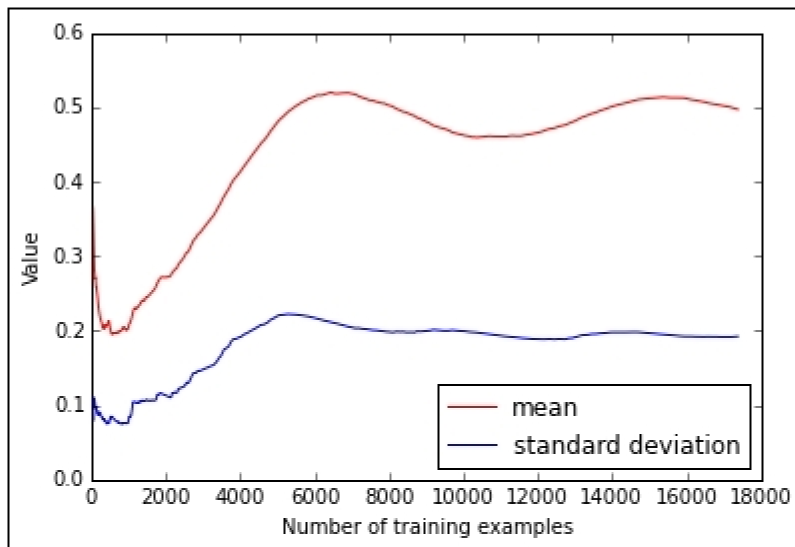
```
    print ('Feature \'temp\': mean=%0.3f, max=%0.3f, min=%0.3f,\
sd=%0.3f' % (running_mean[-1], max_x, min_x, running_std[-1]))

Out: Total rows: 17379
Feature 'temp': mean=0.497, max=1.000, min=0.020, sd=0.193
```

In a few moments, the data will be streamed from the source and key figures relative to the `temp` feature will be recorded as a running estimation of the mean and standard deviation is calculated and stored in two separated lists.

By plotting the values present in the lists, we can examine how much the estimates fluctuated with respect to the final figures and get an idea about how many instances are required before getting a stable mean and standard deviation estimate:

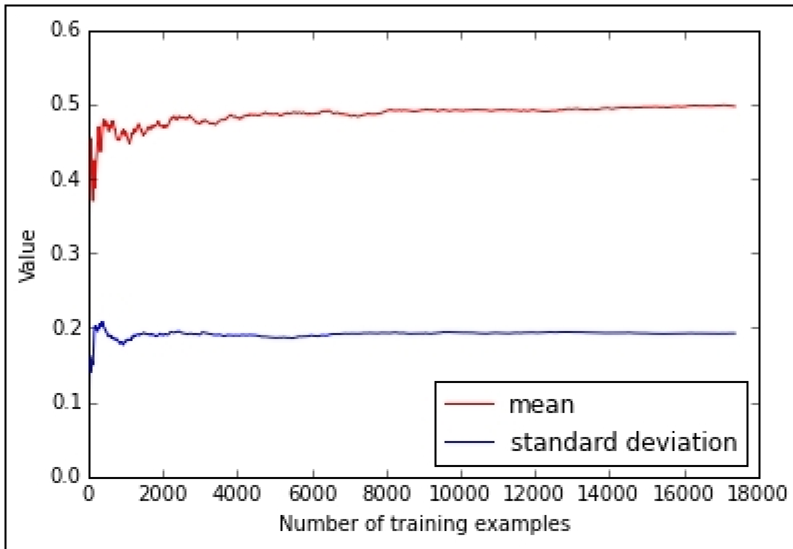
```
In: import matplotlib.pyplot as plt
%matplotlib inline
plt.plot(running_mean,'r-', label='mean')
plt.plot(running_std,'b-', label='standard deviation')
plt.ylim(0.0,0.6)
plt.xlabel('Number of training examples')
plt.ylabel('Value')
plt.legend(loc='lower right', numpoints= 1)
plt.show()
```

If you previously processed the original bike-sharing dataset, you will obtain a plot where clearly there is a trend in the data (due to temporal ordering, because the temperature naturally varies with seasons):



On the contrary, if we had used the shuffled version of the dataset as a source, the `shuffled_hour.csv` file, we could have obtained a couple of much more stable and quickly

converging estimates. Consequently, we would have learned an approximate but more reliable estimate of the mean and standard deviation observing fewer instances from the stream:



The difference in the two charts reminds us of the importance of randomizing the order of the observations. Even learning simple descriptive statistics can be influenced heavily by trends in the data; consequently, we have to pay more attention when learning complex models by SGD.

Describing the target

In addition, the target variable also needs to be explored before starting. We need, in fact, to be sure about what values it assumes, if categorical, and figure out if it is unbalanced when in classes or has a skewed distribution when a number.

If we are learning a numeric response, we can adopt the same strategy shown previously for the features, whereas for classes, a Python dictionary keeping a count of classes (the keys) and their frequencies (the values) will suffice.

As an example, we will download a dataset for classification, the **Forest Covertype** data.

For a fast download and preparation of the data, we will use the `gzip_from_UCI` function as defined in the *Datasets to try the real thing yourself* section of the present chapter:

```
In: UCI_url = 'https://archive.ics.uci.edu/ml/
machine-learning-databases/covtype/covtype.data.gz'
gzip_from_UCI(UCI_url)
```

In case of problems in running the code, or if you prefer to prepare the file by yourself, just go to the UCI website, download the dataset, and unpack it on the directory where Python is currently working:

https://archive.ics.uci.edu/ml/machine-learning-databases/covtype/

Once the data is available on disk, we can scan through the 581,012 instances, converting the last value of each row, representative of the class that we should estimate, to its corresponding forest cover type:

```
In: import os, csv
local_path = os.getcwd()
source = 'covtype.data'
SEP=','
forest_type = {1:"Spruce/Fir", 2:"Lodgepole Pine", \
                3:"Ponderosa Pine", 4:"Cottonwood/Willow",\
                5:"Aspen", 6:"Douglas-fir", 7:"Krummholz"}
forest_type_count = {value:0 for value in forest_type.values()}
forest_type_count['Other'] = 0
lodgepole_pine = 0
spruce = 0
proportions = list()
with open(local_path+'\\'+source, 'rb') as R:
    iterator = csv.reader(R, delimiter=SEP)
    for n, row in enumerate(iterator):
        response = int(row[-1]) # The response is the last value
        try:
            forest_type_count[forest_type[response]] +=1
            if response == 1:
                spruce += 1
            elif response == 2:
                lodgepole_pine +=1
            if n % 10000 == 0:
                proportions.append([spruce/float(n+1),\
                lodgepole_pine/float(n+1)])
        except:
            forest_type_count['Other'] += 1
    print ('Total rows: %i' % (n+1))
    print ('Frequency of classes:')
    for ftype, freq in sorted([(t,v) for t,v \
        in forest_type_count.iteritems()], key = \
        lambda x: x[1], reverse=True):
            print ("%-18s: %6i %04.1f%%" % \
                    (ftype, freq, freq*100/float(n+1)))

Out: Total rows: 581012
Frequency of classes:
Lodgepole Pine    : 283301 48.8%
Spruce/Fir        : 211840 36.5%
Ponderosa Pine    :  35754 06.2%
Krummholz         :  20510 03.5%
Douglas-fir       :  17367 03.0%
Aspen             :   9493 01.6%
```

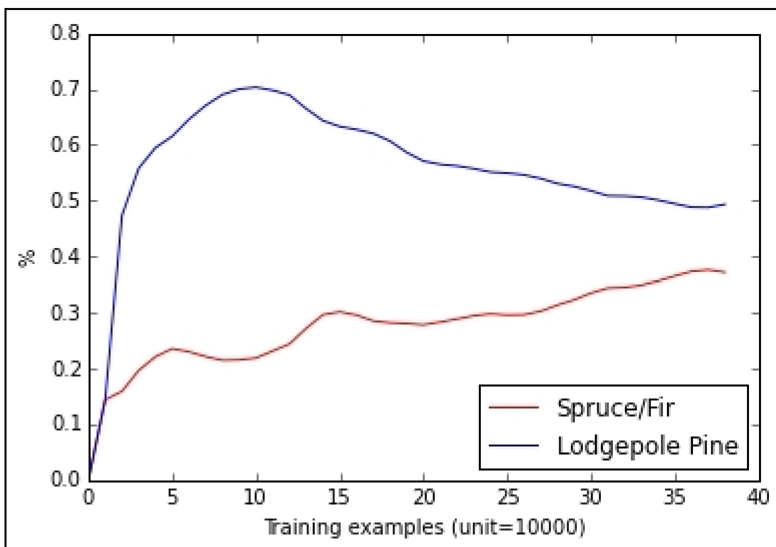
```
Cottonwood/Willow :   2747 00.5%
Other             :      0 00.0%
```

The output displays that two classes, `Lodgepole Pine` and `Spruce/Fir`, account for most observations. If examples are shuffled appropriately in the stream, the SGD will appropriately learn the correct a-priori distribution and consequently adjust its probability emission (a-posteriori probability).

If, contrary to our present case, your objective is not classification accuracy but increasing the **receiver operating characteristic** (**ROC**) **area under the curve** (**AUC**) or f1-score (error functions that can be used for evaluation; for an overview, you can directly consult the Scikit-learn documentation at http://scikit-learn.org/stable/modules/model_evaluation.html regarding a classification model trained on imbalanced data, and then the provided information can help you balance the weights using the `class_weight` parameter when defining `SGDClassifier` or `sample_weight` when partially fitting the model. Both change the impact of the observed instance by overweighting or underweighting it. In both ways, operating these two parameters will change the a-priori distribution. Weighting classes and instances will be discussed in the next chapter.

Before proceeding to training and working with classes, we can check whether the proportion of classes is always consistent in order to convey the correct a-priori probability to the SGD:

```
import matplotlib.pyplot as plt
import numpy as np
%matplotlib inline
proportions = np.array(proportions)
plt.plot(proportions[:,0],'r-', label='Spruce/Fir')
plt.plot(proportions[:,1],'b-', label='Lodgepole Pine')
plt.ylim(0.0,0.8)
plt.xlabel('Training examples (unit=10000)')
plt.ylabel('%')
plt.legend(loc='lower right', numpoints= 1)
plt.show()
```



In the previous figure, you can notice how the percentage of examples change as we progress streaming the data in the existing order. A shuffle is really necessary, in this case, if we want a stochastic online algorithm to learn correctly from data.

Actually, the proportions are changeable; this dataset has some kind of ordering, maybe a geographic one, that should be corrected by shuffling the data or we will risk overestimating or underestimating certain classes with respect to others.

The hashing trick

If, among your features, there are categories (encoded in values or left in textual form), things can get a bit trickier. Normally, in batch learning, you would one-hot encode the categories and get as many new binary features as categories that you have. Unfortunately, in a stream, you do not know in advance how many categories you will deal with, and not even by sampling can you be sure of their number because rare categories may appear really late in the stream or require a too large sample to be discovered. You will have to first stream all the data and take a record of every category that appears. Anyway, streams can be ephemeral and sometimes the number of classes can be so large that they cannot be stored in-memory. The online advertising data is such an example because of its high volumes that are difficult to store away and because the stream cannot be passed over more than once. Moreover, advertising data is quite varied and features change constantly.

Working with texts makes the problem even more strikingly evident because you cannot anticipate what kind of word could be part of the text that you will be analyzing. In a bag-of-words model—where for each text the present words are counted and their frequency value pasted in an element in the feature vector specific to each word—you should be able to map each word to an index in advance. Even if you can manage this, you'll always have to handle the situation when an unknown word (therefore never mapped before) will pop up during testing or when the predictor is in production. Marginally, it should also be added that, being a spoken language, dictionaries made of hundreds of thousands or even millions of different terms are not unusual at all.

To recap, if you can handle knowing in advance the classes in your features, you can deal with them using the one-hot encoder from Scikit-learn (http://Scikit-learn.org/stable/modules/generated/sklearn.preprocessing.OneHotEncoder.html). We actually won't illustrate it here but basically the approach is not at all different from what you would apply when working with batch learning. What we want to illustrate to you is when you cannot really apply one-hot encoding.

There is a solution called the hashing trick because it is based on the hash function and can deal with both text and categorical variables in integer or string form. It can also work with categorical variables mixed with numeric values from quantitative features. The core problem with one-hot encoding is that it assigns a position to a value in the feature vector after having mapped its feature to that position. The hashing trick can univocally map a value to its position without any prior need to evaluate the feature because it leverages the core characteristic of a hashing function—to transform a value or string into an integer value deterministically.

Therefore, the only necessary preparation before applying it is creating a sparse vector large enough to represent the complexity of the data (potentially containing from *2**19* to *2**30* elements, depending

on the available memory, bus architecture of your computer, and type of hash function that you are using). If you are working on some text, you'll also need a tokenizer, that is, a function that will split your text into single words and removes punctuation.

A simple toy example will make this clear. We will be using two specialized functions from the Scikit-learn package: `HashingVectorizer`, a transformer based on the hashing trick that works on textual data, and `FeatureHasher`, which is another transformer, specialized in converting a data row expressed as a Python dictionary to a sparse vector of features.

As the first example, we will turn a phrase into a vector:

```
In: from sklearn.feature_extraction.text import HashingVectorizer
h = HashingVectorizer(n_features=1000, binary=True, norm=None)
sparse_vector = h.transform(['A simple toy example will make clear
how it works.'])
print(sparse_vector)

Out:
  (0, 61)       1.0
  (0, 271)      1.0
  (0, 287)      1.0
  (0, 452)      1.0
  (0, 462)      1.0
  (0, 539)      1.0
  (0, 605)      1.0
  (0, 726)      1.0
  (0, 918)      1.0
```

The resulting vector has unit values only at certain indexes, pointing out an association between a token in the phrase (a word) and a certain position in the vector. Unfortunately, the association cannot be reversed unless we map the hash value for each token in an external Python dictionary. Though possible, such a mapping would be indeed memory consuming because dictionaries can prove large, in the range of millions of items or even more, depending on the language and topics. Actually, we do not need to keep such tracking because hash functions guarantee to always produce the same index from the same token.

A real problem with the hashing trick is the eventuality of a collision, which happens when two different tokens are associated to the same index. This is a rare but possible occurrence when working with large dictionaries of words. On the other hand, in a model composed of millions of coefficients, there are very few that are influential. Consequently, if a collision happens, probably it will involve two unimportant tokens. When using the hashing trick, probability is on your side because with a large enough output vector (for instance, the number of elements is above *2^24*), though collisions are always possible, it will be highly unlikely that they will involve important elements of the model.

The hashing trick can be applied to normal feature vectors, especially when there are categorical variables. Here is an example with `FeatureHasher`:

```
In: from sklearn.feature_extraction import FeatureHasher
h = FeatureHasher(n_features=1000, non_negative=True)
example_row = {'numeric feature':3, 'another numeric feature':2,
'Categorical feature = 3':1, 'f1*f2*f3':1*2*3}
print (example_row)

Out: {'another numeric feature': 2, 'f1*f2*f3': 6, 'numeric
feature': 3, 'Categorical feature = 3': 1}
```

If your Python dictionary contains the feature names for numeric values and a composition of feature name and value for any categorical variable, the dictionary's values will be mapped using the hashed index of the keys creating a one-hot encoded feature vector, ready to be learned by an SGD algorithm:

```
In: print (h.transform([example_row]))
Out:
  (0, 16)       2.0
  (0, 373)      1.0
  (0, 884)      6.0
  (0, 945)      3.0
```

Other basic transformations

As we have drawn the example from our data storage, apart from turning categorical features into numeric ones, another transformation can be applied in order to have the learning algorithm increase its predictive power. Transformations can be applied to features by a function (by applying a square root, logarithm, or other transformation function) or by operations on groups of features.

In the next chapter, we will propose detailed examples regarding polynomial expansion and random kitchen-sink methods. In the present chapter, we will anticipate how to create quadratic features by nested iterations. Quadratic features are usually created when creating polynomial expansions and their aim is to intercept how predictive features interact between them; this can influence the response in the target variable in an unexpected way.

As an example to intuitively clarify why quadratic features can matter in modeling a target response, let's explain the case of the effect of two medicines on a patient. In fact, it could be that each medicine is effective, more or less, against the disease we are fighting against. Anyway, the two medicines are made up of different components that, when ingested together by the patient, tend to nullify each other's effect. In such a case, though both medicines are effective, but together they do not work at all because of their negative interaction.

In such a sense, interactions between features can be found among a large variety of features, not just in medicine, and it is critical to find out the most significant one in order for our model to work better in predicting its target. If we are not aware that certain features interact with respect to our problem, our only choice is to systematically test them all and have our model retain the ones that work better.

In the following simple example, a vector named *v*, an example we imagine has been just streamed in-memory in order to be learned is transformed into another vector *vv* where the original features of *v* are

accompanied by the results of their multiplicative interactions (every feature is multiplied once by all the others). Given the larger number of features, the learning algorithm will be fed using the *vv* vector in place of the original *v* vector in order to achieve a better fit of the data:

```
In: import numpy as np
v = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
vv = np.hstack((v, [v[i]*v[j] for i in range(len(v)) for j in
range(i+1, len(v))]))
print vv

Out:[ 1  2  3  4  5  6  7  8  9 10  2  3  4  5  6  7  8  9 10  6  8
10 12 14 16 18 20 12 15 18 21 24 27 30 20 24 28 32 36 40 30 35 40 45
50 42 48 54 60 56 63 70 72 80 90]
```

Similar transformations, or even more complex ones, can be generated on the fly as the examples stream to the learning algorithm, exploiting the fact that the data batch is small (sometimes reduced to single examples) and expanding the number of features of so a few examples can feasibly be achieved in-memory. In the following chapter, we will explore more examples of such transformations and their successful integration into the learning pipeline.

Testing and validation in a stream

We have withheld showing full examples of training after introducing SGD because we need to introduce how to test and validate in a stream. Using batch learning, testing, and cross-validating is a matter of randomizing the order of the observations, slicing the dataset into folds and taking a precise fold as a test set, or systematically taking all the folds in turn to test your algorithm's learning capabilities.

Streams cannot be kept in-memory, so on the basis that the following instances are already randomized, the best solution is to take validation instances after the stream has unfolded for a while or systematically use a precise, replicable pattern in the data stream.

An out-of-sample approach on part of a stream is actually comparable to a test sample and can be successfully accomplished only knowing in advance the length of the stream. For continuous streams, it is still possible but implies stopping the learning definitely once the test instances start. This method is called a holdout after *n* strategy.

A cross-validation type of approach is possible using a systematic and replicable sampling of validation instances. After having defined a starting buffer, an instance is picked for validation every *n* times. Such an instance is not used for the training but for testing purposes. This method is called a periodic holdout strategy every *n* times.

As validation is done on a single instance base, a global performance measure is calculated, averaging all the error measures collected so far within the same pass over the data or in a window-like fashion, using the most recent set of *k* measures, where *k* is a number of tests that you think is validly representative.

As a matter of fact, during the first pass, all instances are actually unseen to the learning algorithm. It is therefore useful to test the algorithm as it receives cases to learn, verifying its response on an observation before learning it. This approach is called progressive validation.

Trying SGD in action

As a conclusion of the present chapter, we will implement two examples: one for classification based on the Forest Covertype data and one for regression based on the bike-sharing dataset. We will see how to put into practice the previous insights on response and feature distributions and how to use the best validation strategy for each problem.

Starting with the classification problem, there are two noticeable aspects to consider. Being a multiclass problem, first of all we noticed that there is some kind of ordering in the database and distribution of classes along the stream of instances. As an initial step, we will shuffle the data using the `ram_shuffle` function defined during the chapter in the *Paying attention to the ordering of instances* section:

```
In: import os
local_path = os.getcwd()
source = 'covtype.data'
ram_shuffle(filename_in=local_path+'\\'+source, \
            filename_out=local_path+'\\shuffled_covtype.data', \
            header=False)
```

As we are zipping the rows in-memory and shuffling them without much disk usage, we can quickly obtain a new working file. The following code will train `SGDClassifier` with log loss (equivalent to a logistic regression) so that it leverages our previous knowledge of the classes present in the dataset. The `forest_type` list contains all the codes of the classes and it is passed every time (though just one, the first, would suffice) to the `partial_fit` method of the SGD learner.

For validation purposes, we define a cold start at `200.000` observed cases. At every ten instances, one will be left out of training and used for validation. This schema allows reproducibility even if we are going to pass over the data multiple times; at every pass, the same instances will be left out as an out-of-sample test, allowing the creation of a validation curve to test the effect of multiple passes over the same data.

The holdout schema is accompanied by a progressive validation, too. So each case after the cold start is evaluated before being fed to the training. Although progressive validation provides an interesting feedback, such an approach will work only for the first pass; in fact after the initial pass, all the observations (but the ones in the holdout schema) will become in-sample instances. In our example, we are going to make only one pass.

As a reminder, the dataset has `581.012` instances and it may prove a bit long to stream and model with SGD (it is quite a large-scale problem for a single computer). Though we placed a limiter to observe just `250.000` instances, still allow your computer to run for about 15-20 minutes before expecting results:

```
In: import csv, time
import numpy as np
from sklearn.linear_model import SGDClassifier
source = 'shuffled_covtype.data'
SEP=','
forest_type = [t+1 for t in range(7)]
SGD = SGDClassifier(loss='log', penalty=None, random_state=1,
average=True)
accuracy = 0
holdout_count = 0
prog_accuracy = 0
prog_count = 0
cold_start = 200000
k_holdout = 10
with open(local_path+'\\'+source, 'rb') as R:
    iterator = csv.reader(R, delimiter=SEP)
    for n, row in enumerate(iterator):
        if n > 250000: # Reducing the running time of the experiment
            break
        # DATA PROCESSING
        response = np.array([int(row[-1])]) # The response is the
last value
        features = np.array(map(float,row[:-1])).reshape(1,-1)
        # MACHINE LEARNING
        if (n+1) >= cold_start and (n+1-cold_start) % k_holdout==0:
            if int(SGD.predict(features))==response[0]:
                accuracy += 1
            holdout_count += 1
            if (n+1-cold_start) % 25000 == 0 and (n+1) > cold_start:
                    print '%s holdout accuracy: %0.3f' %
(time.strftime('%X'), accuracy / float(holdout_count))
        else:
            # PROGRESSIVE VALIDATION
            if (n+1) >= cold_start:
                if int(SGD.predict(features))==response[0]:
                    prog_accuracy += 1
                prog_count += 1
                if n % 25000 == 0 and n > cold_start:
                    print '%s progressive accuracy: %0.3f' %
(time.strftime('%X'), prog_accuracy / float(prog_count))
            # LEARNING PHASE
            SGD.partial_fit(features, response, classes=forest_type)
print '%s FINAL holdout accuracy: %0.3f' % (time.strftime('%X'),
accuracy / ((n+1-cold_start) / float(k_holdout)))
print '%s FINAL progressive accuracy: %0.3f' % (time.strftime('%X'),
prog_accuracy / float(prog_count))
```

```
Out:
18:45:10 holdout accuracy: 0.627
18:45:10 progressive accuracy: 0.613
18:45:59 holdout accuracy: 0.621
18:45:59 progressive accuracy: 0.617
18:45:59 FINAL holdout accuracy: 0.621
18:45:59 FINAL progressive accuracy: 0.617
```

As the second example, we will try to predict the number of shared bicycles in Washington based on a series of weather and time information. Given the historical order of the dataset, we do not shuffle it and treat the problem as a time series one. Our validation strategy is to test the results after having seen a certain number of examples in order to replicate the uncertainities to forecast from that moment of time onward.

It is also interesting to notice that some of the features are categorical, so we applied the `FeatureHasher` class from Scikit-learn in order to represent having the categories recorded in a dictionary as a joint string made up of the variable name and category code. The value assigned in the dictionary for each of these keys is one in order to resemble a binary variable in the sparse vector that the hashing trick will be creating:

```
In: import csv, time, os
import numpy as np
from sklearn.linear_model import SGDRegressor
from sklearn.feature_extraction import FeatureHasher
source = '\\bikesharing\\hour.csv'
local_path = os.getcwd()
SEP=','
def apply_log(x): return np.log(float(x)+1)
def apply_exp(x): return np.exp(float(x))-1
SGD = SGDRegressor(loss='squared_loss', penalty=None,
random_state=1, average=True)
h = FeatureHasher(non_negative=True)
val_rmse = 0
val_rmsle = 0
predictions_start = 16000
with open(local_path+'\\'+source, 'rb') as R:
    iterator = csv.DictReader(R, delimiter=SEP)
    for n, row in enumerate(iterator):
        # DATA PROCESSING
        target = np.array([apply_log(row['cnt'])])
        features = {k+'_'+v:1 for k,v in row.iteritems() \
        if k in ['holiday','hr','mnth','season', \
           'weathersit','weekday','workingday','yr']}
        numeric_features = {k:float(v) for k,v in \
           row.iteritems() if k in ['hum', 'temp', '\
           atemp', 'windspeed']}
        features.update(numeric_features)
```

```
        hashed_features = h.transform([features])
        # MACHINE LEARNING
        if (n+1) >= predictions_start:
            # HOLDOUT AFTER N PHASE
            predicted = SGD.predict(hashed_features)
            val_rmse += (apply_exp(predicted) \
               - apply_exp(target))**2
            val_rmsle += (predicted - target)**2
            if (n-predictions_start+1) % 250 == 0 \
               and (n+1) > predictions_start:
                  print '%s holdout RMSE: %0.3f' \
                     % (time.strftime('%X'), (val_rmse \
                     / float(n-predictions_start+1))**0.5),
                print 'holdout RMSLE: %0.3f' % ((val_rmsle \
/ float(n-predictions_start+1))**0.5)
        else:
            # LEARNING PHASE
            SGD.partial_fit(hashed_features, target)

print '%s FINAL holdout RMSE: %0.3f' % \
(time.strftime('%X'), (val_rmse \
    / float(n-predictions_start+1))**0.5)
print '%s FINAL holdout RMSLE: %0.3f' % \
(time.strftime('%X'), (val_rmsle \
    / float(n-predictions_start+1))**0.5)

Out:
18:02:54 holdout RMSE: 281.065 holdout RMSLE: 1.899
18:02:54 holdout RMSE: 254.958 holdout RMSLE: 1.800
18:02:54 holdout RMSE: 255.456 holdout RMSLE: 1.798
18:52:54 holdout RMSE: 254.563 holdout RMSLE: 1.818
18:52:54 holdout RMSE: 239.740 holdout RMSLE: 1.737
18:52:54 FINAL holdout RMSE: 229.274
18:52:54 FINAL holdout RMSLE: 1.678
```

Summary

In this chapter, we have seen how learning is possible out-of-core by streaming data, no matter how big it is, from a text file or database on your hard disk. These methods certainly apply to much bigger datasets than the examples that we used to demonstrate them (which actually could be solved in-memory using non-average, powerful hardware).

We also explained the core algorithm that makes out-of-core learning possible—SGD—and we examined its strength and weakness, emphasizing the necessity of streams to be really stochastic (which means in a random order) to be really effective, unless the order is part of the learning objectives. In particular, we introduced the Scikit-learn implementation of SGD, limiting our focus to the linear and logistic regression loss functions.

Finally, we discussed data preparation, introduced the hashing trick and validation strategies for streams, and wrapped up the acquired knowledge on SGD fitting two different models—classification and regression.

In the next chapter, we will keep on enriching our out-of-core capabilities by figuring out how to enable non-linearity in our learning schema and hinge loss for support vector machines. We will also present alternatives to Scikit-learn, such as **Liblinear**, **Vowpal Wabbit**, and **StreamSVM**. Although operating as external shell commands, all of them could be easily wrapped and controlled by Python scripts.

Chapter 3. Fast SVM Implementations

Having experimented with online-style learning in the previous chapter, you may have been surprised by its simplicity yet effectiveness and scalability in comparison to batch learning. In spite of learning just one example at a time, SGD can approximate the results well as if all the data resides in the core memory and you were using a batch algorithm. All you need is that your stream be indeed stochastic (there are no trends in data) and that the learner is tuned well to the problem (the learning rate is often the key parameter to be fixed).

Anyway, examining such achievements closely, the results are still just comparable to batch linear models but not to learners that are more sophisticated and characterized by higher variance than bias, such as SVMs, neural networks, or bagging and boosting ensembles of decision trees.

For certain problems, such as tall and wide but sparse data, just linear combinations may be enough according to the observation that a simple algorithm with more data often wins over more complex ones trained on less data. Yet, even using linear models and by resorting to explicitly mapping existing features into higher-dimensionality ones (using different order of interactions, polynomial expansions, and kernel approximations), we can accelerate and improve the learning of complex nonlinear relationships between the response and features.

In this chapter, we will therefore first introduce linear SVMs as a machine learning algorithm alternative to linear models, powered by a different approach to the problem of learning from data. Then, we will demonstrate how we can create richer features from the existing ones in order to solve our machine learning tasks in a better way when facing large scale data, especially tall data (that is, datasets having many cases to learn from).

In summary, in this chapter, we will cover the following topics:

- Introducing SVMs and providing you with the basic concepts and math formulas to figure out how they work
- Proposing SGD with hinge loss as a viable solution for large scale tasks that uses the same optimization approach as the batch SVM
- Suggesting nonlinear approximations to accompany SGD
- Offering an overview of other large scale online solutions besides SGD algorithm made available by Scikit-learn

Datasets to experiment with on your own

As in the previous chapter, we will be using datasets from the UCI Machine Learning Repository, in particular the bike-sharing dataset (a regression problem) and Forest Covertype Data (a multiclass classification problem).

If you have not done so before or if you need to download both the datasets again, you will need a couple of functions defined in the *Datasets to try the real thing yourself* section of Chapter 2, *Scalable Learning in Scikit-learn*. The needed functions are `unzip_from_UCI` and `gzip_from_UCI`. Both have a Python connect to the UCI repository; download a compressed file and unzip it in the working

Python directory. If you call the functions from an IPython cell, you will find the necessary new directories and files exactly where IPython will look for them.

In case the functions do not work for you, never mind; we will provide you with the link for a direct download. After that, all you will have to do is unpack the data in the current working Python directory, which you can discover by running the following command on your Python interface (IPython or any IDE):

```
In: import os
print "Current directory is: \"%s\"" % (os.getcwd())

Out: Current directory is: "C:\scisoft\WinPython-64bit-2.7.9.4\
notebooks\Packt - Large Scale"
```

The bike-sharing dataset

The dataset comprises of two files in CSV format containing the hourly and daily count of bikes rented in the years between 2011 and 2012 within the Capital bike-share system in Washington D.C., USA. As a reminder, the data features the corresponding weather and seasonal information regarding the day of the rental.

The following code snippet will save the dataset on the local hard disk using the convenient `unzip_from_UCI` wrapper function:

```
In: UCI_url = 'https://archive.ics.uci.edu/ml/
machine-learning-databases/00275/Bike-Sharing-Dataset.zip'
unzip_from_UCI(UCI_url, dest='bikesharing')

Out: Extracting in C:\scisoft\WinPython-64bit-2.7.9.4\notebooks\
bikesharing
    unzipping day.csv
    unzipping hour.csv
```

If run successfully, the code will indicate in what directory the CSV files have been saved and print the names of both the unzipped files. If unsuccessful, just download the file from https://archive.ics.uci.edu/ml/machine-learning-databases/00275/Bike-Sharing-Dataset.zip and unzip the two files, `day.csv` and `hour.csv`, in a directory named `bikesharing` that you previously created in your Python working directory.

The covertype dataset

Donated by Jock A. Blackard, Dr. Denis J. Dean, Dr. Charles W. Anderson, and the Colorado State University, the covertype dataset contains 581,012 examples and a series of 54 cartographic variables, ranging from elevation to soil type, expected to be able to predict the forest cover type, which comprises seven kinds (so this is a multiclass problem). In order to assure comparability with academic studies on the same data, instructions recommend using the first 11,340 records for the training, the next 3,780 records for validation, and finally the remaining 565,892 records as test examples:

```
In: UCI_url = 'https://archive.ics.uci.edu/ml/
machine-learning-databases/covtype/covtype.data.gz'
gzip_from_UCI(UCI_url)
```

In case of problems in running the code or if you prefer to prepare the file by yourself, just go to the UCI website, download the dataset from https://archive.ics.uci.edu/ml/machine-learning-databases/covtype/covtype.data.gz, and unpack it into the directory that Python is currently working on.

Support Vector Machines

Support Vector Machines (**SVMs**) are a set of supervised learning techniques for classification and regression (and also for outlier detection), which is quite versatile as it can fit both linear and nonlinear models thanks to the availability of special functions—kernel functions. The specialty of such kernel functions is to be able to map the input features into a new, more complex feature vector using a limited amount of computations. Kernel functions nonlinearly recombine the original features, making possible the mapping of the response by very complex functions. In such a sense, SVMs are comparable to neural networks as universal approximators, and thus can boast a similar predictive power in many problems.

Contrary to the linear models seen in the previous chapter, SVMs started as a method to solve classification problems, not regression ones.

SVMs were invented at the AT&T laboratories in the '90s by the mathematician, Vladimir Vapnik, and computer scientist, Corinna Cortes (but there are also many other contributors that worked with Vapnik on the algorithm). In essence, an SVM strives to solve a classification problem by finding a particular hyperplane separating the classes in the feature space. Such a particular hyperplane has to be characterized as being the one with the largest separating margin between the boundaries of the classes (the margin is to be intended as the gap, the space between the classes themselves, empty of any example).

Such intuition implies two consequences:

- Empirically, SVMs try to minimize the test error by finding a solution in the training set that is exactly in the middle of the observed classes, thus the solution is clearly computational (it is an optimization based on quadratic programming—https://en.wikipedia.org/wiki/Quadratic_programming).
- As the solution is based on just the boundaries of the classes as set by the adjoining examples (called the support vectors), the other examples can be ignored, making the technique insensible to outliers and less memory-intensive than methods based on matrix inversion such as linear models.

Given such a general overview on the algorithm, we will be spending a few pages pointing out the key formulations that characterize SVMs. Although a complete and detailed explanation of the method is beyond the scope of this book, sketching how it works can indeed help figure out what is happening under the hood of the technique and provide the foundation to understand how it can be made scalable to big data.

Historically, SVMs were thought as *hard-margin* classifiers, just like the perceptron. In fact, SVMs were initially set trying to find two hyperplanes separating the classes whose reciprocal distance was the maximum possible. Such an approach worked perfectly with linearly-separable synthetic data. Anyway, in the hard-margin version, when an SVM faced nonlinearly separable data, it could only succeed using nonlinear transformations of the features. However, they were always deemed to fail when misclassification errors were due to noise in data instead of non-linearity.

For such a reason, softmargins were introduced by means of a cost function that took into account how serious the error was (as hard margins kept track only if an error occurred), thus allowing a certain

tolerance to misclassified cases whose error was not too big because they were placed next to the separating hyperplane.

Since the introduction of softmargins, SVMs also became able to withstand non-separability caused by noise. Softmargins were simply introduced by building the cost function around a slack variable that approximates the number of misclassified examples. Such a slack variable is also called the empirical risk (the risk of making a wrong classification as seen from the training data point of view).

In mathematical formulations, given a matrix dataset *X* of n examples and *m* features and a response vector expressing the belonging to a class in terms of `+1` (belonging) and `-1` (not belonging), a binary classification SVM strives to minimize the following cost function:

$$\frac{\lambda}{2}||w^2|| + \left[\frac{1}{n}\sum_{i=1}^{n} \max\left(0, 1 - y(wX + b)\right)\right]$$

In the preceding function, w is the vector of coefficients that expresses the separating hyperplane together with the bias *b*, representing the offset from the origin. There is also lambda (`λ>=0`), which is the regularization parameter.

For a better understanding of how the cost function works, it is necessary to divide it into two parts. The first part is the regularization term:

$$\frac{\lambda}{2}||w^2||$$

The regularization term is contrasting the minimization process when the vector w assumes high values. The second term is called the loss term or slack variable, and actually is the core of the SVM minimization procedure:

$$\frac{1}{n}\sum_{i=1}^{n} \max\left(0, 1 - y(wX + b)\right)$$

The loss term outputs an approximate value of misclassification errors. In fact, the summation will tend to add a unit value for each classification error, whose total divided by n, the number of examples, will provide an approximate proportion of the classification error.

Often, as in the Scikit-learn implementation, the lambda is removed from the regularization term and replaced by a misclassification parameter C multiplying the loss term:

$$\frac{1}{2}||w^2|| + C\left[\sum_{i=1}^{n} \max\left(0, 1 - y(wX + b)\right)\right]$$

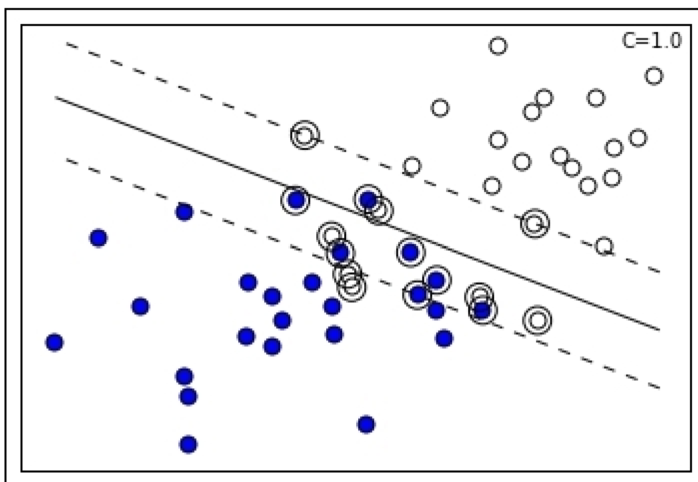
The relation between the previous parameter lambda and the new C is as follows:

$$\lambda = \frac{1}{nC}$$

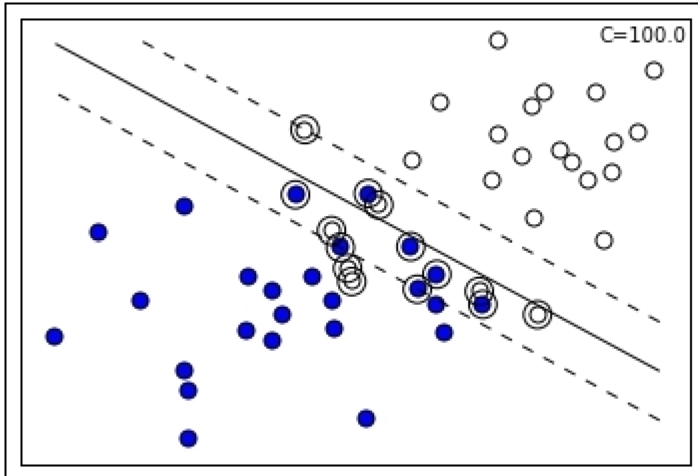
It is just a matter of conventions, as the change from the parameter lambda to C on the optimization's formula doesn't imply different results.

The impact of the loss term is mediated by a hyperparameter C. High values of C impose a high penalization on errors, thus forcing the SVM to try to correctly classify all the training examples. Consequently, larger C values tend to force the margin to be tighter and take into consideration fewer support vectors. Such a reduction of the margin translates into an increased bias and reduced variance.

This leads us to specify the role of certain observations with respect to others; in fact, we define as support vectors those examples that are misclassified or not classified with confidence as they are inside the margin (noisy observations that make class separation impossible). Optimization is possible taking into account only such examples, making SVM a memory-efficient technique indeed:

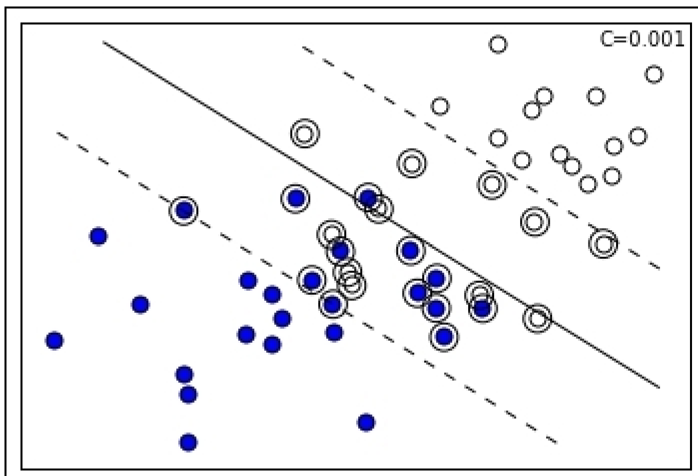


In the preceding visualization, you can notice the projection of two groups of points (blue and white) on two feature dimensions. An SVM solution with the C hyperparameter set to 1.0 can easily discover a separating line (in the plot represented as the continuous line), though there are some misclassified cases on both sides. In addition, the margin can be visualized (defined by the two dashed lines), being identifiable thanks to the support vectors of the respective class further from the separating line. In the chart, support vectors are marked by an external circle and you can actually notice that some support vectors are outside the margin; this is because they are misclassified cases and the SVM has to keep track of them for optimization purposes as their error is considered in the loss term:



Increasing the C value, the margin tends to restrict as the SVM is taking into account fewer support vectors in the optimization process. Consequently, the slope of the separating line also changes.

On the contrary, smaller C values tend to relax the margin, thus increasing the variance. Extremely small C values can even lead the SVM to consider all the example points inside the margin. Smaller C values are ideal when there are many noisy examples. Such a setting forces the SVM to ignore many misclassified examples in the definition of the margin (Errors are weighted less, so they are tolerated more when searching for the maximum margin.):



Continuing on the previous visual example, if we decrease the hyperparameter C, the margin actually expands because the number of the support vectors increases. Consequently, the margin being different, SVM resolves for a different separating line. There is no C value that can be deemed correct before being tested on data; the right value always has to be empirically found by cross-validation. By far, C is considered the most important hyperparameter in an SVM, to be set after deciding what kernel function to use.

Kernel functions instead map the original features into a higher-dimensional space by combining them in a nonlinear way. In such a way, apparently non-separable groups in the original feature space may turn separable in a higher-dimensional representation. Such a projection doesn't need too complex computations in spite of the fact that the process of explicitly transforming the original feature values into new ones can generate a potential explosion in the number of the features when projecting to high dimensionalities. Instead of doing such cumbersome computations, kernel functions can be simply plugged into the decision function, thus replacing the original dot product between the features and coefficient vector and obtaining the same optimization result as the explicit mapping would have had. (Such plugging is called the kernel trick because it is really a mathematical trick.)

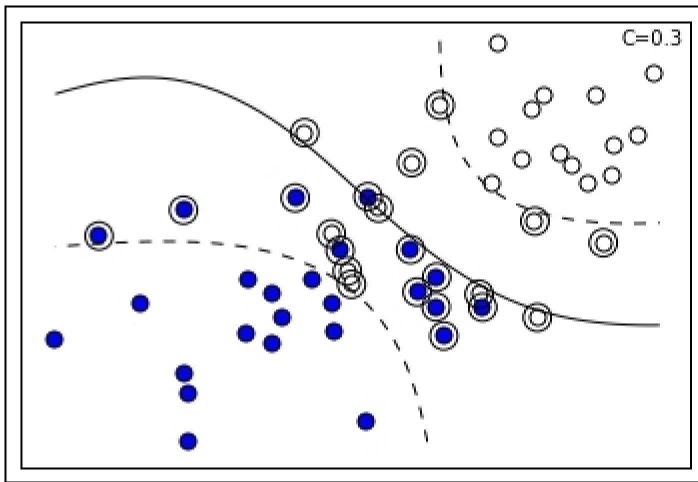
Standard kernel functions are linear functions (implying no transformations), polynomial functions, **radial basis functions** (**RBF**), and sigmoid functions. To provide an idea, the RBF function can be expressed as follows:

$$K(x_i, x) = \exp\left(-\|x_i - x\|^2 / 2\sigma\right)$$

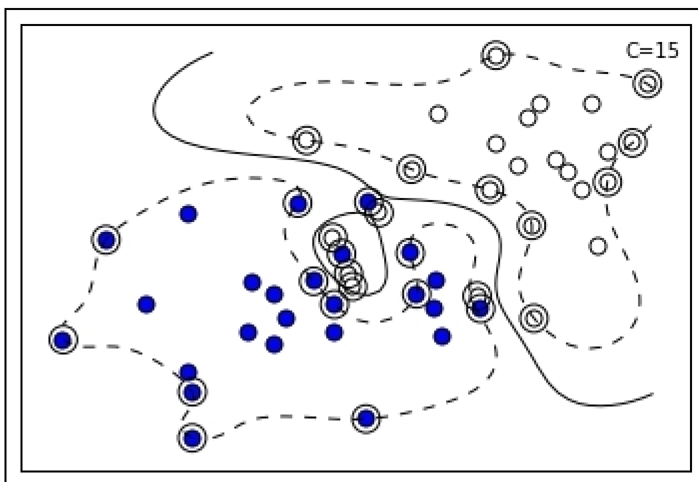
Basically, RBF and the other kernels just plug themselves directly into a variant of the previously seen function to be minimized. The previously seen optimization function is called the primal formulation whereas the analogous re-expression is called the dual formulation:

$$f(x) = \sum_{i=1}^{n} \alpha_i y_i K(x_i, x) + b$$

Though passing from the primal to the dual formulation is quite challenging without a mathematical demonstration, it is important to grasp that the kernel trick, given a kernel function that compares the examples by couples, is just a matter of a limited number of calculations with respect to the infinite dimensional feature space that it can unfold. Such a kernel trick renders the algorithm so particularly effective (comparable to neural networks) with respect to quite complex problems such as image recognition or textual classification:



For instance, the preceding SVM solution is possible thanks to a sigmoid kernel, whereas the following one is due to an RBF one:



As visually noticeable, the RBF kernel allows quite complex definitions of the margin, even splitting it into multiple parts (and an enclave is noticeable in the preceding example).

The formulation of the RBF kernel is as follows:

$$k(x_i, x_j) = \exp\left(-\gamma \|x_i - x_j\|^2\right)$$

Gamma is a hyperparameter for you to define a-priori. The kernel transformation creates some sort of classification bubbles around the support vectors, thus allowing the definition of very complex boundary shapes by merging the bubbles themselves.

The formulation of the sigmoid kernel is as follows:

$$k(x_i, x_j) = \tanh\left(\gamma \langle x_i x_j \rangle^2 + r\right)$$

Here, apart from gamma, r should be also chosen for the best result.

Clearly, solutions based on sigmoid, RBF, and polynomial, (Yes, it implicitly does the polynomial expansion that we will talk about in the following paragraphs.) kernels present more variance than the bias of the estimates, thus requiring a severe validation when deciding their adoption. Though an SVM is resistant to overfitting, it is not certainly immune to it.

Support vector regression is related to support vector classification. It varies just for the notation (more similar to a linear regression, using betas instead of a vector w of coefficients) and loss function:

$$\sum_{j=1}^{m} \beta_j^2 + C \sum_{i=1}^{n} \mathrm{L}_\epsilon \left(y_i - \hat{y}_i\right)$$

Noticeably, the only significant difference is the loss function L-epsilon, which is insensitive to errors (thus not computing them) if examples are within a certain distance epsilon from the regression hyperplane. The minimization of such a cost function optimizes the result for a regression problem, outputting values and not classes.

Hinge loss and its variants

As concluding remarks about the inner nuts and bolts of an SVM, remember that the cost function at the core of the algorithm is the hinge loss:

$$loss(y, \hat{y}) = \max\left(0, 1 - y\hat{y}\right)$$

As seen before, $\hat{y}$ is expressed as the summation of the dot product of X and coefficient vector w with the bias b:

$$\hat{y} = wX + b$$

Reminiscent of the *perceptron*, such a loss function penalizes errors linearly, expressing an error when the example is classified on the wrong side of the margin, proportional to its distance from the margin itself. Though convex, having the disadvantage of not being differentiable everywhere, it is sometimes replaced by always differentiable variants such as the squared hinge loss (also called L2 loss whereas L1 loss is the hinge loss):

$$L2_loss(y, \hat{y}) = \max(0, 1 - y\hat{y})^2$$

Another variant is the Huber loss, which is a quadratic function when the error is equal or below a certain threshold value h but a linear function otherwise. Such an approach mixes L1 and L2 variants of the hinge loss based on the error and it is an alternative quite resistant to outliers as larger error values are not squared, thus requiring fewer adjustments by the learning SVM. Huber loss is also an alternative to log loss (linear models) because it is faster to calculate and able to provide estimates of class probabilities (the hinge loss does not have such a capability).

From a practical point of view, there are no particular reports that Huber loss or L2 hinge loss can consistently perform better than hinge loss. In the end, the choice of a cost function just boils down to testing the available functions with respect to every different learning problem. (According to the principle of the no-free-lunch theorem, in machine learning, there are no solutions suitable for all the problems.)

Understanding the Scikit-learn SVM implementation

Scikit-learn offers an implementation of SVM using two C++ libraries (with a C API to interface with other languages) developed at the National Taiwan University, **A Library for Support Vector Machines** (**LIBSVM**) for SVM classification and regression (http://www.csie.ntu.edu.tw/~cjlin/libsvm/) and LIBLINEAR for classification problems using linear methods on large and sparse datasets (http://www.csie.ntu.edu.tw/~cjlin/liblinear/). Having both the libraries free to use, quite fast in computations, and already tested in quite a number of other solutions, all Scikit-learn implementations in the `sklearn.svm` module rely on one or the other, (The `Perceptron` and `LogisticRegression` classes also make use of them, by the way.) making Python just a convenient wrapper.

On the other hand, `SGDClassifier` and `SGDRegressor` use a different implementation as neither LIBSVM nor LIBLINEAR have an online implementation, both being batch learning tools. In fact, when operating, both LIBSVM and LIBLINEAR perform the best when allocated a suitable memory for kernel operations via the `cache_size` parameter.

The implementations for classification are as follows:

Class	Purpose	Hyperparameters
`sklearn.svm.SVC`	The LIBSVM implementation for binary and multiclass linear and kernel classification	C, kernel, degree, gamma
`sklearn.svm.NuSVC`	same as above	nu, kernel, degree, gamma
`sklearn.svm.OneClassSVM`	Unsupervised detection of outliers	nu, kernel, degree, gamma
`sklearn.svm.LinearSVC`	Based on LIBLINEAR, it is a binary and multiclass linear classifier	Penalty, loss, C

As for regression, the solutions are as follows:

Class	Purpose	Hyperparameters
`sklearn.svm.SVR`	The LIBSVM implementation for regression	C, kernel, degree, gamma, epsilon
`sklearn.svm.NuSVR`	same as above	nu, C, kernel, degree, gamma

As you can see, there are quite a few hyperparameters to be tuned for each version, making SVMs good learners when using default parameters and excellent ones when properly tuned by cross-validation, using `GridSearchCV` from the `grid_search` module in Scikit-learn.

As a golden rule, some parameters influence the result more and so should be fixed beforehand, others being dependent on their values. According to such an empirical rule, you have to correctly set the following parameters (ordered by rank of importance):

- `C`: This is the penalty value that we discussed before. Decreasing it makes the margin larger, thus ignoring more noise but also making for more computations. A best value can be normally looked for in the `np.logspace(-3, 3, 7)` range.
- `kernel`: This is the non-linearity workhorse because an SVM can be set to `linear`, `poly`, `rbf`, `sigmoid`, or a custom kernel (for experts!). The widely used one is certainly `rbf`.
- `degree`: This works with `kernel='poly'`, signaling the dimensionality of the polynomial expansion. It is ignored by other kernels. Usually, values from 2-5 work the best.

- `gamma`: This is a coefficient for `'rbf'`, `'poly'`, and `'sigmoid'`; high values tend to fit data in a better way. The suggested grid search range is `np.logspace(-3, 3, 7)`.
- `nu`: This is for regression and classification with `nuSVR` and `nuSVC`; this parameter approximates the training points that are not classified with confidence, misclassified points, and correct points inside or on the margin. It should be a number in the range [0,1] as it is a proportion relative to your training set. In the end, it acts as C with high proportions enlarging the margin.
- `epsilon`: This parameter specifies how much error an SVR is going to accept by defining an epsilon large range where no penalty is associated with respect to the true value of the point. The suggested search range is `np.insert(np.logspace(-4, 2, 7),0,[0])`.
- `penalty`, `loss` and `dual`: For LinearSVC, these parameters accept the `('l1','squared_hinge',False)`, `('l2','hinge',True)`, `('l2','squared_hinge',True)`, and `('l2','squared_hinge',False)` combinations. The `('l2','hinge',True)` combination is equivalent to the `SVC(kernel='linear')` learner.

As an example for basic classification and regression using SVC and SVR from Scikit-learn's `sklearn.svm` module, we will work with the Iris and Boston datasets, a couple of popular toy datasets (http://scikit-learn.org/stable/datasets/).

First, we will load the Iris dataset:

```
In: from sklearn import datasets
iris = datasets.load_iris()
X_i, y_i = iris.data, iris.target
```

Then, we will fit an SVC with an RBF kernel (C and gamma were chosen on the basis of other known examples in Scikit-learn) and test the results using the `cross_val_score` function:

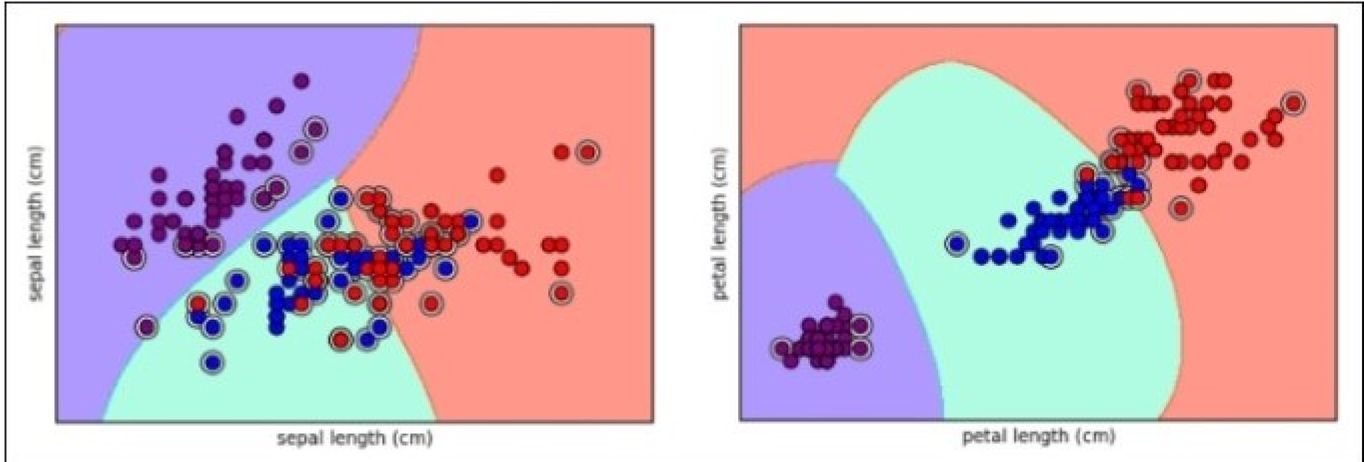
```
from sklearn.svm import SVC
from sklearn.cross_validation import cross_val_score
import numpy as np
h_class = SVC(kernel='rbf', C=1.0, gamma=0.7, random_state=101)
scores = cross_val_score(h_class, X_i, y_i, cv=20,
scoring='accuracy')
print 'Accuracy: %0.3f' % np.mean(scores)

Output: Accuracy: 0.969
```

The fitted model can provide you with an index pointing out what are the support vectors among your training examples:

```
In: h_class.fit(X_i,y_i)
print h_class.support_

Out: [ 13  14  15  22  24  41  44  50  52  56  60  62  63  66  68
```

```
70  72  76  77  83  84  85  98 100 106 110 114 117 118 119 121 123
126 127 129 131 133 134 138 141 146 149]
```

Here is a graphical representation of the support vectors selected by the SVC for the Iris dataset, represented with color decision boundaries (we tested a discrete grid of values in order to be able to project for each area of the chart what class the model will predict):



Tip

If you are interested in replicating the same charts, you can have a look and tweak this code snippet from http://scikit-learn.org/stable/auto_examples/svm/plot_iris.html.

To test an SVM regressor, we decided to try SVR with the Boston dataset. First, we upload the dataset in the core memory and then we randomize the ordering of examples as, noticeably, such a dataset is actually ordered in a subtle fashion, thus making results from not order-randomized cross-validation invalid:

```
In: import numpy as np
from sklearn.datasets import load_boston
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
boston = load_boston()
shuffled = np.random.permutation(boston.target.size)
X_b = scaler.fit_transform(boston.data[shuffled,:])
y_b = boston.target[shuffled]
```

Tip

Due to the fact that we used the `permutation` function from the `random` module in the NumPy package, you may obtain a differently shuffled dataset and consequently a slightly different cross-validated score from the following test. Moreover, the features having different scales, it is a good

practice to standardize the features so that they will have zero-centered mean and unit variance. Especially when using SVM with kernels, standardization is indeed crucial.

Finally, we can fit the SVR model (we decided on some C, gamma, and epsilon parameters that we know work fine) and, using cross-validation, we evaluate it by the root mean squared error:

```
In: from sklearn.svm import SVR
from sklearn.cross_validation import cross_val_score
h_regr = SVR(kernel='rbf', C=20.0, gamma=0.001, epsilon=1.0)
scores = cross_val_score(h_regr, X_b, y_b, cv=20,
scoring='mean_squared_error')
print 'Mean Squared Error: %0.3f' % abs(np.mean(scores))

Out: Mean Squared Error: 28.087
```

Pursuing nonlinear SVMs by subsampling

SVMs have quite a few advantages over other machine learning algorithms:

- They can handle majority of the supervised problems such as regression, classification, and anomaly detection, though they are actually best at binary classification.
- They provide a good handling of noisy data and outliers and they tend to overfit less as they only work with support vectors.
- They work fine with wide datasets (more features than examples); though, as with other machine learning algorithms, an SVM would gain both from dimensionality reduction and feature selection.

As drawbacks, we have to mention the following:

- They provide just estimates, but no probabilities unless you run some time-consuming and computationally-intensive probability calibration by means of Platt scaling
- They scale super-linearly with the number of examples

In particular, the last drawback puts a strong limitation on the usage of SVMs for large datasets. The optimization algorithm at the core of this learning technique—quadratic programming—scales in the Scikit-learn implementation between *O(number of feature * number of samples^2)* and *O(number of feature * number of samples^3)*, a complexity that seriously limits the operability of the algorithm to datasets that are under `10^4` number of cases.

Again, as seen in the last chapter, there are just a few options when you give a batch algorithm and too much data: subsampling, parallelization, and out-of-core learning by streaming. Subsampling and parallelization are seldom quoted as the best solutions, streaming being the favored approach to implement SVMs with large-scale problems.

However, though less used, subsampling is quite easy to implement leveraging reservoir sampling, which can produce random samples rapidly from streams derived from datasets and infinite online streams. By subsampling, you can produce multiple SVM models, whose results can be averaged for better results. The predictions from multiple SVM models can even be stacked, thus creating a new

dataset, and used to build a new model fusing the predictive capabilities of all, which will be described in Chapter 6, *Classification and Regression Trees at Scale*.

Reservoir sampling is an algorithm for randomly choosing a sample from a stream without having prior knowledge of how long the stream is. In fact, every observation in the stream has the same probability of being chosen. Initialized with a sample taken from the first observations in the stream, each element in the sample can be replaced at any moment by the example in the stream according to a probability proportional to the number of elements streamed up so far. So, for instance, when the *i-th* element of the stream arrives, it has a probability of being inserted in place of a random element in the sample. Such an insertion probability is equivalent to the sample dimension divided by *i*; therefore, it is progressively decreasing with respect to the stream length. If the stream is infinite, stopping at any time assures that the sample is representative of the elements seen so far.

In our example, we draw two random, mutually exclusive samples from the stream—one to train and one to test. We will extract such samples from the Covertype database, using the original ordered file. (As we will stream all the data before taking the sample, the random sampling won't be affected by the ordering.) We decided on a training sample of 5,000 examples, a number that should scale well on most desktop computers. As for the test set, we will be using 20,000 examples:

```
In: from random import seed, randint
SAMPLE_COUNT = 5000
TEST_COUNT   = 20000
seed(0) # allows repeatable results
sample = list()
test_sample = list()
for index, line in enumerate(open('covtype.data','rb')):
    if index < SAMPLE_COUNT:
        sample.append(line)
    else:
        r = randint(0, index)
        if r < SAMPLE_COUNT:
            sample[r] = line
        else:
            k = randint(0, index)
            if k < TEST_COUNT:
                if len(test_sample) < TEST_COUNT:
                    test_sample.append(line)
                else:
                    test_sample[k] = line
```

The algorithm should be streaming quite fast on the over `500,000` rows of the data matrix. In fact, we really did no preprocessing during the streaming in order to keep it the fastest possible. We consequently now need to transform the data into a NumPy array and standardize the features:

```
In: import numpy as np
from sklearn.preprocessing import StandardScaler
for n,line in enumerate(sample):
```

```
        sample[n] = map(float,line.strip().split(','))
y = np.array(sample)[:,-1]
scaling = StandardScaler()
X = scaling.fit_transform(np.array(sample)[:,:-1])
```

Once done with the training data *X*, *y*, we have to process the test data in the same way; particularly, we will have to standardize the features using standardization parameters (means and standard deviations) as in the training sample:

```
In: for n,line in enumerate(test_sample):
        test_sample[n] = map(float,line.strip().split(','))
yt = np.array(test_sample)[:,-1]
Xt = scaling.transform(np.array(test_sample)[:,:-1])
```

When both train and test sets are ready, we can fit the SVC model and predict the results:

```
In: from sklearn.svm import SVC
h = SVC(kernel='rbf', C=250.0, gamma=0.0025, random_state=101)
h.fit(X,y)
prediction = h.predict(Xt)
from sklearn.metrics import accuracy_score
print accuracy_score(yt, prediction)

Out: 0.75205
```

Achieving SVM at scale with SGD

Given the limitations of subsampling (first of all, the underfitting with respect to models trained on larger datasets), the only available option when using Scikit-learn for linear SVMs applied to large-scale streams remains the `SGDClassifier` and `SGDRegressor` methods, both available in the `linear_model` module. Let's see how to use them at their best and improve our results on the example datasets.

We are going to leverage the previous examples seen in the chapter for linear and logistic regression and transform them into an efficient SVM. As for classification, it is required that you set the loss type using the `loss` hyperparameter. The possible values for the parameter are `'hinge'`, `'squared_hinge'`, and `'modified_huber'`. All such loss functions were previously introduced and discussed in this same chapter when dealing with the SVM formulation.

All of them imply applying a soft-margin linear SVM (no kernels), thus resulting in an SVM resistant to misclassifications and noisy data. However, you may also try to use the loss `'perceptron'`, a type of loss that results in a hinge loss without margin, a solution suitable when it is necessary to resort to a model with more bias than the other possible loss choices.

Two aspects have to be taken into account to gain the best results when using such a range of hinge loss functions:

- When using any loss function, the stochastic gradient descent turns lazy, updating the vector of coefficients only when an example violates the previously defined margins. This is quite contrary to the loss function in the log or squared error, when actually every example is considered for the update of the coefficients' vector. In case there are many features involved in the learning, such a lazy approach leads to a resulting sparser vector of coefficients, thus reducing the overfitting. (A denser vector implies more overfitting because some coefficients are likely to catch more noise than signals from the data.)
- Only the `'modified_huber'` loss allows probability estimation, making it a viable alternative to the log loss (as found in the stochastic logistic regression). A modified Huber is also a better performer when dealing with multiclass **one-vs-all** (**OVA**) predictions as the probability outputs of the multiple models are better than the standard decision functions characteristic of hinge loss (probabilities work better than the raw output of the decision functions as they are on the same scale, bounded from 0 to 1). This loss function works by deriving a probability estimate directly from the decision function: `(clip(decision_function(X), -1, 1) + 1) / 2.`

As for regression problems, `SGDRegressor` offers two SVM loss options:

`'epsilon_insensitive'`

`'squared_epsilon_insensitive'`

Both activate a linear support vector regression, where errors (residual from the prediction) within the epsilon value are ignored. Past the epsilon value, the `epsilon_insensitive` loss considers the error as it is. The `squared_epsilon_insensitive` loss operates in a similar way though the error here is more penalizing as it is squared, with larger errors influencing the model building more.

In both cases, setting the correct epsilon hyperparameter is critical. As a default value, Scikit-learn suggests epsilon=0.1, but the best value for your problem has to be found by means of grid-search supported by cross-validation, as we will see in the next paragraphs.

Note that among regression losses, there is also a `'huber'` loss available that does not activate an SVM kind of optimization but just modifies the usual `'squared_loss'` to be insensible to outliers by switching from squared to linear loss past a distance of the value of the epsilon parameter.

As for our examples, we will be repeating the streaming process a certain number of times in order to demonstrate how to set different hyperparameters and transform features; we will use some convenient functions in order to reduce the number of repeated lines of code. Moreover, in order to speed up the execution of the examples, we will put a limit to the number of cases or tolerance values that the algorithm refers to. In such a way, both the training and validation times are kept at a minimum and no example will require you to wait more minutes than the time for a cup of tea or coffee.

As for the convenient wrapper functions, the first one will have the purpose to initially stream part or all the data once (we set a limit using the `max_rows` parameter). After completing the streaming, the function will be able to figure out all the categorical features' levels and record the different ranges of the numeric features. As a reminder, recording ranges is an important aspect to take care of. Both SGD

and SVM are algorithms sensible to different range scales and they perform worse when working with a number outside the [-1,1] range.

As an output, our function will return two trained Scikit-learn objects: `DictVectorizer` (able to transform feature ranges present in a dictionary into a feature vector) and `MinMaxScaler` to rescale numeric variables in the [0,1] range (useful to keep values sparse in the dataset, thus keeping memory usage low and achieving fast computations when most values are zero). As a unique constraint, it is necessary for you to know the feature names of the numeric and categorical variables that you want to use for your predictive model. Features not enclosed in the lists' feed to the `binary_features` or `numeric_features` parameters actually will be ignored. When the stream has no features' names, it is necessary for you to name them using the `fieldnames` parameter:

```
In: import csv, time, os
import numpy as np
from sklearn.linear_model import SGDRegressor
from sklearn.feature_extraction import DictVectorizer
from sklearn.preprocessing import MinMaxScaler
from scipy.sparse import csr_matrix

def explore(target_file, separator=',', fieldnames= None,
binary_features=list(), numeric_features=list(), max_rows=20000):
    """
    Generate from an online style stream a DictVectorizer and a
MinMaxScaler.

    Parameters
----------
    target file = the file to stream from
    separator = the field separator character
    fieldnames = the fields' labels (can be omitted and read from
file)
    binary_features = the list of qualitative features to consider
    numeric_features = the list of numeric futures to consider
    max_rows = the number of rows to be read from the stream (can be
None)
    """
    features = dict()
    min_max  = dict()
    vectorizer = DictVectorizer(sparse=False)
    scaler = MinMaxScaler()
    with open(target_file, 'rb') as R:
        iterator = csv.DictReader(R, fieldnames, delimiter=separator)
        for n, row in enumerate(iterator):
            # DATA EXPLORATION
            for k,v in row.iteritems():
                if k in binary_features:
                    if k+'_'+v not in features:
```

```
                        features[k+'_'+v]=0
                elif k in numeric_features:
                    v = float(v)
                    if k not in features:
                        features[k]=0
                        min_max[k] = [v,v]
                    else:
                        if v < min_max[k][0]:
                            min_max[k][0]= v
                        elif v > min_max[k][1]:
                            min_max[k][1]= v
                else:
                    pass # ignore the feature
            if max_rows and n > max_rows:
                break
    vectorizer.fit([features])
    A = vectorizer.transform([{f:0 if f not in min_max else
min_max[f][0] for f in vectorizer.feature_names_},
{f:1 if f not in min_max else min_max[f][1] for f in
vectorizer.feature_names_}])
    scaler.fit(A)
    return vectorizer, scaler
```

Tip

This code snippet can be reused easily for your own machine learning applications for large-scale data. In case your stream is an online one (a continuous streaming) or a too long one, you can apply a different limit on the number of observed examples by setting the `max_rows` parameter.

The second function will instead just pull out the data from the stream and transform it into a feature vector, normalizing the numeric features if a suitable `MinMaxScaler` object is provided instead of a `None` setting:

```
In: def pull_examples(target_file, vectorizer, binary_features,
numeric_features, target, min_max=None, separator=',',
fieldnames=None, sparse=True):
    """
    Reads a online style stream and returns a generator of
normalized feature vectors

    Parameters
----------
    target file = the file to stream from
    vectorizer = a DictVectorizer object
    binary_features = the list of qualitative features to consider
    numeric_features = the list of numeric features to consider
    target = the label of the response variable
```

```
    min_max = a MinMaxScaler object, can be omitted leaving None
    separator = the field separator character
    fieldnames = the fields' labels (can be omitted and read from
file)
    sparse = if a sparse vector is to be returned from the generator
    """
    with open(target_file, 'rb') as R:
        iterator = csv.DictReader(R, fieldnames, delimiter=separator)
        for n, row in enumerate(iterator):
            # DATA PROCESSING
            stream_row = {}
            response = np.array([float(row[target])])
            for k,v in row.iteritems():
                if k in binary_features:
                    stream_row[k+'_'+v]=1.0
                else:
                    if k in numeric_features:
                        stream_row[k]=float(v)
            if min_max:
                features =
min_max.transform(vectorizer.transform([stream_row]))
            else:
                features = vectorizer.transform([stream_row])
            if sparse:
                yield(csr_matrix(features), response, n)
            else:
                yield(features, response, n)
```

Given these two functions, now let's try again to model the first regression problem seen in the previous chapter, the bike-sharing dataset, but using a hinge loss this time instead of the mean squared errors that we used before.

As the first step, we provide the name of the file to stream and a list of qualitative and numeric variables (as derived from the header of the file and initial exploration of the file). The code of our wrapper function will return some information on the hot-coded variables and value ranges. In this case, most of the variables will be binary ones, a perfect situation for a sparse representation, as most values in our dataset are plain zero:

```
In: source = '\\bikesharing\\hour.csv'
local_path = os.getcwd()
b_vars = ['holiday','hr','mnth',
'season','weathersit','weekday','workingday','yr']
n_vars = ['hum', 'temp', 'atemp', 'windspeed']
std_row, min_max = explore(target_file=local_path+'\\'+source,
binary_features=b_vars, numeric_features=n_vars)
print 'Features: '
for f,mv,mx in zip(std_row.feature_names_, min_max.data_min_,
```

```
min_max.data_max_):
    print '%s:[%0.2f,%0.2f] ' % (f,mv,mx)

Out:
Features:
atemp:[0.00,1.00]
holiday_0:[0.00,1.00]
holiday_1:[0.00,1.00]
...
workingday_1:[0.00,1.00]
yr_0:[0.00,1.00]
yr_1:[0.00,1.00]
```

As you can notice from the output, qualitative variables have been encoded using their variable name and adding, after an underscore character, their value and transformed into binary features (which has the value of one when the feature is present, otherwise it is set to zero). Note that we are always using our SGD models with the `average=True` parameter in order to assure a faster convergence (this corresponds to using the **Averaged Stochastic Gradient Descent** (**ASGD**) model as discussed in the previous chapter.):

```
In:from sklearn.linear_model import SGDRegressor
SGD = SGDRegressor(loss='epsilon_insensitive', epsilon=0.001,
penalty=None, random_state=1, average=True)
val_rmse = 0
val_rmsle = 0
predictions_start = 16000

def apply_log(x): return np.log(x + 1.0)
def apply_exp(x): return np.exp(x) - 1.0

for x,y,n in pull_examples(target_file=local_path+'\\'+source,
                           vectorizer=std_row, min_max=min_max,
                           binary_features=b_vars,
numeric_features=n_vars, target='cnt'):
    y_log = apply_log(y)
# MACHINE LEARNING
    if (n+1) >= predictions_start:
        # HOLDOUT AFTER N PHASE
        predicted = SGD.predict(x)
        val_rmse += (apply_exp(predicted) - y)**2
        val_rmsle += (predicted - y_log)**2
        if (n-predictions_start+1) % 250 == 0 and (n+1) >
predictions_start:
            print n,
            print '%s holdout RMSE: %0.3f' % (time.strftime('%X'),
(val_rmse / float(n-predictions_start+1))**0.5),
            print 'holdout RMSLE: %0.3f' % ((val_rmsle /
```

```
float(n-predictions_start+1))**0.5)
    else:
        # LEARNING PHASE
        SGD.partial_fit(x, y_log)
print '%s FINAL holdout RMSE: %0.3f' % (time.strftime('%X'),
(val_rmse / float(n-predictions_start+1))**0.5)
print '%s FINAL holdout RMSLE: %0.3f' % (time.strftime('%X'),
(val_rmsle / float(n-predictions_start+1))**0.5)

Out:
16249 07:49:09 holdout RMSE: 276.768 holdout RMSLE: 1.801
16499 07:49:09 holdout RMSE: 250.549 holdout RMSLE: 1.709
16749 07:49:09 holdout RMSE: 250.720 holdout RMSLE: 1.696
16999 07:49:09 holdout RMSE: 249.661 holdout RMSLE: 1.705
17249 07:49:09 holdout RMSE: 234.958 holdout RMSLE: 1.642
07:49:09 FINAL holdout RMSE: 224.513
07:49:09 FINAL holdout RMSLE: 1.596
```

We are now going to try the classification problem of forest covertype:

```
In: source = 'shuffled_covtype.data'
local_path = os.getcwd()
n_vars = ['var_'+'0'*int(j<10)+str(j) for j in range(54)]
std_row, min_max = explore(target_file=local_path+'\\'+source,
binary_features=list(),
                  fieldnames= n_vars+['covertype'],
numeric_features=n_vars, max_rows=50000)
print 'Features: '
for f,mv,mx in zip(std_row.feature_names_, min_max.data_min_,
min_max.data_max_):
    print '%s:[%0.2f,%0.2f] ' % (f,mv,mx)

Out:
Features:
var_00:[1871.00,3853.00]
var_01:[0.00,360.00]
var_02:[0.00,61.00]
var_03:[0.00,1397.00]
var_04:[-164.00,588.00]
var_05:[0.00,7116.00]
var_06:[58.00,254.00]
var_07:[0.00,254.00]
var_08:[0.00,254.00]
var_09:[0.00,7168.00]
...
```

After having sampled from the stream and fitted our `DictVectorizer` and `MinMaxScaler` objects, we can start our learning process using a progressive validation this time (the error measure is given by testing the model on the cases before they are used for the training), given the large number of examples available. Every certain number of examples as set by the `sample` variable in the code, the script reports the situation with average accuracy from the most recent examples:

```
In: from sklearn.linear_model import SGDClassifier
SGD = SGDClassifier(loss='hinge', penalty=None, random_state=1,
average=True)
accuracy = 0
accuracy_record = list()
predictions_start = 50
sample = 5000
early_stop = 50000
for x,y,n in pull_examples(target_file=local_path+'\\'+source,
                           vectorizer=std_row,
                           min_max=min_max,
                           binary_features=list(),
numeric_features=n_vars,
                           fieldnames= n_vars+['covertype'],
target='covertype'):
    # LEARNING PHASE
    if n > predictions_start:
        accuracy += int(int(SGD.predict(x))==y[0])
        if n % sample == 0:
            accuracy_record.append(accuracy / float(sample))
            print '%s Progressive accuracy at example %i: %0.3f' %
(time.strftime('%X'), n, np.mean(accuracy_record[-sample:]))
            accuracy = 0
    if early_stop and n >= early_stop:
            break
    SGD.partial_fit(x, y, classes=range(1,8))

Out: ...
19:23:49 Progressive accuracy at example 50000: 0.699
```

Tip

Having to process over 575,000 examples, we set an early stop to the learning process after 50,000. You are free to modify such parameters according to the power of your computer and time availability. Be warned that the code may take some time. We experienced about 30 minutes of computing on an Intel Core i3 processor clocking at 2.20 GHz.

Feature selection by regularization

In a batch context, it is common to operate feature selection by the following:

- A preliminary filtering based on completeness (incidence of missing values), variance, and high multicollinearity between variables in order to have a cleaner dataset of relevant and operable features.
- Another initial filtering based on the univariate association (chi-squared test, F-value, and simple linear regression) between the features and response variable in order to immediately remove the features that are of no use for the predictive task because they are little or not related to the response.
- During modeling, a recursive approach inserting and/or excluding features on the basis of their capability to improve the predictive power of the algorithm, as tested on a holdout sample. Using a smaller subset of just relevant features allows the machine learning algorithm to be less affected by overfitting because of noisy variables and the parameters in excess due to the high dimensionality of the features.

Applying such approaches in an online setting is certainly still possible, but quite expensive in terms of the required time because of the quantity of data to stream to complete a single model. Recursive approaches based on a large number of iterations and tests require a nimble dataset that can fit in memory. As just previously quoted, in such a case, subsampling would be a good option in order to figure out features and models later to be applied to a larger scale.

Keeping on our out-of-core approach, regularization is the ideal solution as a way to select variables while streaming and filter out noisy or redundant features. Regularization works fine with online algorithms as it operates as the online machine learning algorithm is working and fitting its coefficients from the examples, without any need to run other streams for the purpose of selection. Regularization is, in fact, just a penalty value, which is added to the optimization of the learning process. It is dependent on the features' coefficient and a parameter named `alpha` setting the impact of regularization. The regularization balancing intervenes when coefficients' weights are updated by the model. At that time, regularization acts by reducing the resulting weights if the value of the update is not large enough. The trick of excluding or attenuating redundant variables is achieved because of the regularization `alpha` parameter, which has to be empirically set at the correct magnitude for the best result with respect to each specific data to be learned.

SGD implements the same regularization strategies to be found in batch algorithms:

- L1 penalty pushing to zero redundant and not so important variables
- L2 reducing the weight of less important features
- Elastic Net mixing the effects of L1 and L2 regularization

L1 regularization is the perfect strategy when there are unusual and redundant variables as it will push the coefficients of such features to zero, making them irrelevant when calculating the prediction.

L2 is suitable when there are many correlations between the variables as its strategy is just to reduce the weights of the features whose variation is less important for the loss function minimization. With L2, all the variables keep on contributing to the prediction, though some less so.

Elastic Net mixes L1 and L2 using a weighted sum. This solution is interesting as sometimes L1 regularization is unstable when dealing with highly correlated variables, choosing one or the other with respect to the seen examples. Using `ElasticNet`, many unusual features will still be pushed to zero as in L1 regularization, but correlated ones will be attenuated as in L2.

Both `SGDClassifier` and `SGDRegressor` can implement L1, L2, and Elastic Net regularization using the `penalty`, `alpha`, and `l1_ratio` parameters.

Tip

The alpha parameter is the most critical parameter after deciding what kind of penalty or about the mix of the two. Ideally, you can test suitable values in the range from `0.1` to `10^-7`, using the list of values produced by `10.0**-np.arange(1,7)`.

If `penalty` determinates what kind of regularization is chosen, `alpha`, as mentioned, will determinate its strength. As `alpha` is a constant that multiplies the penalization term; low alpha values will bring little influence on the final coefficient, whereas high values will significantly affect it. Finally, `l1_ratio` represents, when `penalty='elasticnet'`, how much percentage is the L1 penalization with respect to L2.

Setting regularization with SGD is very easy. For instance, you may try changing the previous code example inserting a penalty L2 into `SGDClassifier`:

```
SGD = SGDClassifier(loss='hinge', penalty='l2', alpha= 0.0001,
random_state=1, average=True)
```

If you prefer to test an Elastic-Net mixing the effects of the two regularization approaches, all you have to do is explicit the ratio between L1 and L2 by setting `l1_ratio`:

```
SGD = SGDClassifier(loss=''hinge'', penalty=''elasticnet'', \ alpha=
0.001, l1_ratio=0.5, random_state=1, average=True)
```

As the success of regularization depends on plugging the right kind of penalty and best alpha, regularization will be seen in action in our examples when dealing with the problem of hyperparameter optimization.

Including non-linearity in SGD

The fastest way to insert non-linearity into a linear SGD learner (and basically a no-brainer) is to transform the vector of the example received from the stream into a new vector including both power transformations and a combination of the features upto a certain degree.

Combinations can represent interactions between the features (explicating when two features concur to have a special impact on the response), thus helping the SVM linear model to include a certain amount of non-linearity. For instance, a two-way interaction is made by the multiplication of two features. A three-way is made by multiplying three features and so on, creating even more complex interactions for higher-degree expansions.

In Scikit-learn, the preprocessing module contains the `PolynomialFeatures` class, which can automatically transform the vector of features by polynomial expansion of the desired degree:

```
In: from sklearn.linear_model import SGDRegressor
from  sklearn.preprocessing import PolynomialFeatures

source = '\\bikesharing\\hour.csv'
local_path = os.getcwd()
b_vars = ['holiday','hr','mnth',
'season','weathersit','weekday','workingday','yr']
n_vars = ['hum', 'temp', 'atemp', 'windspeed']
std_row, min_max = explore(target_file=local_path+'\\'+source,
binary_features=b_vars, numeric_features=n_vars)

poly = PolynomialFeatures(degree=2, interaction_only=False,
include_bias=False)
SGD = SGDRegressor(loss='epsilon_insensitive', epsilon=0.001,
penalty=None, random_state=1, average=True)

val_rmse = 0
val_rmsle = 0
predictions_start = 16000

def apply_log(x): return np.log(x + 1.0)
def apply_exp(x): return np.exp(x) - 1.0

for x,y,n in pull_examples(target_file=local_path+'\\'\
+source,vectorizer=std_row, min_max=min_max, \
sparse = False, binary_features=b_vars,\numeric_features=n_vars,
target='cnt'):
    y_log = apply_log(y)
# Extract only quantitative features and expand them
    num_index = [j for j, i in enumerate(std_row.feature_names_) if
i in n_vars]
```

```
    x_poly = poly.fit_transform(x[:,num_index])[:,len(num_index):]
    new_x = np.concatenate((x, x_poly), axis=1)

    # MACHINE LEARNING
    if (n+1) >= predictions_start:
        # HOLDOUT AFTER N PHASE
        predicted = SGD.predict(new_x)
        val_rmse += (apply_exp(predicted) - y)**2
        val_rmsle += (predicted - y_log)**2
        if (n-predictions_start+1) % 250 == 0 and (n+1) >
predictions_start:
            print n,
            print '%s holdout RMSE: %0.3f' % (time.strftime('%X'),
(val_rmse / float(n-predictions_start+1))**0.5),
            print 'holdout RMSLE: %0.3f' % ((val_rmsle /
float(n-predictions_start+1))**0.5)
    else:
        # LEARNING PHASE
        SGD.partial_fit(new_x, y_log)
print '%s FINAL holdout RMSE: %0.3f' % (time.strftime('%X'),
(val_rmse / float(n-predictions_start+1))**0.5)
print '%s FINAL holdout RMSLE: %0.3f' % (time.strftime('%X'),
(val_rmsle / float(n-predictions_start+1))**0.5)

Out: ...
21:49:24 FINAL holdout RMSE: 219.191
21:49:24 FINAL holdout RMSLE: 1.480
```

Tip

`PolynomialFeatures` expects a dense matrix, not a sparse one as an input. Our `pull_examples` function allows the setting of a sparse parameter, which, normally set to `True`, can instead be set to `False`, thus returning a dense matrix as a result.

Trying explicit high-dimensional mappings

Though polynomial expansions are a quite powerful transformation, they can be computationally expensive when we are trying to expand to higher degrees and quickly contrast the positive effects of catching important non-linearity by overfitting caused by over-parameterization (when you have too many redundant and not useful features). As seen in SVC and SVR, kernel transformations can come to our aid. SVM kernel transformations, being implicit, require the data matrix in-memory in order to work. There is a class of transformations in Scikit-learn, based on random approximations, which, in the context of a linear model, can achieve very similar results as a kernel SVM.

The `sklearn.kernel_approximation` module contains a few such algorithms:

- `RBFSampler`: This approximates a feature map of an RBF kernel
- `Nystroem`: This approximates a kernel map using a subset of the training data

- `AdditiveChi2Sampler`: This approximates feature mapping for an additive chi2 kernel, a kernel used in computer vision
- `SkewedChi2Sampler`: This approximates feature mapping similar to the skewed chi-squared kernel also used in computer vision

Apart from the Nystroem method, none of the preceding classes require to learn from a sample of your data, making them perfect for online learning. They just need to know how an example vector is shaped (how many features there are) and then they will produce many random non-linearities that can, hopefully, fit well to your data problem.

There are no complex optimization algorithms to explain in these approximation algorithms; in fact, optimization itself is replaced by randomization and the results largely depend on the number of output features, pointed out by the `n_components` parameters. The more the output features, the higher the probability that by chance you'll get the right non-linearities working perfectly with your problem.

It is important to notice that, if chance has really such a great role in creating the right features to improve your predictions, then reproducibility of the results turns out to be essential and you should strive to obtain it or you won't be able to consistently retrain and tune your algorithm in the same way. Noticeably, each class is provided with a `random_state` parameter, thus allowing the controlling of random feature generation and being able to recreate it later on the same just as on different computers.

The theoretical fundamentals of such feature creation techniques are explained in the scientific articles, *Random Features for Large-Scale Kernel Machines* by *A. Rahimi and Benjamin Recht* (http://www.eecs.berkeley.edu/~brecht/papers/07.rah.rec.nips.pdf) and *Weighted Sums of Random Kitchen Sinks: Replacing minimization with randomization in learning* by *A. Rahimi and Benjamin Recht* (http://www.eecs.berkeley.edu/~brecht/papers/08.rah.rec.nips.pdf).

For our purposes, it will suffice to know how to implement the technique and have it contribute to improving our SGD models, both linear and SVM-based:

```
In: source = 'shuffled_covtype.data'
local_path = os.getcwd()
n_vars = ['var_'+str(j) for j in range(54)]
std_row, min_max = explore(target_file=local_path+'\\'+source,
binary_features=list(),
                  fieldnames= n_vars+['covertype'],
numeric_features=n_vars, max_rows=50000)

from sklearn.linear_model import SGDClassifier
from sklearn.kernel_approximation import RBFSampler

SGD = SGDClassifier(loss='hinge', penalty=None, random_state=1,
average=True)
rbf_feature = RBFSampler(gamma=0.5, n_components=300, random_state=0)
accuracy = 0
accuracy_record = list()
predictions_start = 50
```

```
sample = 5000
early_stop = 50000
for x,y,n in pull_examples(target_file=local_path+'\\'+source,
                           vectorizer=std_row,
                           min_max=min_max,
                           binary_features=list(),
                           numeric_features=n_vars,
                           fieldnames= n_vars+['covertype'],
target='covertype', sparse=False):

    rbf_x = rbf_feature.fit_transform(x)
    # LEARNING PHASE
    if n > predictions_start:
        accuracy += int(int(SGD.predict(rbf_x))==y[0])
        if n % sample == 0:
            accuracy_record.append(accuracy / float(sample))
            print '%s Progressive accuracy at example %i: %0.3f' %
(time.strftime('%X'), n, np.mean(accuracy_record[-sample:]))
            accuracy = 0
    if early_stop and n >= early_stop:
            break
    SGD.partial_fit(rbf_x, y, classes=range(1,8))

Out: ...
07:57:45 Progressive accuracy at example 50000: 0.707
```

Hyperparameter tuning

As in batch learning, there are no shortcuts in out-of-core algorithms when testing the best combinations of hyperparameters; you need to try a certain number of combinations to figure out a possible optimal solution and use an out-of-sample error measurement to evaluate their performance.

As you actually do not know if your prediction problem has a simple smooth convex loss or a more complicated one and you do not know exactly how your hyperparameters interact with each other, it is very easy to get stuck into some sub-optimal local-minimum if not enough combinations are tried. Unfortunately, at the moment there are no specialized optimization procedures offered by Scikit-learn for out-of-core algorithms. Given the necessarily long time to train an SGD on a long stream, tuning the hyperparameters can really become a bottleneck when building a model on your data using such techniques.

Here, we present a few rules of thumb that can help you save time and efforts and achieve the best results.

First, you can tune your parameters on a window or a sample of your data that can fit in-memory. As we have seen with kernel SVMs, using a reservoir sample is quite fast even if your stream is huge. Then you can do your optimization in-memory and use the optimal parameters found on your stream.

As Léon Bottou from Microsoft Research has remarked in his technical paper, *Stochastic Gradient Descent Tricks*:

> *"The mathematics of stochastic gradient descent are amazingly independent of the training set size."*

This is true for all the key parameters but especially for the learning rate; the learning rate that works better with a sample will work the best with the full data. In addition, the ideal number of passes over data can be mostly guessed by trying to converge on a small sampled dataset. As a rule of thumb, we report the indicative number of `10**6` examples examined by the algorithm—as pointed out by the Scikit-learn documentation—a number that we have often found accurate, though the ideal number of iterations may change depending on the regularization parameters.

Though most of the work can be done at a relatively small scale when using SGD, we have to define how to approach the problem of fixing multiple parameters. Traditionally, manual search and grid search have been the most used approaches, grid search solving the problem by systematically testing all the combinations of possible parameters at significant values (using, for instance, the log scale checking at the different power degree of 10 or of 2).

Recently, James Bergstra and Yoshua Bengio in their paper, *Random Search for Hyper-Parameter Optimization*, pointed out a different approach based on the random sampling of the values of the hyperparameters. Such an approach, though based on random choices, is often equivalent in results to grid search (but requiring fewer runs) when the number of hyperparameters is low and can exceed the performance of a systematic search when the parameters are many and not all of them are relevant for the algorithm performance.

We leave it to the reader to discover more reasons why this simple and appealing approach works so well in theory by referring to the previously mentioned paper by Bergstra and Bengio. In practice, having experienced its superiority with respect to other approaches, we propose an approach that works well for streams based on Scikit-learn's `ParameterSampler` function in the following example code snippet. `ParameterSampler` is able to randomly sample different sets of hyperparameters (both from distribution functions or lists of discrete values) to be applied to your learning SGD by means of the `set_params` method afterward:

```
In: from sklearn.linear_model import SGDRegressor
from sklearn.grid_search import ParameterSampler

source = '\\bikesharing\\hour.csv'
local_path = os.getcwd()
b_vars = ['holiday','hr','mnth',
'season','weathersit','weekday','workingday','yr']
n_vars = ['hum', 'temp', 'atemp', 'windspeed']
std_row, min_max = explore(target_file=local_path+'\\'+source,
binary_features=b_vars, numeric_features=n_vars)

val_rmse = 0
val_rmsle = 0
predictions_start = 16000
tmp_rsmle = 10**6

def apply_log(x): return np.log(x + 1.0)
def apply_exp(x): return np.exp(x) - 1.0

param_grid = {'penalty':['l1', 'l2'], 'alpha': 10.0**-np.arange(2,5)}
random_tests = 3
search_schedule = list(ParameterSampler(param_grid,
n_iter=random_tests, random_state=5))
results = dict()

for search in search_schedule:
    SGD = SGDRegressor(loss='epsilon_insensitive', epsilon=0.001,
penalty=None, random_state=1, average=True)
    params =SGD.get_params()
    new_params = {p:params[p] if p not in search else search[p] for
p in params}
    SGD.set_params(**new_params)
    print str(search)[1:-1]
    for iterations in range(200):
        for x,y,n in
pull_examples(target_file=local_path+'\\'+source,
                                   vectorizer=std_row,
min_max=min_max, sparse = False,
                                   binary_features=b_vars,
```

```
numeric_features=n_vars, target='cnt'):
            y_log = apply_log(y)

# MACHINE LEARNING
            if (n+1) >= predictions_start:
                # HOLDOUT AFTER N PHASE
                predicted = SGD.predict(x)
                val_rmse += (apply_exp(predicted) - y)**2
                val_rmsle += (predicted - y_log)**2
            else:
                # LEARNING PHASE
                SGD.partial_fit(x, y_log)

        examples = float(n-predictions_start+1) * (iterations+1)
        print_rmse = (val_rmse / examples)**0.5
        print_rmsle = (val_rmsle / examples)**0.5
        if iterations == 0:
            print 'Iteration %i - RMSE: %0.3f - RMSE: %0.3f' %
(iterations+1, print_rmse, print_rmsle)
        if iterations > 0:
            if tmp_rmsle / print_rmsle <= 1.01:
                print 'Iteration %i - RMSE: %0.3f - RMSE: %0.3f\n' %
(iterations+1, print_rmse, print_rmsle)
                results[str(search)]= {'rmse':float(print_rmse),
'rmsle':float(print_rmsle)}
                break
        tmp_rmsle = print_rmsle

Out:
'penalty': 'l2', 'alpha': 0.001
Iteration 1 - RMSE: 216.170 - RMSE: 1.440
Iteration 20 - RMSE: 152.175 - RMSE: 0.857

'penalty': 'l2', 'alpha': 0.0001
Iteration 1 - RMSE: 714.071 - RMSE: 4.096
Iteration 31 - RMSE: 184.677 - RMSE: 1.053

'penalty': 'l1', 'alpha': 0.01
Iteration 1 - RMSE: 1050.809 - RMSE: 6.044
Iteration 36 - RMSE: 225.036 - RMSE: 1.298
```

The code leverages the fact that the bike-sharing dataset is quite small and doesn't require any sampling. In other contexts, it makes sense to limit the number of treated rows or create a smaller sample before by means of reservoir sampling or other sampling techniques for streams seen so far. If you would like to explore optimization in more depth, you can change the `random_tests` variable, fixing the number of sampled hyperparameters' combinations to be tested. Then, you modify the `if tmp_rmsle /`

`print_rmsle <= 1.01` condition using a number nearer to `1.0`—if not `1.0` itself—thus letting the algorithm fully converge until some possible gain in predictive power is feasible.

Tip

Though it is recommended to use distribution functions rather than picking from lists of values, you can still appropriately use the hyperparameters' ranges that we suggested before by simply enlarging the number of values to be possibly picked from the lists. For instance, for alpha in L1 and L2 regularization, you could use NumPy's function, `arrange`, with a small step such as `10.0**-np.arange(1, 7, step=0.1)`, or use NumPy `logspace` with a high number for the `num` parameter: `1.0/np.logspace(1,7,num=50).`

Other alternatives for SVM fast learning

Though the Scikit-learn package provides enough tools and algorithms to learn out-of-core, there are other interesting alternatives among free software. Some are based on the same libraries that Scikit-learn itself uses, such as the Liblinear/SBM and others are completely new, such as sofia-ml, LASVM and Vowpal Wabbit. For instance, Liblinear/SBMis based on selective block minimization and implemented as a fork `liblinear-cdblock` of the original library (https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/#large_linear_classification_when_data_cannot_fit_in_memory). Liblinear/SBM achieves to fit nonlinear SVMs on large amounts of data that cannot fit in-memory using the trick of training the learner using new samples of data and mixing it with previous samples already used for minimization (hence the *blocked* term in the name of the algorithm).

SofiaML (https://code.google.com/archive/p/sofia-ml/) is another alternative. SofiaML is based on an online SVM optimization algorithm called Pegasos SVM. This algorithm is an online SVM approximation, just as another software called LaSVM created by Leon Bottou (http://leon.bottou.org/projects/lasvm). All these solutions can work with sparse data, especially textual, and solve regression, classification, and ranking problems. To date, no alternative solution that we tested proved as fast and versatile as Vowpal Wabbit, the software that we are going to present in the next sections and use to demonstrate how to integrate external programs with Python.

Nonlinear and faster with Vowpal Wabbit

Vowpal Wabbit (**VW**) is an open source project for a fast online learner initially released in 2007 by John Langford, Lihong Li, and Alex Strehl from Yahoo! Research (http://hunch.net/?p=309) and then successively sponsored by Microsoft Research, as John Langford became the principal researcher at Microsoft. The project has developed over the years, arriving today at version 8.1.0 (at the time this chapter was written), with almost one hundred contributors working on it. (For a visualization of the development of the contributions over time, there is an interesting video using the software **Gource** at https://www.youtube.com/watch?v=-aXelGLMMgk.). To date, VW is still being constantly developed and keeps on increasing its learning capabilities at each development iteration.

The striking characteristic of VW is that it is very fast compared to other solutions available (LIBLINEAR, Sofia-ml, svmsgd, and Scikit-learn). Its secret is simple, yet extremely effective: it can load data and learn from it at the same time. An asynchronous thread does the parsing of the examples flowing in as a number of learning threads work on a disjoint set of features, thus assuring a high computational efficiency even when parsing involves high-dimensional feature creation (such as

quadratic or cubic polynomial expansion). In most cases, the real bottleneck of the learning process is the transmission bandwidth of the disk or network transmitting the data to VW.

VW can work out classification (even multiclass and multilabel), regression (OLS and quantile), and active learning problems, offering a vast range of accompanying learning tools (called reductions) such as matrix factorization, **Latent Dirichlet Allocation** (**LDA**), neural networks, n-grams for language models, and bootstrapping.

Installing VW

VW can be retrieved from the online versioning repository GitHub (https://github.com/JohnLangford/vowpal_wabbit), where it can be Git-cloned or just downloaded in the form of a packed zip. Being developed on Linux systems, it is easily compiled on any POSIX environment by a simple sequence of make and make install commands. Detailed instructions for installation are available directly on its installation page and you can download Linux precompiled binaries directly from the author (https://github.com/JohnLangford/vowpal_wabbit/wiki/Download).

A VW version working on Windows operating systems, unfortunately, is a bit more difficult to obtain. In order to create one, the first reference is the documentation itself of VW where a compiling procedure is explained in detail (https://github.com/JohnLangford/vowpal_wabbit/blob/master/README.windows.txt).

Tip

On the accompanying website of the book, we will provide both 32-bit and 64-bit Windows binaries of the 8.1.0 version of VW that we have used for the book.

Understanding the VW data format

VW can work with a particular data format and is invoked from a shell. John Langford uses this sample dataset for his online tutorial (https://github.com/JohnLangford/vowpal_wabbit/wiki/Tutorial), representing three houses whose roofs could be replaced. We find it interesting proposing it to you and commenting on it together:

```
In:
with open('house_dataset','wb') as W:
    W.write("0 | price:.23 sqft:.25 age:.05 2006\n")
    W.write("1 2 'second_house | price:.18 sqft:.15 age:.35 1976\n")
    W.write("0 1 0.5 'third_house | price:.53 sqft:.32 age:.87 1924\
n")

with open('house_dataset','rb') as R:
    for line in R:
        print line.strip()

Out:
0 | price:.23 sqft:.25 age:.05 2006
```

```
1 2 'second_house | price:.18 sqft:.15 age:.35 1976
0 1 0.5 'third_house | price:.53 sqft:.32 age:.87 1924
```

The first noticeable aspect of the file format is that it doesn't have a header. This is because VW uses the hashing trick to allocate the features into a sparse vector, therefore knowing in advance the features that are not necessary at all. Data blocks are divided by pipes (the character `|`) into namespaces, to be intended as different clusters of features, each one containing one or more features.

The first namespace is always the one containing the response variable. The response can be a real number (or an integer) pointing out a numeric value to be regressed, a binary class, or a class among multiple ones. The response is always the first number to be found on a line. A binary class can be encoded using `1` for positive and `-1` for negative (using 0 as a response is allowed only for regression). Multiple classes should be numbered from `1` onward and having gap numbers is not advisable because VW asks for the last class and considers all the integers between `1` and the last one.

The number immediately after the response value is the weight (telling you if you have to consider an example as a multiple example or as a fraction of one), then the base, which plays the role of the initial prediction (a kind of bias). Finally, preceded by the apostrophe character (`'`), there is the label, which can be a number or text to be later found in the VW outputs (in a prediction, you have an identifier for every estimation). Weight, base, and labels are not compulsory: if omitted, weight will be imputed as `1` and base and label won't matter.

Following the first namespace, you can add as many namespaces as you want, labeling each one by a number or string. In order to be considered the label of the namespace, it should be stuck to the pipe, for instance, `|label`.

After the label of the namespace, you can add any feature by its name. The feature name can be anything, but should contain a pipe or colon. You can just put entire texts in the namespaces and every word will be treated as a feature. Every feature will be considered valued as `1`. If you want to assign a different number, just stick a colon at the end of the feature name and put its value after it.

For instance, a valid row readable by Vowpal Wabbit is:

```
0 1 0.5 'third_house | price:.53 sqft:.32 age:.87 1924
```

In the first namespace, the response is `0`, the example weights `1`, the base is `0.5`, and its label is `third_house`. The namespace is nameless and is constituted by four features such as `price` (value is `.53`), `sqft` (value is `.32`), `age` (value is `.87`), and `1924` (value is `1`).

If you have a feature in an example but not in another one, the algorithm will pretend that the feature value is zero in the second example. Therefore, a feature such as `1924` in the preceding example can be intended as a binary variable as, when it is present, it is automatically valued `1`, when missing `0`. This also tells you how VW handles missing values—it automatically considers them as `0` values.

Tip

You can easily handle missing values by putting a new feature when a value is missing. If the feature is age, for instance, you can add a new feature, `age_missing`, which will be a binary variable having value as `1`. When estimating the coefficients, this variable will act as a missing value estimator.

On the author's website, you can also find an input validator, verifying that your input is correct for VW and displaying how it is interpreted by the software:

http://hunch.net/~vw/validate.html

Python integration

There are a few packages integrating it with Python (**vowpal_porpoise**, **Wabbit Wappa**, or **pyvw**) and installing them is easy in Linux systems, but much harder on Windows. No matter whether you are working with Jupyter or IDE, the easiest way to use VW integrated with Python's scripts is to leverage the `Popen` function from the `subprocess` package. That makes VW run in parallel with Python. Python just waits for VW to complete its operation by capturing its output and printing it on the screen:

```
In: import subprocess

def execute_vw(parameters):
    execution = subprocess.Popen('vw '+parameters, \
                shell=True, stderr=subprocess.PIPE)
    line = ""
    history = ""
    while True:
        out = execution.stderr.read(1)
        history += out
        if out == '' and execution.poll() != None:
            print '------------ COMPLETED ------------\n'
            break
        if out != '':
            line += out
            if '\n' in line[-2:]:
                print line[:-2]
                line = ''
    return history.split('\r\n')
```

The functions return a list of the outputs of the learning process, making it easy to process it, extracting relevant reusable information (like the error measure). As a precondition for its correct functioning, place the VW executable (the `vw.exe` file) in the Python working directory or system path where it can be found.

By invoking the function on the previously recorded housing dataset, we can have a look at how it works and what outputs it produces:

```
In:
params = "house_dataset"
results = execute_vw(params)

Out:
Num weight bits = 18
learning rate = 0.5
initial_t = 0
power_t = 0.5
using no cache
Reading datafile = house_dataset
num sources = 1
average  since         example        example  current  current
current
loss     last          counter         weight    label  predict
features
0.000000 0.000000            1            1.0   0.0000
0.0000       5
0.666667 1.000000            2            3.0   1.0000
0.0000       5

finished run
number of examples per pass = 3
passes used = 1
weighted example sum = 4.000000
weighted label sum = 2.000000
average loss = 0.750000
best constant = 0.500000
best constant's loss = 0.250000
total feature number = 15
------------ COMPLETED ------------
```

The initial rows of the output just recall the used parameters and provide confirmation of which data file is being used. Most interesting is the progressive reported by the number of streamed examples (reported by the power of 2, so example 1, 2, 4, 8, 16, and so on). With respect to the loss function, an average loss measure is reported, progressive for the first iteration, based on the hold-out set afterward, whose loss is signaled by postponing the letter `h` (if holding out is excluded, it is possible that just the in-sample measure is reported). On the `example weight` column, the weight of the example is reported and then the example is furthermore described as `current label`, `current predict` and displays the number of features found on that line (`current features`). All such information should help you keep an eye on the stream and learning process.

After the learning is completed, a few reporting measures are reported. The average loss is the most important, in particular when a hold-out is used. Using such loss is most useful for comparative reasons as it can be immediately compared with `best constant's loss` (the baseline predictive power of a simple constant) and with different runs using different parameter configurations.

Another very useful function to integrate VW and Python is a function that we prepared automatically converting CSV files to VW data files. You can find it in the following code snippet. It will help us replicate the previous bike-sharing and covertype problems using VW this time, but it can be easily reused for your own projects:

```
In: import csv

def vw_convert(origin_file, target_file, binary_features,
numeric_features, target, transform_target=lambda(x):x,
               separator=',', classification=True, multiclass=False,
fieldnames= None, header=True, sparse=True):
    """
    Reads a online style stream and returns a generator of
normalized feature vectors

    Parameters
----------
    original_file = the CSV file you are taken the data from
    target file = the file to stream from
    binary_features = the list of qualitative features to consider
    numeric_features = the list of numeric features to consider
    target = the label of the response variable
    transform_target = a function transforming the response
    separator = the field separator character
    classification = a Boolean indicating if it is classification
    multiclass =  a Boolean for multiclass classification
    fieldnames = the fields' labels (can be omitted and read from
file)
    header = a boolean indicating if the original file has an header
    sparse = if a sparse vector is to be returned from the generator
    """
    with open(target_file, 'wb') as W:
        with open(origin_file, 'rb') as R:
iterator = csv.DictReader(R, fieldnames, delimiter=separator)
            for n, row in enumerate(iterator):
                if not header or n>0:
                # DATA PROCESSING
                    response = transform_target(float(row[target]))
                    if classification and not multiclass:
                            if response == 0:
                                stream_row = '-1 '
                            else:
                                stream_row = '1 '
                    else:
                        stream_row = str(response)+' '
                    quantitative = list()
                    qualitative  = list()
```

```
                    for k,v in row.iteritems():
                        if k in binary_features:
                            qualitative.append(str(k)+\
'_'+str(v)+':1')
                        else:
                            if k in numeric_features and
(float(v)!=0 or not sparse):

quantitative.append(str(k)+':'+str(v))
if quantitative:
                        stream_row += '|n '+\
' '.join(quantitative)
                    if qualitative:
                        stream_row += '|q '+\
' '.join(qualitative)
W.write(stream_row+'\n')
```

A few examples using reductions for SVM and neural nets

VW works on minimizing a general cost function, which is as follows:

$$\lambda_1 \|w\|_1 + \frac{\lambda_2}{2} \|w\|_2^2 + \sum_{i=1}^{n} \text{loss}\,(x_i, y_i, w)$$

As in other formulations seen before, w is the coefficient vector and the optimization is obtained separately for each *xi* and *yi* according to the chosen loss function (OLS, logistic, or hinge). Lambda1 and lambda2 are the regularization parameters that, by default, are zero but can be set using the `--l1` and `--l2` options in the VW command line.

Given such a basic structure, VW has been made more complex and complete over time using the reduction paradigm. A reduction is just a way to reuse an existing algorithm in order to solve new problems without coding new solving algorithms from scratch. In other words, if you have a complex machine learning problem A, you just reduce it to B. Solving B hints at a solution of A. This is also justified by the growing interest in machine learning and the exploding number of problems that cannot be solved creating hosts of new algorithms. It is an interesting approach leveraging the existing possibilities offered by basic algorithms and the reason why VW has grown its applicability over time though the program has stayed quite compact. If you are interested in this approach, you can have a look at these two tutorials from John Langford: http://hunch.net/~reductions_tutorial/ and http://hunch.net/~jl/projects/reductions/reductions.html.

For other illustration purposes, we will briefly introduce you to a couple of reductions to implement an SVM with an `RBFkernel` and a shallow neural network using VW in a purely out-of-core way. We will be using some toy datasets for the purpose.

Here is the Iris dataset, changed to a binary classification problem to guess the Iris Versicolor from the Setosa and Virginica:

```
In: import numpy as np
from sklearn.datasets import load_iris, load_boston
from random import seed
iris = load_iris()
seed(2)
re_order = np.random.permutation(len(iris.target))
with open('iris_versicolor.vw','wb') as W1:
    for k in re_order:
        y = iris.target[k]
        X = iris.values()[1][k,:]
        features = ' |f '+' '.join([a+':'+str(b) for a,b in
zip(map(lambda(a): a[:-5].replace(' ','_'), iris.feature_names),X)])
        target = '1' if y==1 else '-1'
        W1.write(target+features+'\n')
```

Then for a regression problem, we will be using the Boston house pricing dataset:

```
In: boston = load_boston()
seed(2)
re_order = np.random.permutation(len(boston.target))
with open('boston.vw','wb') as W1:
     for k in re_order:
        y = boston.target[k]
        X = boston.data[k,:]
        features = ' |f '+' '.join([a+':'+str(b) for a,b in
zip(map(lambda(a): a[:-5].replace(' ','_'), iris.feature_names),X)])
        W1.write(str(y)+features+'\n')
```

First, we will be trying SVM. `kvsm` is a reduction based on the LaSVM algorithm (*Fast Kernel Classifiers with Online and Active Learning*—http://www.jmlr.org/papers/volume6/bordes05a/bordes05a.pdf) without a bias term. The VW version typically works in just one pass and with a 1-2 reprocessing of a randomly picked support vector (though some problems may need multiple passes and reprocessing). In our case, we are just using a single pass and a couple of reprocessings in order to fit an RBF kernel on our binary problem (KSVM works only for classification problems). Implemented kernels are linear, radial basis function, and polynomial. In order to have it work, use the `--ksvm` option, set a number for reprocessing (default is 1) by `--reprocess`, choose the kernel with `--kernel` (options are `linear`, `poly`, and `rbf`). Then, if the kernel is polynomial, set an integer number for `--degree`, or a float (default is 1.0) for `--bandwidth` if you are using RBF. You also have to compulsorily specify l2 regularization; otherwise, the reduction won't work properly. In our example, we make an `RBFkernel` with bandwidth 0.1:

```
In: params = '--ksvm --l2 0.000001 --reprocess 2 -b 18 --kernel rbf
--bandwidth=0.1 -p iris_bin.test -d iris_versicolor.vw'
results = execute_vw(params)
```

```
accuracy = 0
with open('iris_bin.test', 'rb') as R:
    with open('iris_versicolor.vw', 'rb') as TRAIN:
        holdouts = 0.0
        for n,(line, example) in enumerate(zip(R,TRAIN)):
            if (n+1) % 10==0:
                predicted = float(line.strip())
                y = float(example.split('|')[0])
                accuracy += np.sign(predicted)==np.sign(y)
                holdouts += 1
print 'holdout accuracy: %0.3f' % ((accuracy / holdouts)**0.5)

Out: holdout accuracy: 0.966
```

Neural networks are another cool addition to VW; thanks to the work of Paul Mineiro (http://www.machinedlearnings.com/2012/11/unpimp-your-sigmoid.html), VW can implement a single-layer neural network with hyperbolic tangent (*tanh*) activation and, optionally, dropout (using the `--dropout` option). Though it is only possible to decide the number of neurons, the neural reduction works fine with both regression and classification problems and can smoothly take on other transformations by VW as inputs (such as quadratic variables and n-grams), making it a very well-integrated, versatile (neural networks can solve quite a lot of problems), and fast solution. In our example, we apply it to the Boston dataset using five neurons and dropout:

```
In: params = 'boston.vw -f boston.model --loss_function squared -k
--cache_file cache_train.vw --passes=20 --nn 5 --dropout'
results = execute_vw(params)
params = '-t boston.vw -i boston.model -k --cache_file cache_test.vw
-p boston.test'
results = execute_vw(params)
val_rmse = 0
with open('boston.test', 'rb') as R:
    with open('boston.vw', 'rb') as TRAIN:
        holdouts = 0.0
        for n,(line, example) in enumerate(zip(R,TRAIN)):
            if (n+1) % 10==0:
                predicted = float(line.strip())
                y = float(example.split('|')[0])
                val_rmse += (predicted - y)**2
                holdouts += 1
print 'holdout RMSE: %0.3f' % ((val_rmse / holdouts)**0.5)

Out: holdout RMSE: 7.010
```

Faster bike-sharing

Let's try VW on the previously created bike-sharing example file in order to explain the output components. As the first step, you have to transform the CSV file into a VW file and the previous `vw_convert` function will come in handy doing this. As before, we will apply a logarithmic transformation on the numeric response, using the `apply_log` function passed by the `transform_target` parameter of the `vw_convert` function:

```
In: import os
import numpy as np

def apply_log(x):
    return np.log(x + 1.0)

def apply_exp(x):
    return np.exp(x) - 1.0

local_path = os.getcwd()
b_vars = ['holiday','hr','mnth',
'season','weathersit','weekday','workingday','yr']
n_vars = ['hum', 'temp', 'atemp', 'windspeed']
source = '\\bikesharing\\hour.csv'
origin = target_file=local_path+'\\'+source
target = target_file=local_path+'\\'+'bike.vw'
vw_convert(origin, target, binary_features=b_vars,
numeric_features=n_vars, target = 'cnt', transform_target=apply_log,
               separator=',', classification=False,
multiclass=False, fieldnames= None, header=True)
```

After a few seconds, the new file should be ready. We can immediately run our solution, which is a simple linear regression (the default option in VW). The learning is expected to run for 100 passes, controlled by the out-of-sample validation that VW automatically implements (drawing systematically and in a repeatable way, a single observation out of every 10 as validation). In this case, we decide to set a holdout sample after 16,000 examples (using the `--holdout_after` option). When the validation error on the validation increases (instead of decreasing), VW stops after a few iterations (three by default, but the number can be changed using the `--early_terminate` option), avoiding overfitting the data:

```
In: params = 'bike.vw -f regression.model -k --cache_file
cache_train.vw --passes=100 --hash strings --holdout_after 16000'
results = execute_vw(params)

Out: …
finished run
number of examples per pass = 15999
passes used = 6
weighted example sum = 95994.000000
```

```
weighted label sum = 439183.191893
average loss = 0.427485 h
best constant = 4.575111
total feature number = 1235898
------------ COMPLETED ------------
```

The final report indicates that six passes (out of 100 possible ones) have been completed and the out-of-sample average loss is 0.428. As we are interested in `RMSE` and `RMSLE`, we have to calculate them ourselves.

We then predict the results in a file (`pred.test`) in order to be able to read them and calculate our error measure using the same holdout strategy as in the training set. The results are indeed much better (in a fraction of the time) than what we previously obtained with Scikit-learn's SGD:

```
In: params = '-t bike.vw -i regression.model -k --cache_file
cache_test.vw -p pred.test'
results = execute_vw(params)
val_rmse = 0
val_rmsle = 0
with open('pred.test', 'rb') as R:
    with open('bike.vw', 'rb') as TRAIN:
        holdouts = 0.0
        for n,(line, example) in enumerate(zip(R,TRAIN)):
            if n > 16000:
                predicted = float(line.strip())
                y_log = float(example.split('|')[0])
                y = apply_exp(y_log)
                val_rmse += (apply_exp(predicted) - y)**2
                val_rmsle += (predicted - y_log)**2
                holdouts += 1

print 'holdout RMSE: %0.3f' % ((val_rmse / holdouts)**0.5)
print 'holdout RMSLE: %0.3f' % ((val_rmsle / holdouts)**0.5)

Out:
holdout RMSE: 135.306
holdout RMSLE: 0.845
```

The covertype dataset crunched by VW

The covertype problem can also be solved better and more easily by VW than we managed before. This time, we will have to set some parameters and decide on the **error correcting tournament** (**ECT**, invoked by the`--ect` parameter on VW), where each class competes in an elimination tournament to be the label for an example. In many examples, ECT can outperform **one-against-all** (**OAA**), but this is not a general rule, and ECT is one of the approaches to be tested when dealing with multiclass problems. (The other possible option is `--log_multi`, using online decision trees to split the sample in smaller sets where we apply single predictive models.) We also set the learning rate to 1.0 and create a third-

degree polynomial expansion using the `--cubic` parameter, pointing out which namespaces have to be multiplied by each other (In this case, the namespace f for three times is expressed by the `nnn` string followed by `--cubic`.):

```
In: import os
local_path = os.getcwd()
n_vars = ['var_'+'0'*int(j<10)+str(j) for j in range(54)]
source = 'shuffled_covtype.data'
origin = target_file=local_path+'\\'+source
target = target_file=local_path+'\\'+'covtype.vw'
vw_convert(origin, target, binary_features=list(), fieldnames=
n_vars+['covertype'], numeric_features=n_vars,
    target = 'covertype', separator=',', classification=True,
multiclass=True, header=False, sparse=False)
params = 'covtype.vw --ect 7 -f multiclass.model -k --cache_file
cache_train.vw --passes=2 -l 1.0 --cubic nnn'
results = execute_vw(params)

Out:
finished run
number of examples per pass = 522911
passes used = 2
weighted example sum = 1045822.000000
weighted label sum = 0.000000
average loss = 0.235538 h
total feature number = 384838154
------------ COMPLETED ------------
```

Tip

In order for the example to be speedy, we limit the number of passes to just two. If you have time, raise the number to 100 and witness how the obtained accuracy can be improved furthermore.

Here, we wouldn't need to inspect the error measure further as the reported average loss is the complement to 1.0 of the accuracy measure; we just calculate it for completeness, confirming that our holdout accuracy is exactly `0.769`:

```
In: params = '-t covtype.vw -i multiclass.model -k --cache_file
cache_test.vw -p covertype.test'
results = execute_vw(params)
accuracy = 0
with open('covertype.test', 'rb') as R:
    with open('covtype.vw', 'rb') as TRAIN:
        holdouts = 0.0
        for n,(line, example) in enumerate(zip(R,TRAIN)):
            if (n+1) % 10==0:
                predicted = float(line.strip())
```

```
                y = float(example.split('|')[0])
                accuracy += predicted ==y
                holdouts += 1
print 'holdout accuracy: %0.3f' % (accuracy / holdouts)

Out: holdout accuracy: 0.769
```

Summary

In this chapter, we expanded on the initial discussion of out-of-core algorithms by adding SVMs to simple regression-based linear models. Most of the time, we focused on Scikit-learn implementations—mostly SGD—and concluded with an overview of external tools that can be integrated with Python scripts, such as Vowpal Wabbit by John Langford. Along the way, we completed our overview on model improvement and validation technicalities when working out-of-core by discussing reservoir sampling, regularization, explicit and implicit nonlinear transformations, and hyperparameter optimization.

In the next chapter, we will get involved with even more complex and powerful learning approaches while presenting deep learning and neural networks in large scale problems. If your projects revolve around the analysis of images and sounds, what we have seen so far may not yet be the magic solution you were looking for. The next chapter will provide all the desired solutions.

Chapter 4. Neural Networks and Deep Learning

In this chapter, we will cover one of the most exciting fields in artificial intelligence and machine learning: Deep Learning. This chapter will walk through the most important concepts necessary to apply deep learning effectively. The topics that we will cover in this chapter are as follows:

- Essential neural network theory
- Running neural networks on the GPU or CPU
- Parameter tuning for neural networks
- Large scale deep learning on H2O
- Deep learning with autoencoders (pretraining)

Deep learning emerged from the subfield of artificial intelligence that developed neural networks. Strictly speaking, any large neural network can be considered *deep-learning*. However, recent developments in deep architectures require more than setting up large neural networks. The difference between deep architectures and normal multilayer networks is that a deep architecture consists of multiple preprocessing and unsupervised steps that detect latent dimension in the data to be later fed into further stages of the network. The most important thing to know about deep learning is that new features are learned and transformed through these deep architectures in order to improve the overall learning accuracy. So, an important distinction between the current generation of deep learning methods and other machine learning approaches is that, with deep learning, the task of feature engineering is in part automated. Don't worry too much if these concepts sound abstract, they will be made clear later in this chapter along with practical examples. These deep learning methods introduce new complexities, which make applying them effectively quite challenging.

The biggest challenges are their difficulty in training, computation time, and parameter tuning. Solutions for these difficulties will be dealt with in this chapter.

In the last decade, interesting applications of deep learning can be found in computer vision, natural language processing, and audio processing, applications such as Facebook's deep face project created by a research group at Facebook partly led by the well-known deep learning scholar, Yann LeCun. Deep face aims to extract and identify human faces from digital images. Google has its own project, **DeepMind**, led by Geoffrey Hinton. Google recently introduced **TensorFlow**, an open source library providing deep learning applications, which will be covered in detail in the next chapter.

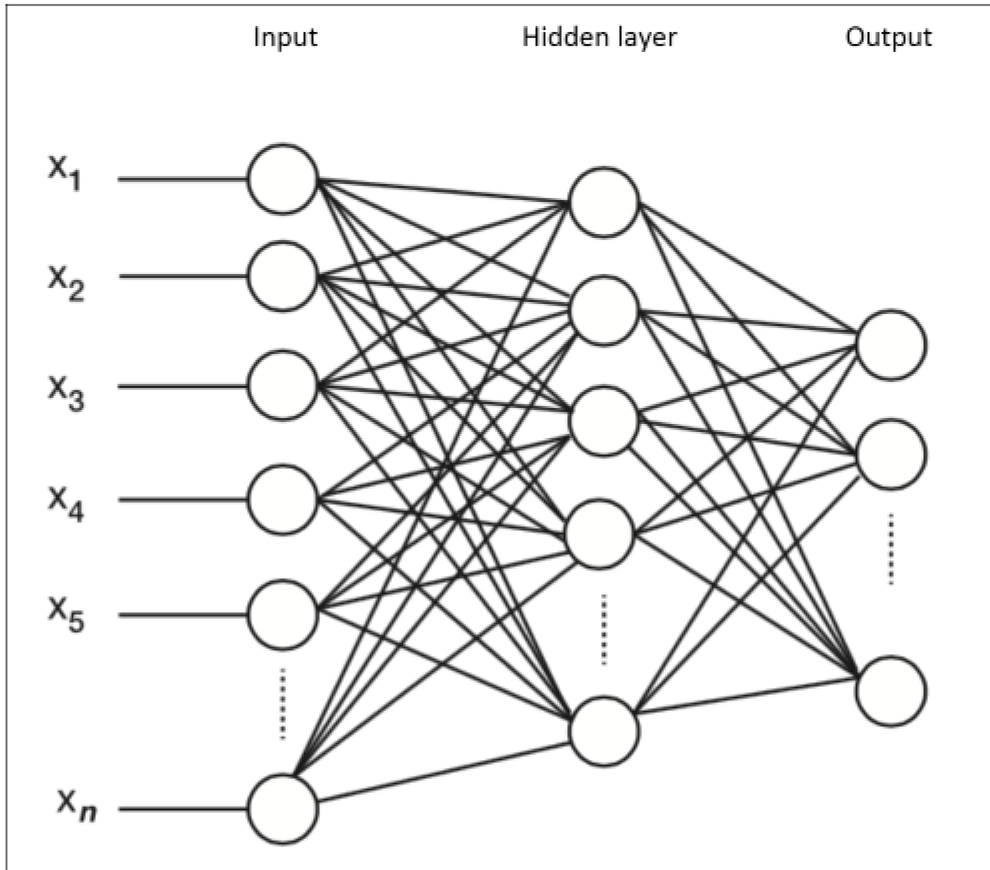
Before we start unleashing autonomous intelligent agents passing Turing tests and math competitions, let's step back a bit and run through the basics.

The neural network architecture

Let's now focus on how neural networks are organized, starting from their architecture and a few definitions.

A network where the flow of learning is passed forward all the way to the outputs in one pass is referred to as a **feedforward neural network**.

A basic feedforward neural network can easily be depicted by a network diagram, as shown here:



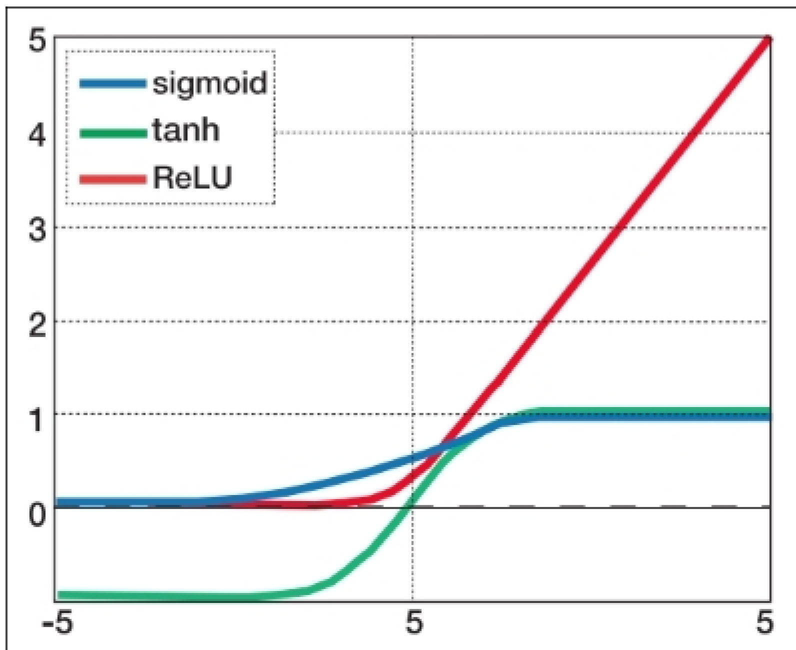
In the network diagram, you can see that this architecture consists of an input layer, hidden layer, and output layer. The input layer contains the feature vectors (where each observation has *n* features), and the output layer consists of separate units for each class of the output vector in the case of classification and a single numerical vector in the case of regression.

The strength of the connections between the units is expressed through weights later to be passed to an activation function. The goal of an activation function is to transform its input to an output that makes binary decisions more separable.

These activation functions are preferably differentiable so they can be used to learn.

The widely-used activation functions are **sigmoid** and **tanh**, and even more recently the **rectified linear unit** (**ReLU**) has gained traction. Let's compare the most important activation functions so that we understand their advantages and drawbacks. Note that we mention the output range and active range of the function. The output range is simply the actual output of the function itself. The active range, however, is a little more complicated; it is the range where the gradient has the most variance in the final weight updates. This means that outside of this range, the gradient is near zero and does not add to the parameter updates during learning. This problem of a close-to-zero gradient is also referred to as the

vanishing gradient problem and is solved by the ReLU activation function, which at this time is the most popular activation for larger neural networks:



It is important to note that features need to be scaled to the *active range* of the chosen activation function. Most up-to-date packages will have this as a standard preprocessing procedure so you don't need to do this yourself:

sigmoid	$\frac{1}{(1 + e^{-t})}$	Active range: [sqrt(3), sqrt(3)] Output range: (0, 1)

Sigmoid functions are often used for mathematical convenience because their derivatives are very easy to calculate, which we will use to calculate the weight updates in training algorithms:

tanh function	$\frac{e^t - e^{-t}}{e^t + e^{-t}}$	Active range: [-2,2] Output range: (-1,+1)

Interestingly the tanh and logistic sigmoid functions are related linearly and tanh can be seen as a rescaled version of the sigmoid function so that its range is between `-1` and `1`.

rectified linear unit (ReLU)	$f(x) = \max(0, x)$	Active range: [0, inf]

This function is the best choice for deeper architectures. It can be seen as a ramp function whose range lies above `0` to infinity. You can see that it is much easier to calculate than the sigmoid function. The biggest benefit of this function is that it bypasses the vanishing gradient problem. If ReLU is an option during a deep learning project, use it.

Softmax for classification

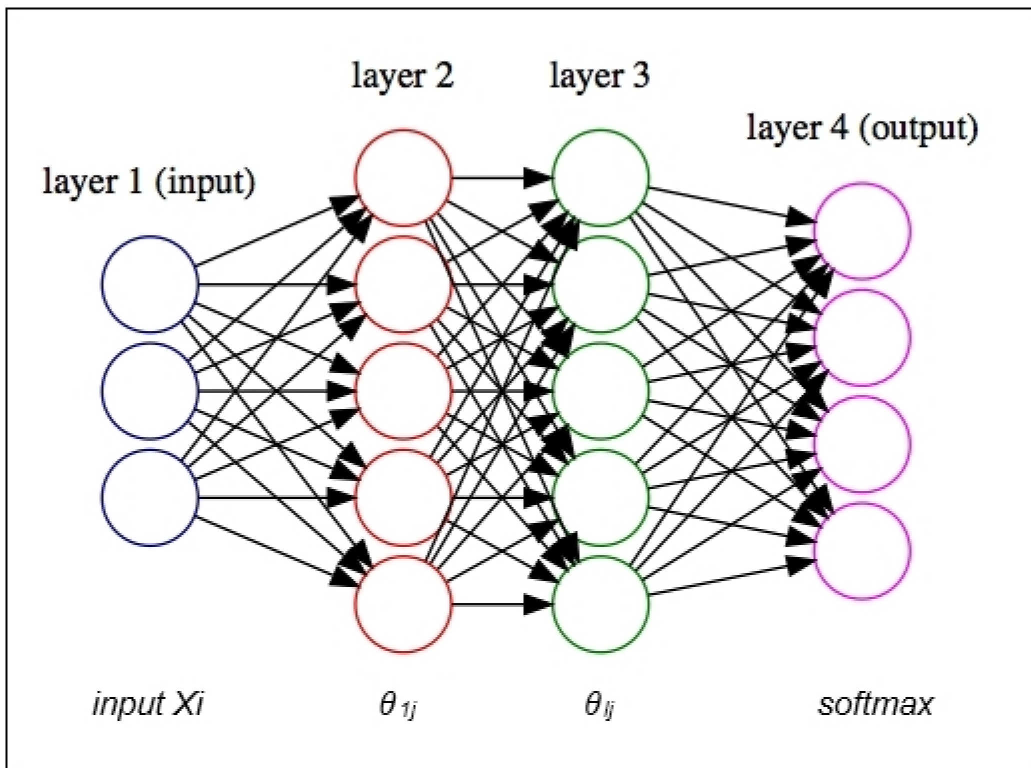
So far, we have seen that activation functions transform the values within a certain range after they are multiplied with the weight vectors. We also need to transform the outputs of the last hidden layer before providing balanced classes or probability outputs (log-likelihood values).

This will convert the output of the previous layer to probability values so that a final class prediction can be made. The exponentiation in this case will return a near-zero value whenever the output is significantly less than the maximum of all the values; this way the differences are amplified:

$$softmax(k, x_1, \ldots, x_n) = \frac{e^{xk}}{\sum_{i=1}^{n} e^{xi}}$$

Forward propagation

Now that we understand activation functions and the final outputs of a network, let's see how the input features are fed through the network to provide a final prediction. Computations with huge chunks of units and connections might look like a complex task, but fortunately the feedforward process of a neural network comes down to a sequence of vector computations:



We arrive at a final prediction by performing the following steps:

1. Performing a dot-product on the inputs with the weights between the first and second layer and transforming the result with the activation function.
2. Performing a dot-product on the outputs of the first hidden layer with the weights between the second and third layer. These results are then transformed with the activation function on each unit of the second hidden layer.
3. Finally, we arrive at our prediction by multiplying the vector with the activation function (softmax for classification).

We can treat each layer in the network as a vector and apply simple vector multiplications. More formally, this will look like the following: θ

= the weight vector of layer *x*

b1 and *b2* are the bias units

f = the activation function

$z_{(2)}=\theta_{(1)}x+b_{(1)}$

$a_{(2)}=f(z_{(2)})$ *transformation after layer 1*

$z_{(3)}=\theta_{(2)}a_{(2)}+b_{(2)}$

$h_{W,b}(x)=a_{(3)}=f_{(softmax)}(z_{(3)})$ *final output with softmax transformation*

Note

Note that this example is based on a single hidden layer network architecture.

Let's perform a simple feedforward pass on a neural network with two hidden layers with basic NumPy. We apply a `softmax` function to the final output:

```
import numpy as np
import math
b1=0 #bias unit 1
b2=0 #bias unit 2

def sigmoid(x):      # sigmoid function
    return 1 /(1+(math.e**-x))

def softmax(x):     #softmax function
    l_exp = np.exp(x)
    sm = l_exp/np.sum(l_exp, axis=0)
    return sm

# input dataset with 3 features
X = np.array([  [.35,.21,.33],
    [.2,.4,.3],
    [.4,.34,.5],
    [.18,.21,16] ])
len_X = len(X) # training set size
input_dim = 3 # input layer dimensionality
output_dim = 1 # output layer dimensionality
hidden_units=4

np.random.seed(22)
# create random weight vectors
theta0 = 2*np.random.random((input_dim, hidden_units))
theta1 = 2*np.random.random((hidden_units, output_dim))

# forward propagation pass
d1 = X.dot(theta0)+b1
l1=sigmoid(d1)
l2 = l1.dot(theta1)+b2
#let's apply softmax to the output of the final layer
output=softmax(l2)
```

Note

Note that the bias unit enables the function to move up and down and will help fit the target values more closely. Each hidden layer consists of one bias unit.

Backpropagation

With our simple feedforward example, we have taken our first steps in training the model. Neural networks are trained quite similarly to gradient descent methods that we have seen with other machine learning algorithms. Namely, we upgrade the parameters of a model in order to find the global minimum of the error function. An important difference with neural networks is that we now have to deal with multiple units across the network that we need to train independently. We do this using the partial derivative of the cost function and calculating how much the error curve drops when we change the particular parameter vector by a certain amount (the learning rate). We start with the layer closest to the output and calculate the gradient with respect to the derivative of our loss function. If there are hidden layers, we move to the second hidden layer and update the weights until the first layer in the feedforward network is reached.

The core idea of backpropagation is quite similar to other machine learning algorithms, with the important complication that we are dealing with multiple layers and units. We have seen that each layer in the network is represented by a weight vector θij. So, how do we solve this issue? It might seem intimidating that we have to train a large number of weights independently. However, quite conveniently, we can use vectorized operations. Just like we did with the forward pass, we calculate the gradients and update the weights applied to the weight vectors (θ ij).

We can summarize the following steps in the backpropagation algorithm:

1. Feedforward pass: We randomly initialize the weight vectors and multiply the input with the subsequent weight vectors toward a final output.
2. Calculate the error: We calculate the error/loss of the output of the feedforward step.

   Randomly initialize the weight vectors.
3. Backpropagation to the last hidden layer (with respect to the output). We calculate the gradient of this error and change weights toward the direction of the gradient. We do this by multiplying the weight vector θ j with the gradients performed.
4. Update the weights till the stopping criterion is reached (minimum error or number of training rounds):

   $$\theta ij := \theta ij - \boldsymbol{\eta} * \Delta_{\theta} J(\theta ij)$$

We have now covered a feedforward pass of an arbitrary two-layer neural network; let's apply backpropagation with SGD in NumPy to the same input that we used in the previous example. Take special note of how we upgrade the weight parameters:

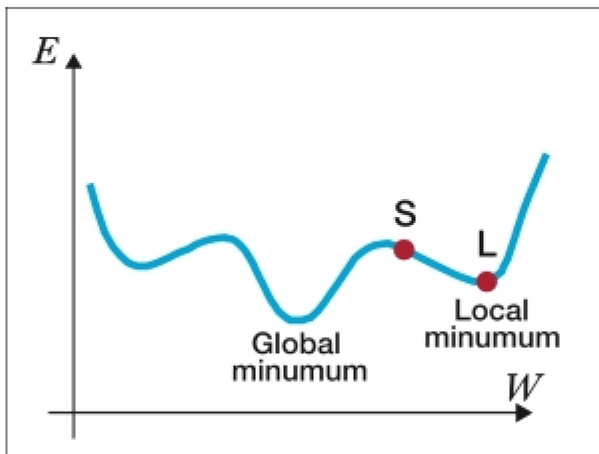
```
import numpy as np
import math
def sigmoid(x):      # sigmoid function
    return 1 /(1+(math.e**-x))

def deriv_sigmoid(y): #the derivative of the sigmoid function
    return y * (1.0 - y)

alpha=.1    #this is the learning rate
X = np.array([  [.35,.21,.33],
    [.2,.4,.3],
    [.4,.34,.5],
    [.18,.21,16] ])
y = np.array([[0],
        [1],
        [1],
        [0]])
np.random.seed(1)
#We randomly initialize the layers
theta0 = 2*np.random.random((3,4)) - 1
theta1 = 2*np.random.random((4,1)) - 1

for iter in range(205000): #here we specify the amount of training
rounds.
    # Feedforward the input like we did in the previous exercise
    input_layer = X
    l1 = sigmoid(np.dot(input_layer,theta0))
    l2 = sigmoid(np.dot(l1,theta1))

    # Calculate error
    l2_error = y - l2

    if (iter% 1000) == 0:
        print "Neuralnet accuracy:" +
str(np.mean(1-(np.abs(l2_error))))

    # Calculate the gradients in vectorized form
    # Softmax and bias units are left out for instructional
simplicity
    l2_delta = alpha*(l2_error*deriv_sigmoid(l2))
    l1_error = l2_delta.dot(theta1.T)
    l1_delta = alpha*(l1_error * deriv_sigmoid(l1))

    theta1 += l1.T.dot(l2_delta)
    theta0 += input_layer.T.dot(l1_delta)
```

Now look how the accuracy increases with each pass over the network:

```
Neuralnet accuracy:0.983345051044
Neuralnet accuracy:0.983404936523
Neuralnet accuracy:0.983464255273
Neuralnet accuracy:0.983523015841
Neuralnet accuracy:0.983581226603
Neuralnet accuracy:0.983638895759
Neuralnet accuracy:0.983696031345
Neuralnet accuracy:0.983752641234
Neuralnet accuracy:0.983808733139
Neuralnet accuracy:0.98386431462
Neuralnet accuracy:0.983919393086
Neuralnet accuracy:0.983973975799
Neuralnet accuracy:0.984028069878
Neuralnet accuracy:0.984081682304
Neuralnet accuracy:0.984134819919
```

Common problems with backpropagation

One familiar problem with neural networks is that, during optimization with backpropagation, the gradient can get stuck in local minima. This occurs when the error minimization is tricked into seeing a minimum (the point **S** in the image) where it is really just a local bump to pass the peak **S**:



Another common problem is when the gradient descent misses the global minimum, which can sometimes result in surprisingly poor performing models. This problem is referred to as **overshooting**.

It is possible to solve both these problems by choosing a lower learning rate when the model is overshooting or choose a higher learning rate when getting stuck in local minima. Sometimes this adjustment still doesn't lead to a satisfying and quick convergence. Recently, a range of solutions has been found to mitigate these problems. Learning algorithms with tweaks to the vanilla `SGDalgorithms` that we just covered have been developed. It is important to understand them so that you can choose the right one for any given task. Let's cover these learning algorithms in more detail.

Backpropagation with mini batch

Batch gradient descent computes the gradient using the whole dataset but backpropagation SGD can also work with so-called **mini batches**, where a sample of the dataset with size *k* (batches) is used to update the learning parameter. The amount of error irregularity between each update can be smoothened out with mini batch, which might avoid getting stuck in and overshooting local minima. In most neural network packages, we can change the batch size of the algorithm (we will look at this later). Depending on the amount of training examples, a batch size anywhere between 10 and 300 can be helpful.

Momentum training

Momentum is a method that adds a fraction of the previous weight update to the current one:

$$v_{t+1} = \mu v_t - \eta \nabla \mathcal{L}(\theta_t)$$
$$\theta_{t+1} = \theta_t + v_{t+1}$$

Here, a fraction of the previous weight update is added to the current one. A high momentum parameter can help increase the speed of convergence reaching the global minimum faster. Looking at the formulation, you can see a *v* parameter. This is the equivalent of the velocity of the gradient updates with a learning rate η. A simple way to understand this is to see that when the gradient keeps pointing in the same direction over multiple instances, the speed of convergence increases with each step toward the minimum. This also removes irregularities between the gradients by a certain margin. Most packages will have this momentum parameter available (as we will see in a later example). When we set this parameter too high, we have to keep in mind that there is a risk of overshooting the global minimum. On the other hand, when we set the momentum parameter too low, the coefficient might get stuck in local minima and can also slow down learning. Ideal settings for the momentum coefficient are normally in the .5 and .99 range.

Nesterov momentum

Nesterov momentum is a newer and improved version of classical momentum. In addition to classical momentum training, it will *look ahead* in the direction of the gradient. In other words, Nesterov momentum takes a simple step going from x to y, and moves a little bit further in that direction so that *x* to *y* becomes *x* to *{y (v1 +1)}* in the direction given by the previous point. I will spare you the technical details, but remember that it consistently outperforms normal momentum training in terms of convergence. If there is an option for Nesterov momentum, use it.

Adaptive gradient (ADAGRAD)

ADAGRAD provides a feature-specific learning rate that utilizes information from the previous upgrades:

$$g_{t+1} = g_t + \nabla\mathcal{L}(\theta_t)^2$$

$$\theta_{t+1} = \theta_t - \frac{\eta\nabla\mathcal{L}(\theta_t)}{\sqrt{g_{t+1} + \epsilon}}$$

ADAGRAD updates the learning rate for each parameter according to information from previously iterated gradients for that parameter. This is done by dividing each term by the square root of the sum of squares of its previous gradient. This allows the learning rate to decrease over time because the sum of squares will continue to increase with each iteration. A decreasing learning rate has the advantage of decreasing the risk of overshooting the global minimum quite substantially.

Resilient backpropagation (RPROP)

RPROP is an adaptive method that does not look at historical information, but merely looks at the sign of the partial derivative over a training instance and updates the weights accordingly.

$$\Delta_{ij}^{(t)} = \begin{cases} n^{+} * \Delta_{ij}^{(t-1)}, if \frac{\partial E^{(t-1)}}{\partial w_{ij}} * \frac{\partial E^{(t)}}{\partial w_{ij}} > 0 \\ n^{-} * \Delta_{ij}^{(t-1)}, if \frac{\partial E^{(t-1)}}{\partial w_{ij}} * \frac{\partial E^{(t)}}{\partial w_{ij}} < 0 \\ \Delta_{ij}^{(t-1)}, else \end{cases}$$

$$where\, 0 < \eta^{-} < 1 < \eta^{+}$$

A direct adaptive method for faster backpropagation learning: The RPROP Algorithm. Martin Riedmiller 1993

RPROP is an adaptive method that does not look at historical information, but merely looks at the sign of the partial derivative over a training instance and updates the weights accordingly. Inspecting the preceding image closely, we can see that once the partial derivative of the error changes its sign (> *0* or < *0*), the gradient starts moving in the opposite direction, leading toward the global minimum correcting for the overshooting. However, if the sign doesn't change at all, larger steps are taken toward the global

minimum. Lots of articles have proven the superiority of RPROP over ADAGRAD but in practice, this is not confirmed consistently. Another important thing to keep in mind is that RPROP does not work properly with mini batches.

RMSProp

RMSProp is an adaptive learning method without shrinking the learning rate:

$$\theta_{t+1} = \theta t - \frac{\eta}{\sqrt{E[g2]t+e}} g_t$$

RMSProp is also an adaptive learning method that utilizes ideas from momentum learning and ADAGRAD, with the important addition that it avoids the shrinkage of the learning rate over time. With this technique, the shrinkage is controlled with an exponential decay function over the average of the gradients.

The following is the list of gradient descent optimization algorithms:

	Applications	**Common problems**	**Practical tips**
Regular SGD	Widely applicable	Overshooting, stuck in local minima	Use with momentum and mini-batch
ADAGRAD	Smaller datasets <10k	Slow convergence	Use a learning rate between .01 and .1. Widely applicable. Works with sparse data
RPROP	Larger datasets >10k	Not effective with mini-batches	Use RMSProp when possible
RMSProp	Larger datasets >10k	Not effective with wide and shallow nets	Particularly useful for wide sparse data

What and how neural networks learn

Now that we have a basic understanding of backpropagation in all its forms, it is time to address the most difficult task in neural network projects: How do we chose the right architecture? One crucial

capability of neural networks is that the weights within an architecture can transform the input into a nonlinear feature space and thereby solve nonlinear classification (decision boundaries) and regression problems. Let's do a simple yet insightful exercise to demonstrate this idea in the `neurolab` package. We will only use `neurolab` for a short exercise; for scalable learning problems, we will propose other methods.

First, install the `neurolab` package with `pip`.

Install `neurolab` from the terminal:

```
> $pip install neurolab
```

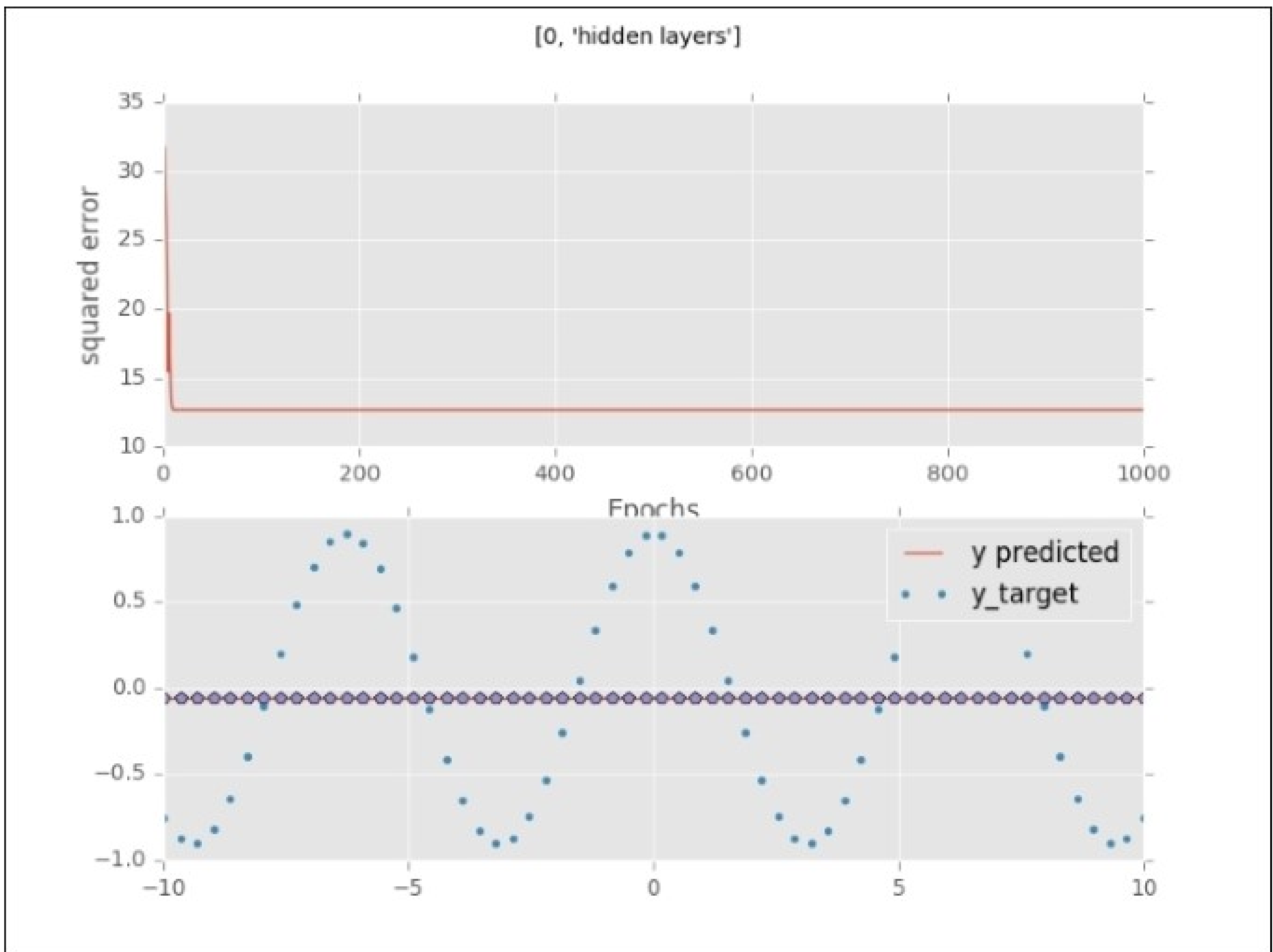
With this example, we will generate a simple nonlinear cosine function with `numpy` and train a neural network to predict the cosine function from a variable. We will set up several neural network architectures to see how well each architecture is able to predict the cosine target variable:

```
import neurolab as nl
import numpy as np
from sklearn import preprocessing
import matplotlib.pyplot as plt
plt.style.use('ggplot')
# Create train samples
x = np.linspace(-10,10, 60)
y = np.cos(x) * 0.9
size = len(x)
x_train = x.reshape(size,1)
y_train = y.reshape(size,1)

# Create network with 4 layers and random initialized
# just experiment with the amount of layers

d=[[1,1],[45,1],[45,45,1],[45,45,45,1]]
for i in range(4):
    net = nl.net.newff([[-10, 10]],d[i])
    train_net=nl.train.train_gd(net, x_train, y_train, epochs=1000,
show=100)
    outp=net.sim(x_train)
# Plot results (dual plot with error curve and predicted values)
    import matplotlib.pyplot
    plt.subplot(2, 1, 1)
    plt.plot(train_net)
    plt.xlabel('Epochs')
    plt.ylabel('squared error')
    x2 = np.linspace(-10.0,10.0,150)
    y2 = net.sim(x2.reshape(x2.size,1)).reshape(x2.size)
    y3 = outp.reshape(size)
    plt.subplot(2, 1, 2)
```

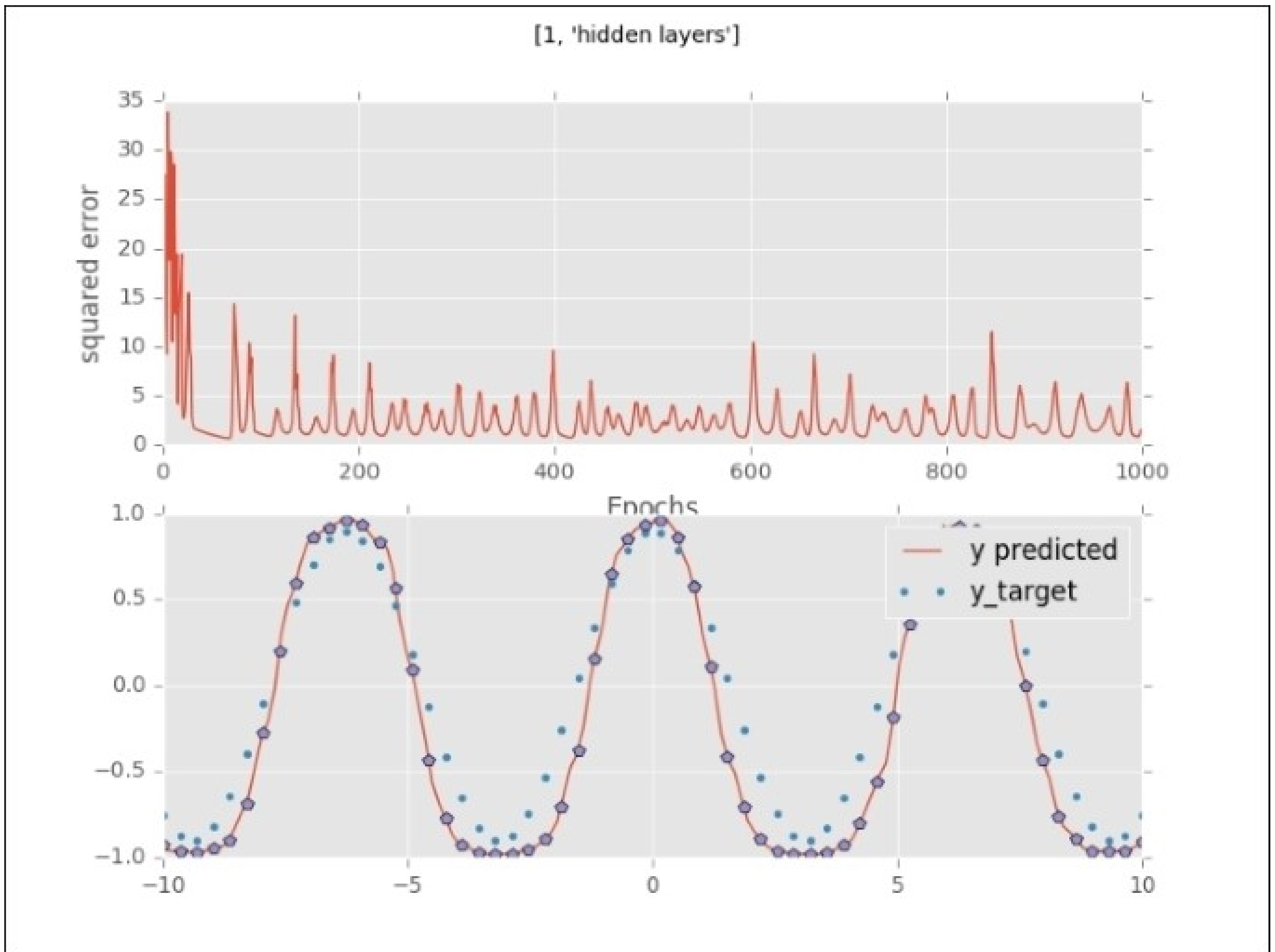
```
plt.suptitle([i ,'hidden layers'])
plt.plot(x2, y2, '-',x , y, '.', x, y3, 'p')
plt.legend(['y predicted', 'y_target'])
plt.show()
```

Now look closely at how the error curve behaves and how the predicted values start to approximate the target values as we add more layers to the neural network.

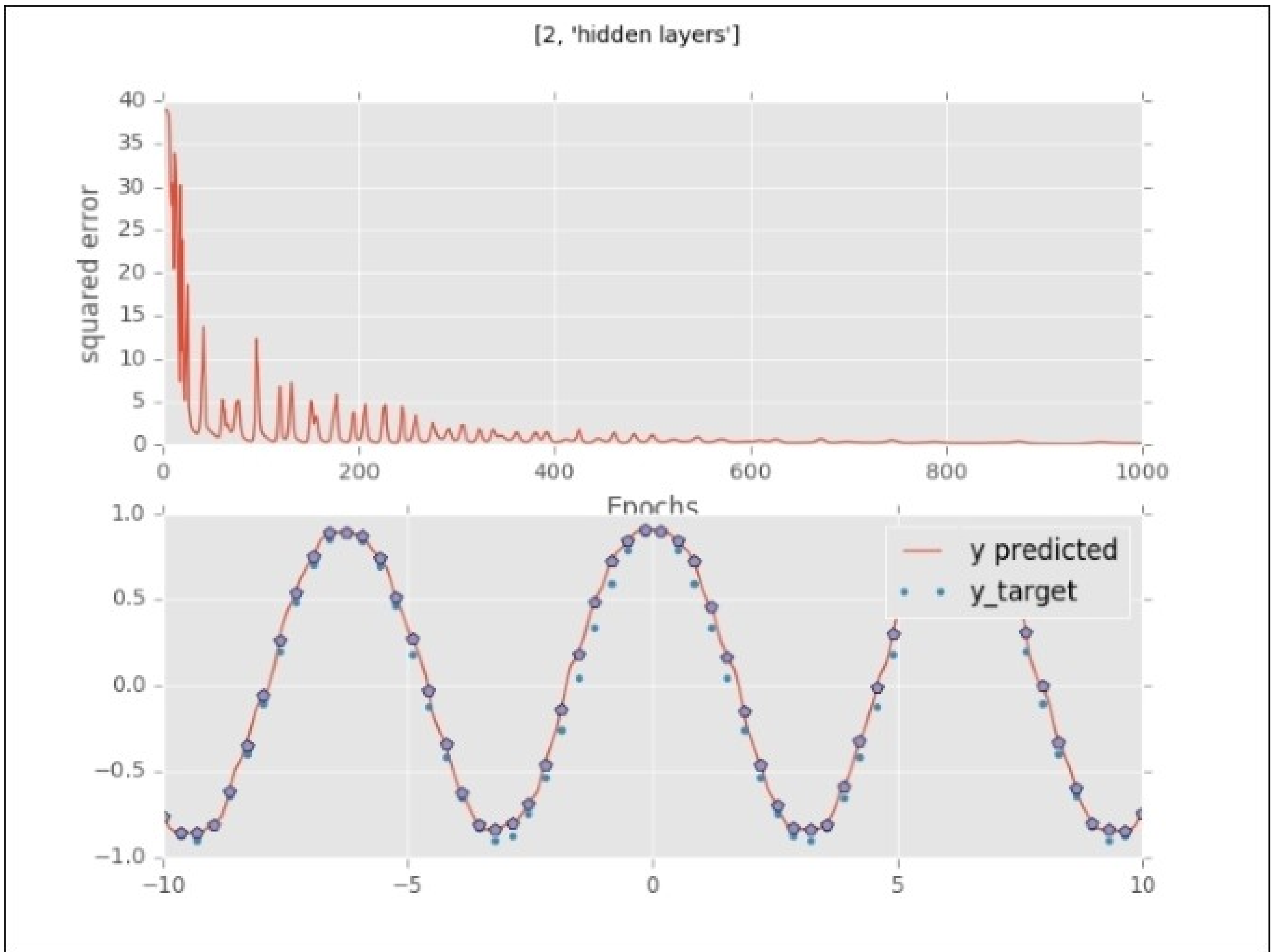
With zero hidden layers, the neural network projects a straight line through the target values. The error curve falls to a minimum quickly with a bad fit:



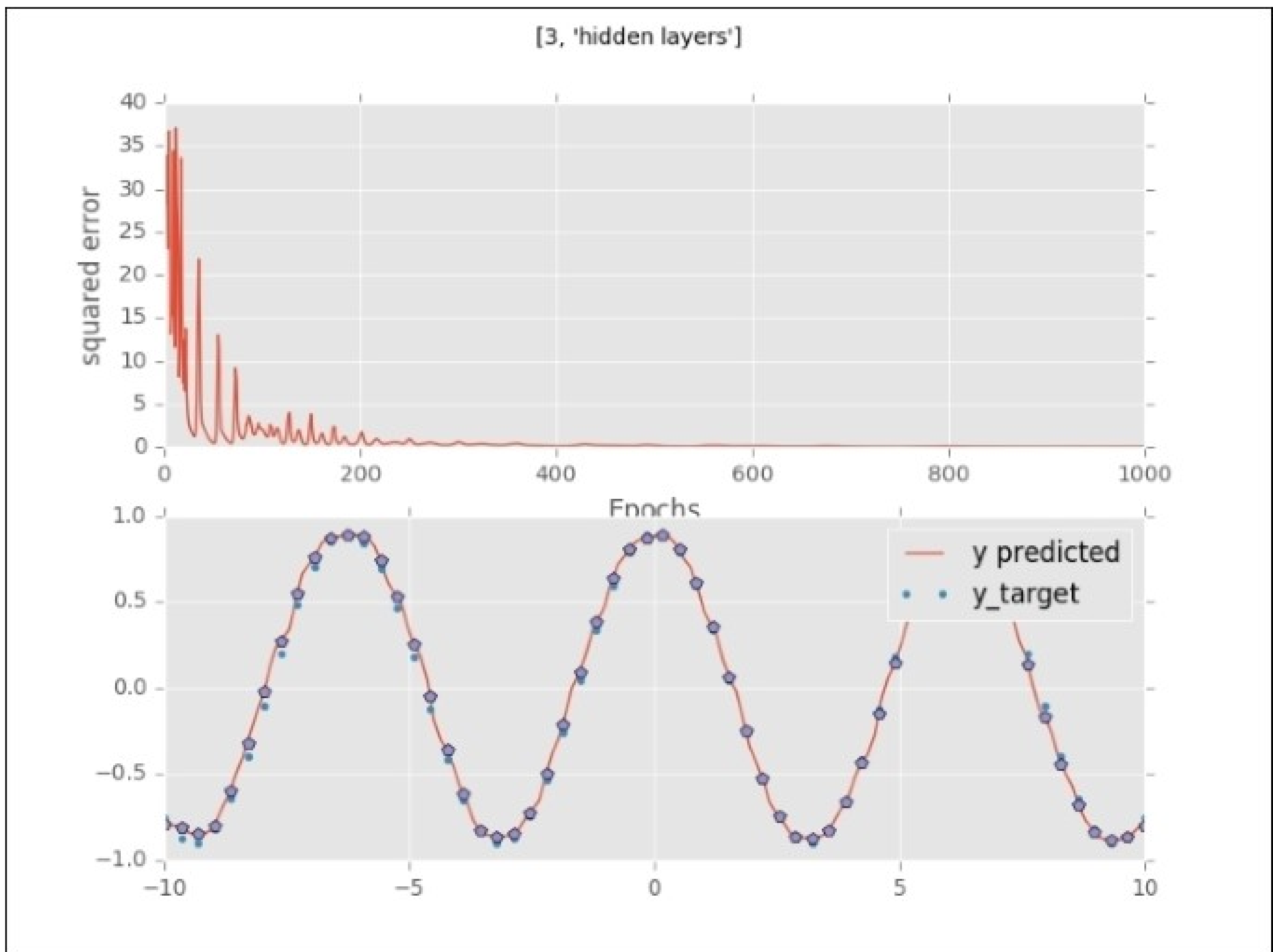
With one hidden layer, the network starts to approximate the target output. Watch how irregular the error curve is:



With two hidden layers, the neural network approximates the target value even more closely. The error curve drops faster and behaves less irregularly:



An almost perfect fit with three hidden layers. The error curve drops much faster (around **220** iterations).



The orange line in the upper plot is a visualization of how the error drops with each epoch (a full pass through the training set). It shows us that we need a certain number of passes through the training set to arrive at a global minimum. If you inspect this error curve more closely, you will see that the error curve behaves differently with each architecture. The lower plot (the dotted line) shows how the predicted values start to approximate the target values. With no hidden layer, the neural network is incapable of detecting nonlinear functions, but once we add hidden layers, the network starts to learn nonlinear functions and increasingly complex functions. In fact, neural networks can learn *any* possible function. This capability of learning every possible function is called the **universal approximation theorem**. We can modify this approximation by adding hidden neurons (units and layers) to the neural network. We do need to be cautious, however, that we don't overfit; adding a high amount of layers and units will lead to memorization of the training data instead of fitting a generalizable function. Quite often, too many layers in a network can be detrimental to predictive accuracy.

Choosing the right architecture

As we have seen, the combinatorial space of possible neural network architectures is almost infinite. So how can we know in advance which architecture will be suitable for our project? We need some sort of

heuristic or rule of thumb in order to design an architecture for a specific task. In the last section, we used a simple example with only one output and one feature. However, the recent wave of neural network architectures that we refer to as *deep learning* is very complex and it is crucial to be able to construct the right neural network architectures for any given task. As we have mentioned before, a typical neural network consists of the input layer, one or more hidden layers, and an output layer. Let's look at each layer of the architecture in detail so that we can have a sense of setting up the right architecture for any given task.

The input layer

When we mention the input layer, we are basically talking about the features that will be used as the input of the neural network. The preprocessing steps that are required are highly dependent on the shape and content of the data. If we have features that are measured on different scales, we need to rescale and normalize the data. In cases where we have a high amount of features, a dimension reduction technique such as PCA or SVD will become recommendable.

The following preprocessing techniques can be applied to inputs before learning:

- Normalization, scaling, & outlier detection
- Dimensionality reduction (SVD and factor analysis)
- Pretraining (autoencoders and Boltzmann machines)

We will cover each of these methods in the upcoming examples.

The hidden layer

How do we choose the amount of units in hidden layers? How many hidden layers do we add to the network? We have seen in the the previous example that a neural network without a hidden layer is incapable of learning a nonlinear function (both in curve fitting for regression and in decision boundaries with classification). So if there is a nonlinear pattern or decision boundary to project, we will need hidden layers. When it comes to selecting the amount of units in the hidden layer, we generally want fewer units in the hidden layer than the amount of units in the input layer and more units than the amount of output units:

- Preferably fewer hidden units than the amount of input features
- More units than the amount of output units (classes for classification)

Sometimes, when the target function is very complex in shape, there is an exception. In a case where we add more units than input dimensions, we add an **expansion** of the feature space. Networks with such layers are commonly referred to as **wide networks**.

Complex networks can learn more complex functions, but this does not mean that we can simply keep on stacking layers. It is advisory to keep the amount of layers in check because too many layers will cause problems with overfitting, higher CPU load, and even underfitting. Usually between one and four hidden layers will be sufficient.

Tip

Use preferably between one and four layers as a starting point.

The output layer

Each neural network has one output layer and, just like the input layer, is highly dependent on the structure of the data in question. For classification, we will generally use the `softmax` function. In this case, we should use the same amount of units as the amount of classes we predict.

Neural networks in action

Let's get some hands-on experience with training neural nets for classification. We will use sknn, the Scikit-learn wrapper for lasagne and Pylearn2. You can find out more about the package at https://github.com/aigamedev/scikit-neuralnetwork/.

We will use this tool because of its practical and Pythonic interface. It is a great introduction to more sophisticated frameworks like Keras.

The sknn library can run both on CPU or GPU, whichever you might prefer. Note that if you choose to utilize the GPU, sknn will operate on Theano:

```
For CPU (most stable) :
# Use the GPU in 32-bit mode, from sknn.platform import gpu32

from sknn.platform import cpu32, threading
# Use the CPU in 64-bit mode.from sknn.platform import cpu64

from sknn.platform import cpu64, threading

GPU:
# Use the GPU in 32-bit mode,
from sknn.platform import gpu32
# Use the CPU in 64-bit mode.
from sknn.platform import cpu64
```

Parallelization for sknn

We can utilize parallel processing in the following way, but this comes with a warning. It is not the most stable method:

```
from sknn.platform import cpu64, threading
```

We can specify Scikit-learn to utilize a specific amount of threads:

```
from sknn.platform import cpu64, threads2 #any desired amount of
threads
```

When you have specified the appropriate number of threads, you can parallelize your code by implementing `n_jobs=nthreads` in the cross-validation.

Now that we have covered the most important concepts and prepared our environment, let's implement a neural network.

For this example, we will use the convenient yet rather boring Iris dataset.

After this, we will apply preprocessing in the form of normalization and scaling and start building our model:

```
import numpy as np
from sklearn.datasets import load_iris
from sknn.mlp import Classifier, Layer
from sklearn import preprocessing
from sklearn.cross_validation import train_test_split
from sklearn import cross_validation
from sklearn import datasets


# import the familiar Iris data-set
iris = datasets.load_iris()
X_train, X_test, y_train, y_test = train_test_split(iris.data,
iris.target, test_size=0.2, random_state=0)
```

Here we apply preprocessing, normalization, and scaling to our inputs:

```
X_trainn = preprocessing.normalize(X_train, norm='l2')
X_testn = preprocessing.normalize(X_test, norm='l2')

X_trainn = preprocessing.scale(X_trainn)
X_testn = preprocessing.scale(X_testn)
```

Let's set up our neural network architecture and parameters. Let's start with a neural network with two layers. In the `Layer` part, we specify the settings of each layer independently. (We will see this method again in Tensorflow and Keras.) The Iris dataset consists of four features, but because in this particular case a *wide* neural network works quite well, we will use 13 units in each hidden layer. Note that sknn applies SGD by default:

```
clf = Classifier(
    layers=[
    Layer("Rectifier", units=13),
    Layer("Rectifier", units=13),
    Layer("Softmax")],    learning_rate=0.001,
    n_iter=200)

model1=clf.fit(X_trainn, y_train)
y_hat=clf.predict(X_testn)
scores = cross_validation.cross_val_score(clf, X_trainn, y_train,
cv=5)
```

```
print 'train mean accuracy %s' % np.mean(scores)
print 'vanilla sgd test %s' % accuracy_score(y_hat,y_test)


OUTPUT:]
train sgd mean accuracy 0.949909090909
sgd test 0.933333333333
```

A decent result on the training set, but we might be able to do better.

We talked about how Nesterov momentum can shorten the length toward the global minimum; let's run this algorithm with `nesterov` to see if we can increase accuracy and improve convergence:

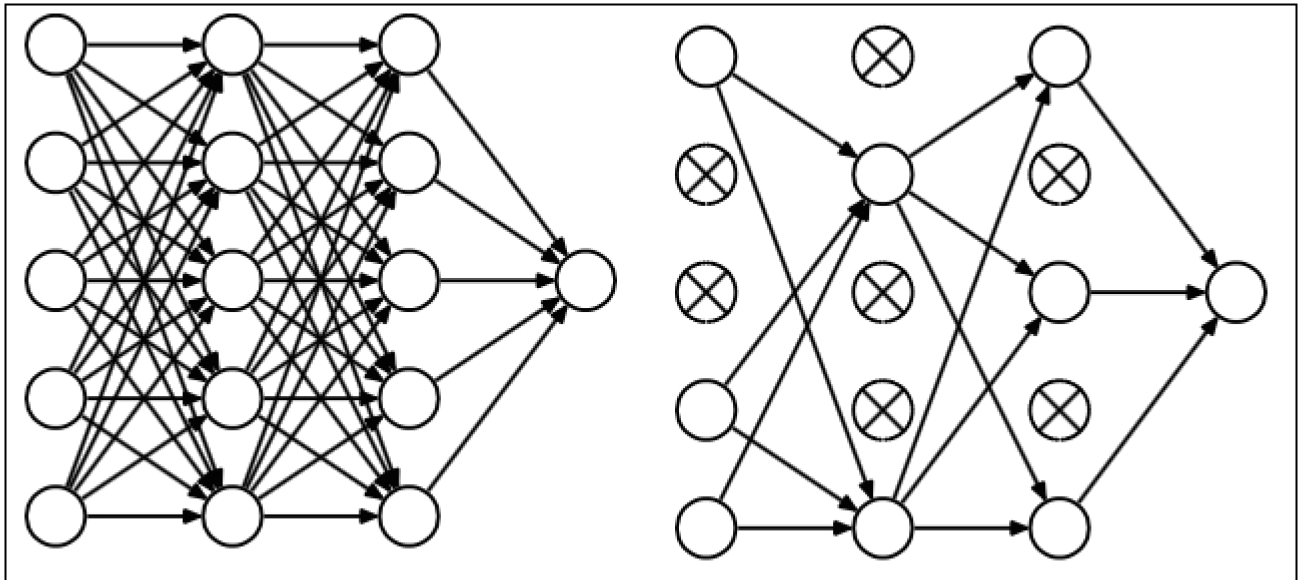
```
clf = Classifier(
    layers=[
    Layer("Rectifier", units=13),
    Layer("Rectifier", units=13),
    Layer("Softmax")],
learning_rate=0.001,learning_rule='nesterov',random_state=101,
    n_iter=1000)

model1=clf.fit(X_trainn, y_train)
y_hat=clf.predict(X_testn)
scores = cross_validation.cross_val_score(clf, X_trainn, y_train,
cv=5)
print 'Nesterov train mean accuracy %s' % np.mean(scores)
print 'Nesterov  test %s' % accuracy_score(y_hat,y_test)


OUTPUT]
Nesterov train mean accuracy 0.966575757576
Nesterov  test 0.966666666667
```

Our model is improved with Nesterov momentum in this case.

Neural networks and regularization

Even though we didn't overtrain our model in our last example, it is necessary to think about regularization strategies for neural networks. Three of the most widely-used ways in which we can apply regularization to a neural network are as follows:

- **L1** and **L2** regularization with weight decay as a parameter for the regularization strength
- **Dropout** means that deactivating units within the neural network at random can force other units in the network to take over



On the left hand, we see an architecture with dropout applied, randomly deactivating units in the network. On the right hand, we see an ordinary neural network (marked with X).

- **Averaging** or ensembling multiple neural networks (each with different settings)

Let's try dropout for this model and see if works:

```
clf = Classifier(
    layers=[
    Layer("Rectifier", units=13),
    Layer("Rectifier", units=13),
    Layer("Softmax")],
    learning_rate=0.01,
    n_iter=2000,
    learning_rule='nesterov',
    regularize='dropout', #here we specify dropout
    dropout_rate=.1,#dropout fraction of neural units in entire
network
    random_state=0)
model1=clf.fit(X_trainn, y_train)
```

```
scores = cross_validation.cross_val_score(clf, X_trainn, y_train,
cv=5)
print np.mean(scores)
y_hat=clf.predict(X_testn)
print accuracy_score(y_hat,y_test)
```

```
OUTPUT]
dropout train score 0.933151515152
dropout test score 0.866666666667
```

In this case, dropout didn't lead to satisfactory results so we should leave it out altogether. Feel free to experiment with other methods as well. Just change the `learning_rule` parameter and see what it does to the overall accuracy. The models that you can try are `sgd`, `momentum`, `nesterov`, `adagrad`, and `rmsprop`. From this example, you have learned that Nesterov momentum can increase the overall accuracy. In this case, `dropout` was not the best regularization method and was detrimental to model performance. Considering that this large number of parameters all interact and produce unpredictable results, we really need a tuning method. This is exactly what we are going to do in the next section.

Neural networks and hyperparameter optimization

As the parameter space of neural networks and deep learning models is so wide, optimization is a hard task and computationally very expensive. A wrong neural network architecture can be a recipe for failure. These models can only be accurate if we apply the right parameters and choose the right architecture for our problem. Unfortunately, there are only a few applications that provide tuning methods. We found that the best parameter tuning method at the moment is **randomized search**, an algorithm that iterates over the parameter space at random sparing computational resources. The sknn library is really the only library that has this option. Let's walk through the parameter tuning methods with the following example based on the wine-quality dataset.

In this example, we first load the wine dataset. Than we apply transformation to the data, from where we tune our model based on chosen parameters. Note that this dataset has 13 features; we specify the units within each layer to be between 4 and 20. We don't use mini-batch in this case; the dataset is simply too small:

```
import numpy as np
import scipy as sp
import pandas as pd
from sklearn.grid_search import RandomizedSearchCV
from sklearn.grid_search import GridSearchCV, RandomizedSearchCV
from scipy import stats
from sklearn.cross_validation import train_test_split
from sknn.mlp import  Layer, Regressor, Classifier as skClassifier


# Load data
df = pd.read_csv('http://archive.ics.uci.edu/ml/
machine-learning-databases/wine-quality/winequality-red.csv ' , sep
= ';')
X = df.drop('quality' , 1).values # drop target variable


y1 = df['quality'].values # original target variable
y = y1 <= 5 # new target variable: is the rating <= 5?

# Split the data into a test set and a training set
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=42)

print X_train.shape


max_net = skClassifier(layers= [Layer("Rectifier",units=10),
```

```
                                       Layer("Rectifier",units=10),
                                       Layer("Rectifier",units=10),
                                       Layer("Softmax")])
params={'learning_rate': sp.stats.uniform(0.001, 0.05,.1),
'hidden0__units': sp.stats.randint(4, 20),
'hidden0__type': ["Rectifier"],
'hidden1__units': sp.stats.randint(4, 20),
'hidden1__type': ["Rectifier"],
'hidden2__units': sp.stats.randint(4, 20),
'hidden2__type': ["Rectifier"],
'batch_size':sp.stats.randint(10,1000),
'learning_rule':["adagrad","rmsprop","sgd"]}
max_net2 =
RandomizedSearchCV(max_net,param_distributions=params,n_iter=25,cv=3,
scoring='accuracy',verbose=100,n_jobs=1,\
                             pre_dispatch=None)
model_tuning=max_net2.fit(X_train,y_train)



print "best score %s" % model_tuning.best_score_
print "best parameters %s" % model_tuning.best_params_


OUTPUT:]
[CV]  hidden0__units=11, learning_rate=0.100932183167,
hidden2__units=4, hidden2__type=Rectifier, batch_size=30,
hidden1__units=11, learning_rule=adagrad, hidden1__type=Rectifier,
hidden0__type=Rectifier, score=0.655914 -   3.0s
[Parallel(n_jobs=1)]: Done  74 tasks       | elapsed:  3.0min
[CV] hidden0__units=11, learning_rate=0.100932183167,
hidden2__units=4, hidden2__type=Rectifier, batch_size=30,
hidden1__units=11, learning_rule=adagrad, hidden1__type=Rectifier,
hidden0__type=Rectifier
[CV]  hidden0__units=11, learning_rate=0.100932183167,
hidden2__units=4, hidden2__type=Rectifier, batch_size=30,
hidden1__units=11, learning_rule=adagrad, hidden1__type=Rectifier,
hidden0__type=Rectifier, score=0.750000 -   3.3s
[Parallel(n_jobs=1)]: Done  75 tasks       | elapsed:  3.0min
[Parallel(n_jobs=1)]: Done  75 out of  75 | elapsed:  3.0min finished
best score 0.721366278222

best parameters {'hidden0__units': 14, 'learning_rate':
0.03202394348494512, 'hidden2__units': 19, 'hidden2__type':
'Rectifier', 'batch_size': 30, 'hidden1__units': 17,
'learning_rule': 'adagrad', 'hidden1__type': 'Rectifier',
'hidden0__type': 'Rectifier'}
```

Note

Warning: As the parameter space is searched at random, the results can be inconsistent.

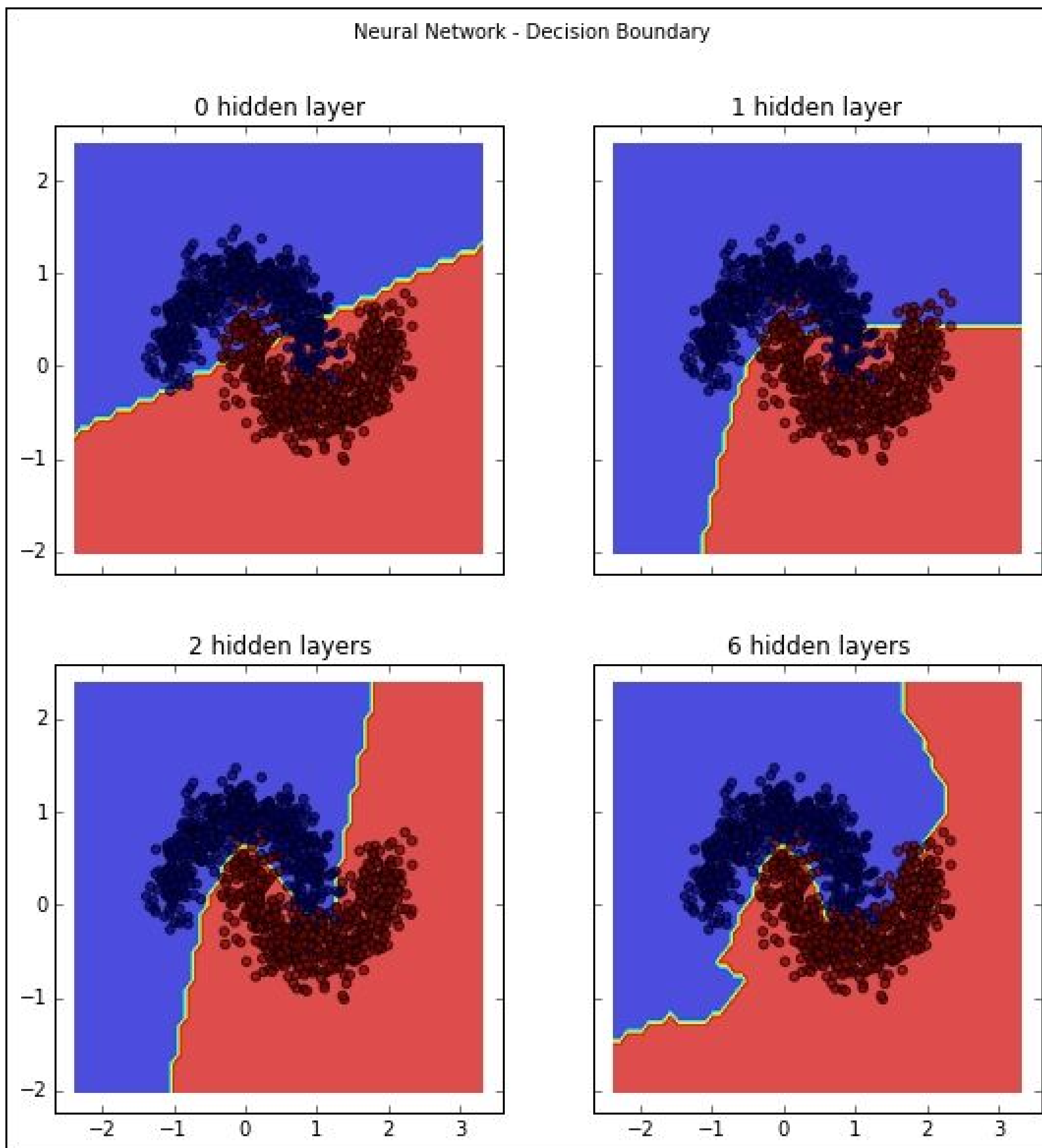
We can see that the best parameters for our model are, most importantly, the first layer with 14 units, the second layer contains 17 units, and the third layer contains 19 units. This is quite a complex architecture that we might never have been able to deduce ourselves, which demonstrates the importance of hyperparameter optimization.

Neural networks and decision boundaries

We have covered in the previous section that, by adding hidden units to a neural network, we can approximate the target function more closely. However, we haven't applied it to a classification problem. To do this, we will generate data with a nonlinear target value and look at how the decision surface changes once we add hidden units to our architecture. Let's see the universal approximation theorem at work! First, let's generate some non-linearly separable data with two features, set up our neural network architectures, and see how our decision boundaries change with each architecture:

```
%matplotlib inline
from sknn.mlp import Classifier, Layer
from sklearn import preprocessing
import numpy as np
import matplotlib.pyplot as plt
from sklearn import datasets
from itertools import product


X,y= datasets.make_moons(n_samples=500, noise=.2, random_state=222)
from sklearn.datasets import make_blobs

net1 = Classifier(
   layers=[
       Layer("Softmax")],random_state=222,
   learning_rate=0.01,
   n_iter=100)
net2 = Classifier(
   layers=[
       Layer("Rectifier", units=4),
       Layer("Softmax")],random_state=12,
   learning_rate=0.01,
   n_iter=100)
net3 =Classifier(
   layers=[
       Layer("Rectifier", units=4),
       Layer("Rectifier", units=4),
       Layer("Softmax")],random_state=22,
   learning_rate=0.01,
   n_iter=100)
net4 =Classifier(
   layers=[
       Layer("Rectifier", units=4),
       Layer("Rectifier", units=4),
       Layer("Rectifier", units=4),
       Layer("Rectifier", units=4),
       Layer("Rectifier", units=4),
```

```
       Layer("Rectifier", units=4),
       Layer("Softmax")],random_state=62,
   learning_rate=0.01,
   n_iter=100)

net1.fit(X, y)
net2.fit(X, y)
net3.fit(X, y)
net4.fit(X, y)


# Plotting decision regions
x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.1),
                     np.arange(y_min, y_max, 0.1))

f, arxxx = plt.subplots(2, 2, sharey='row',sharex='col', figsize=(8,
8))
plt.suptitle('Neural Network - Decision Boundary')
for idx, clf, ti in zip(product([0, 1], [0, 1]),
                        [net1, net2, net3,net4],
                        ['0 hidden layer', '1 hidden layer',
                         '2 hidden layers','6 hidden layers']):

   Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])
   Z = Z.reshape(xx.shape)

   arxxx[idx[0], idx[1]].contourf(xx, yy, Z, alpha=0.5)
   arxxx[idx[0], idx[1]].scatter(X[:, 0], X[:, 1], c=y, alpha=0.5)
   arxxx[idx[0], idx[1]].set_title(ti)

plt.show()
```



In this screenshot, we can see that, as we add hidden layers to the neural network, we can learn increasingly complex decision boundaries. An interesting side note is that the network with two layers produced the most accurate predictions.

Note

Note that the results might be different between runs.

Deep learning at scale with H2O

In previous sections, we covered neural networks and deep architectures running on a local computer and we found that neural networks are already highly vectorized but still computationally expensive. There is not much that we can do if we want to make the algorithm more scalable on a desktop computer other than utilizing Theano and GPU computing. So if we want to scale deep learning algorithms more drastically, we will need to find a tool that can run algorithms out-of-core instead of on a local CPU/GPU. H2O is, at this moment, the only open source out-of-core platform that can run deep learning algorithms quickly. It is also cross-platform; besides Python, there are APIs for R, Scala, and Java.

H2O is compiled on a Java-based platform developed for a wide range of data science-related tasks such as datahandling and machine learning. H2O runs on distributed and parallel CPUs in-memory so that data will be stored in the H2O cluster. The H2O platform—as of yet—has applications for **General Linear Models** (**GLM**), Random Forests, **Gradient Boosting Machines** (**GBM**), K Means, Naive Bayes, Principal Components Analysis, Principal Components Regression, and, of course our main focus for this chapter, Deep Learning.

Great, now we are ready to perform our first H2O out-of-core analysis.

Let's start the H2O instance and load a file in H2O's distributed memory system:

```
import sys
sys.prefix = "/usr/local"
import h2o

h2o.init(start_h2o=True)

Type this to get interesting information about the specifications of
your cluster.
Look at the memory that is allowed and the number of cores.

h2o.cluster_info()
```

This will look more or less like the following (slight differences might occur between trials and systems):

```
OUTPUT:]

Java Version: java version "1.8.0_60"
Java(TM) SE Runtime Environment (build 1.8.0_60-b27)
Java HotSpot(TM) 64-Bit Server VM (build 25.60-b23, mixed mode)


Starting H2O JVM and connecting: .................. Connection
successful!
------------------------------
```

```
----------------------------------------
H2O cluster uptime:             2 seconds 346 milliseconds
H2O cluster version:            3.8.2.3
H2O cluster name:
H2O_started_from_python**********nzb520
H2O cluster total nodes:        1
H2O cluster total free memory:  3.56 GB
H2O cluster total cores:        8
H2O cluster allowed cores:      8
H2O cluster healthy:            True
H2O Connection ip:              1**.***.***.***
H2O Connection port:            54321
H2O Connection proxy:
Python Version:                 2.7.10
-------------------------------
----------------------------------------
-------------------------------
----------------------------------------
H2O cluster uptime:             2 seconds 484 milliseconds
H2O cluster version:            3.8.2.3
H2O cluster name:
H2O_started_from_python_quandbee_nzb520
H2O cluster total nodes:        1
H2O cluster total free memory:  3.56 GB
H2O cluster total cores:        8
H2O cluster allowed cores:      8
H2O cluster healthy:            True
H2O Connection ip:              1**.***.***.***
H2O Connection port:            54321
H2O Connection proxy:
Python Version:                 2.7.10
-------------------------------
----------------------------------------
Sucessfully closed the H2O Session.
Successfully stopped H2O JVM started by the h2o python module.
```

Large scale deep learning with H2O

In H2O deep learning, the dataset that we will use to train is the famous MNIST dataset. It consists of pixel intensities of 28 x 28 images of handwritten digits. The training set has 70,000 training items with 784 features together with a label for each record containing the target label *digits*.

Now that we are more comfortable with managing data in H2O, let's perform a deep learning example.

In H2O, we don't have to transform or normalize the input data; it is standardized internally and automatically. Each feature is transformed into the N(0,1) space.

Let's import the famous handwritten digits image dataset MNIST from the Amazon server to the H2O cluster:

```
import h2o
h2o.init(start_h2o=True)
train_url ="https://h2o-public-test-data.s3.amazonaws.com/bigdata/
laptop/mnist/train.csv.gz"
test_url="https://h2o-public-test-data.s3.amazonaws.com/bigdata/
laptop/mnist/test.csv.gz"


train=h2o.import_file(train_url)
test=h2o.import_file(test_url)

train.describe()
test.describe()


y='C785'
x=train.names[0:784]
train[y]=train[y].asfactor()
test[y]=test[y].asfactor()

from h2o.estimators.deeplearning import H2ODeepLearningEstimator

model_cv=H2ODeepLearningEstimator(distribution='multinomial'

,activation='RectifierWithDropout',hidden=[32,32,32],
                                        input_dropout_ratio=.2,
                                        sparse=True,
                                        l1=.0005,
                                            epochs=5)
```

The output of this print model will provide a lot of detailed information. The first table that you will see is the following one. This provides all the specifics about the architecture of the neural network. You can see that we have used a neural network with an input dimension of 717 with three hidden layers (consisting of 32 units each) with softmax activation applied to the output layer and ReLU between the hidden layers:

```
model_cv.train(x=x,y=y,training_frame=train,nfolds=3)
print model_cv

OUTPUT]
```

```
Model Details
=============
H2ODeepLearningEstimator :  Deep Learning
Model Key:  DeepLearning_model_python_1463889677812_3

Status of Neuron Layers: predicting C785, 10-class classification, multinomial distribution, CrossEntropy loss, 25,418
weights/biases, 371.3 KB, 300,525 training samples, mini-batch size 1
```

	layer	units	type	dropout	l1	l2	mean_rate	rate_RMS	momentum	mean_weight	weight_RMS	mean_bias	bias_RMS
	1	717	Input	20.0									
	2	32	RectifierDropout	50.0	0.0005	0.0	0.0370441	0.1916480	0.0	-0.0061157	0.0612413	0.4243763	0.0918573
	3	32	RectifierDropout	50.0	0.0005	0.0	0.0004112	0.0002142	0.0	-0.0279839	0.1946866	0.7527754	0.2369041
	4	32	RectifierDropout	50.0	0.0005	0.0	0.0006548	0.0002914	0.0	-0.0397208	0.2000279	0.6407341	0.3597416
	5	10	Softmax		0.0005	0.0	0.0025825	0.0024549	0.0	-0.2988227	0.8903637	-1.0314634	0.8309324

```
ModelMetricsMultinomial: deeplearning
** Reported on train data. **

MSE: 0.142497867237
R^2: 0.982924289006
LogLoss: 0.455262748035
```

If you want a short overview of model performance, this is a very practical method.

In the following table, the most interesting metrics are the training classification error and validation classification error over each fold. You can easily compare these in case you want to validate your model:

```
print model_cv.scoring_history()
```

```
              timestamp    duration  training_speed    epochs  iterations  \
0   2016-05-22 06:09:35   0.000 sec            None  0.000000           0
1   2016-05-22 06:09:36   3.161 sec  30039 rows/sec  0.500650           1
2   2016-05-22 06:09:41   8.279 sec  40768 rows/sec  4.008217           8
3   2016-05-22 06:09:43  10.002 sec  40360 rows/sec  5.008750          10

   samples  training_MSE  training_r2  training_logloss  \
0        0           NaN          NaN               NaN
1    30039      0.434316     0.947955          1.154869
2   240493      0.163368     0.980423          0.507394
3   300525      0.142498     0.982924          0.455263

   training_classification_error
0                            NaN
1                       0.327284
2                       0.114081
3                       0.096430
```

Our training classification error of **.096430** and accuracy in the .907 range on the MNIST dataset is pretty good; it's almost as good as Yann LeCun's convolutional neural network submission.

H2O provides a convenient method to acquire validation metrics as well. We can do this by passing the validation dataframe to the cross-validation function:

```
model_cv.train(x=x,y=y,training_frame=train,validation_frame=test,nfo
lds=3)
print model_cv
```

Scoring History:

training_r2	training_logloss	training_classification_error	validation_MSE	validation_r2	validation_logloss	validation_classification_error
nan	nan	nan	nan	nan	nan	nan
0.9412354	1.2943441	0.3213827	0.4909360	0.9414521	1.2906389	0.3221
0.9857803	0.4101554	0.0889877	0.1234898	0.9852729	0.4234574	0.0954

In this case, we can easily compare **training_classification_error** (`.089`) with our **validation_classification_error** (`.0954`).

Maybe we can improve our score; let's use a hyperparameter optimization model.

Gridsearch on H2O

Considering that our previous model performed quite well, we will focus our tuning efforts on the architecture of our network. H2O's gridsearch function is quite similar to Scikit-learn's randomized search; namely, instead of searching the full parameter space, it iterates over a random list of parameters. First, we will set up a parameter list that we will pass to the gridsearch function. H2O will provide us with an output of each model and the corresponding score in the parameters' search:

```
hidden_opt = [[18,18],[32,32],[32,32,32],[100,100,100]]
# l1_opt = [s/1e6 for s in range(1,1001)]

# hyper_parameters = {"hidden":hidden_opt, "l1":l1_opt}
hyper_parameters = {"hidden":hidden_opt}

#important: here we specify the search parameters
#be careful with these, training time can explode (see max_models)
search_c = {"strategy":"RandomDiscrete",

"max_models":10, "max_runtime_secs":100,
```

```
"seed":222}


from h2o.grid.grid_search import H2OGridSearch

model_grid = H2OGridSearch(H2ODeepLearningEstimator,
hyper_params=hyper_parameters)

#We have added a validation set to the gridsearch method in order to
have a better #estimate of the model performance.


model_grid.train(x=x, y=y, distribution="multinomial", epochs=1000,
training_frame=train, validation_frame=test,
    score_interval=2, stopping_rounds=3,
stopping_tolerance=0.05,search_criteria=search_c)

print model_grid


# Grid Search Results for H2ODeepLearningEstimator:


OUTPUT]

deeplearning Grid Build Progress:
[##################################################] 100%
    hidden  \
0    [100, 100, 100]
1       [32, 32, 32]
2           [32, 32]
3           [18, 18]

    model_ids   logloss
0  Grid_DeepLearning_py_1_model_python_1464790287811_3_model_3
0.148162  ←------
1  Grid_DeepLearning_py_1_model_python_1464790287811_3_model_2
0.173675
2  Grid_DeepLearning_py_1_model_python_1464790287811_3_model_1
0.212246
3  Grid_DeepLearning_py_1_model_python_1464790287811_3_model_0
0.227706
```

We can see that our best architecture would be one with three layers with 100 units each. We can also clearly see that gridsearch increases training time substantially even on a powerful computing cluster

like the one that H2O operates on. So, even on H2O, we should use gridsearch with caution and be conservative with the parameters that are parsed in the model.

Now let's shutdown the H2O instance before we proceed:

```
h2o.shutdown(prompt=False)
```

Deep learning and unsupervised pretraining

In this section, we will introduce the most important concept in deep learning: how to improve learning by unsupervised pretraining. With unsupervised pretraining, we use neural networks to find latent features and factors in the data to later pass to a neural network. This method has the powerful capability of training networks to learn tasks that other machine learning methods can't, without handcrafting features. We will get into the specifics and introduce a new powerful library.

Deep learning with theanets

Scikit-learn's neural network application is especially interesting for parameter tuning purposes. Unfortunately, its capabilities for unsupervised neural network applications are limited. For the next subject, where we dive into more sophisticated deep learning methods, we need another library. In this chapter, we will focus on theanets. We love theanets because of its ease of use and stability; it's a very smooth and well-maintained package developed by Lief Johnson at the University of Texas. Setting up a neural network architecture works quite similarly to sklearn; namely, we instantiate a learning objective (classification or regression), specify the layers, and train it. For more information, you can visit http://theanets.readthedocs.org/en/stable/.

All you have to do is install theanets with `pip`:

```
$ pip install theanets
```

As theanets is built on top of Theano, you also need to have the Theano properly installed. Let's run a basic neural network model to see how theanets works. The resemblance with Scikit-learn will be obvious. Note that we use momentum for this example, and softmax is used by default in theanets so we don't have to specify it:

```
import climate # This package provides the reporting of iterations
from sklearn.metrics import confusion_matrix
import numpy as np
from sklearn import datasets
from sklearn.cross_validation import train_test_split
from sklearn.metrics import mean_squared_error
import theanets
import theano
import numpy as np
import matplotlib.pyplot as plt
import climate
from sklearn.cross_validation import train_test_split
import theanets
from sklearn.metrics import confusion_matrix
from sklearn import preprocessing
from sklearn.metrics import accuracy_score
from sklearn import datasets
climate.enable_default_logging()

digits = datasets.load_digits()
digits = datasets.load_digits()
X = np.asarray(digits.data, 'float32')

Y = digits.target

Y=np.array(Y, dtype=np.int32)
```

```
#X = (X - np.min(X, 0)) / (np.max(X, 0) + 0.0001)  # 0-1 scaling

X_train, X_test, y_train, y_test = train_test_split(X, Y,
                                                    test_size=0.2,
                                                    random_state=0)


# Build a classifier model with 64 inputs, 1 hidden layer with 100
units  and 10 outputs.
net = theanets.Classifier([64,100,10])

# Train the model using Resilient backpropagation and momentum.
net.train([X_train,y_train], algo='sgd', learning_rate=.001,
momentum=0.9,patience=0,
validate_every=N,
min_improvement=0.8)

# Show confusion matrices on the training/validation splits.
print(confusion_matrix(y_test, net.predict(X_test)))
print (accuracy_score(y_test, net.predict(X_test)))
```

OUTPUT]

```
[[27  0  0  0  0  0  0  0  0  0]
 [ 0 32  0  0  0  1  0  0  0  2]
 [ 0  1 34  0  0  0  0  1  0  0]
 [ 0  0  0 29  0  0  0  0  0  0]
 [ 0  0  0  0 29  0  0  1  0  0]
 [ 0  0  0  0  0 38  0  0  0  2]
 [ 0  1  0  0  0  0 43  0  0  0]
 [ 0  0  0  0  1  0  0 38  0  0]
 [ 0  2  1  0  0  0  0  0 36  0]
 [ 0  0  0  0  0  1  0  0  0 40]]
0.961111111111
```

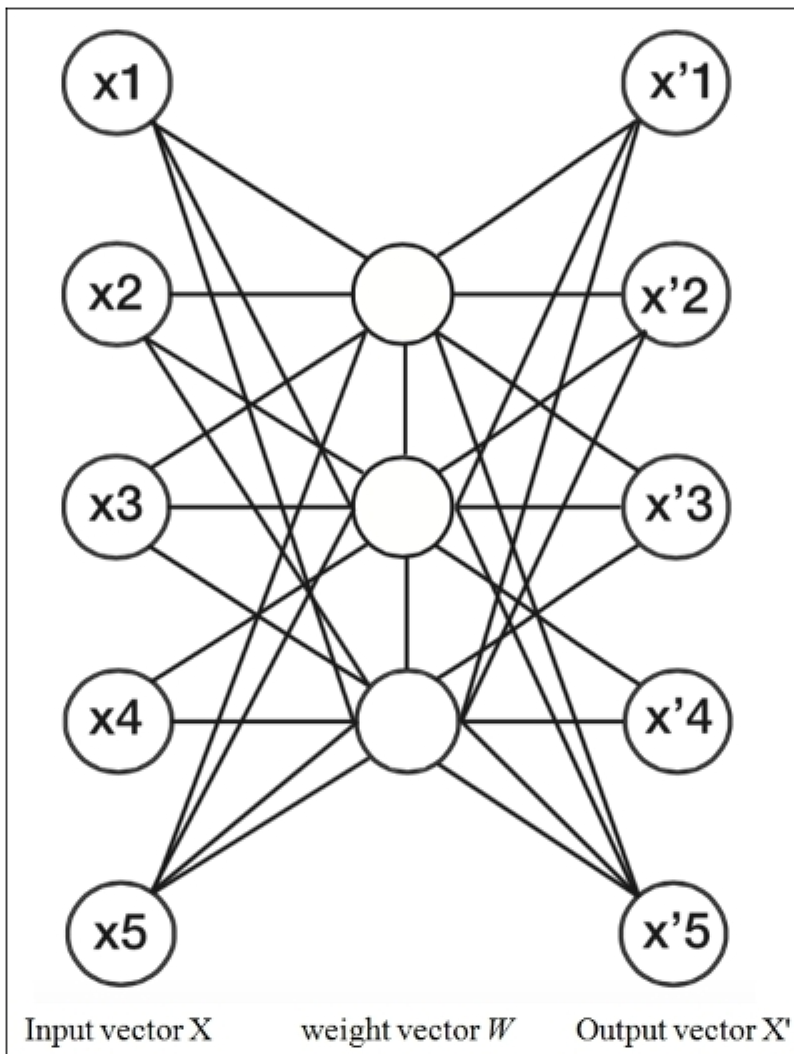
Autoencoders and unsupervised learning

Up until now, we discussed neural networks with multiple layers and a wide variety of parameters to optimize. The current generation of neural networks that we often refer to as deep learning is capable of more; it is capable of learning new features automatically so that very little feature engineering and domain expertise is required. These features are created by unsupervised methods on unlabeled data later to be fed into a subsequent layer in a neural network. This method is referred to as (unsupervised) **pretraining**. This approach has been proven to be highly successful in image recognition, language learning, and even vanilla machine learning projects. The most important and dominant technique in recent years is called **denoising autoencoders** and algorithms based on Boltzmann techniques. **Boltzmann machines**, which were the building blocks for **Deep Belief Networks** (**DBN**), have lately fallen out of favor in the deep learning community because they turned out to be hard to train and optimize. For this reason, we will focus only on autoencoders. Let's cover this important topic in small manageable steps.

Autoencoders

We try to find a function (*F*) that has an output as its input with the least possible error $F(x) \approx 'x$. This function is commonly referred to as the **identity function**, which we try to optimize so that x is as close as possible to $'x$. The difference between x and $'x$ is referred to as **reconstruction error**.

Let's look at a simple single-layer architecture to get an intuition of what's going on. We will see that these architectures are very flexible and need careful tuning:



Single-layer autoencoder architecture

It is important to understand that when we have fewer units in the hidden layer than the input space, we force the weights to compress the input data.

In this case, we have a dataset containing five features. In the middle is a hidden layer containing three units (W_{ij}). These units have the same property as the weight vector that we have seen in neural networks; namely, they are made up of weights that can be trained with backpropagation. With the output of the hidden layer, we get the feature representations as output by the same feedforward vector operations as we have seen with neural networks.

The process of calculating the vector $'x$ is quite similar to what we have seen with forward propagation by calculating the dot products of the weight vectors of each layer:

W=the weights

h_i=sigmoid($(W_1.x*x)+b_1(i,1)$)

$\hat{x}$=sigmoid($W_2.x*h_i)+b_2(i,1)$

The reconstruction error can be measured with the squared error or in cross-entropy form, which we have seen in so many other methods. In this case, $\hat{y}$ represents the reconstructed output and y the true input:

Cross entropy $L(x,y)=-\frac{1}{m}\sum_{i=1}^{m}[y_n\log\hat{y}_n+(1-y_n)\log(1-\hat{y}_n)]$

An important notion is that, with only one hidden layer, the dimensions in the data captured by the autoencoder model approximate the results of **Principal Component Analysis** (**PCA**). However, an autoencoder behaves much differently if there is non-linearity involved. The autoencoder will detect different latent factors that PCA will never be able to detect. Now that we know more about the architecture of the autoencoder and how we can calculate the error from its identity approximation, let's look at these **sparsity parameters** with which we compress the input.

You might ask: why do we even need this scarcity parameter? Can't we just run the algorithm to find the identity function and move on?

Unfortunately, it is not quite that simple. There are cases where the identity function projects the input almost perfectly but still fails to extract the latent dimensions of the input features. In that case, the function simply memorizes the input data instead of extracting meaningful features. We can do two things. First, we deliberately add noise to the signal (**denoising autoencoders**) and second, we introduce a sparsity parameter, forcing the deactivation of weakly-activated units. Let's first look at how sparsity works.

We discussed the activation threshold of a biological neuron; we can think of a neuron as being *active* if its potential is close to 1 or being *inactive* if its output value is close to 0. We can constraint the neurons to be inactive most of the time by increasing the activation threshold. We do this by decreasing the average activation probability of each neuron/unit. Looking at the following formula, we can see how we can minimize the activation threshold:

$$\hat{p}_j = \frac{1}{m}\sum_{i=1}^{m}\left[a_j^{(2)}\left(x^{(i)}\right)\right]$$

$\hat{p}_j$: The average activation threshold of each neuron in the hidden layer.

ρ: The desired activation threshold of the network, which we specify upfront. In most cases, this value is set at .05.

a: The weight vector of the hidden layers.

Here, we see an opportunity for optimization by penalizing a training round on the error rate between $\hat{p}_j$ and ρ.

In this chapter, we will not worry too much about the technical details of this optimization objective. In most packages, we can use a very simple instruction to do this (as we will see in the next example). The most important thing to understand is that with autoencoders, we have two main learning objectives: minimizing the error between the input vector x and output vector 'x by optimizing the identity function, and minimizing the difference between the desired activation threshold and average activation of each neuron in the network.

The second way in which we can force the autoencoder to detect latent features is by introducing noise in the model; this is where the name **denoising autoencoders** comes from. The idea is that by *corrupting* the input, we force the autoencoder to learn a more robust representation of the data. In the upcoming example, we will simply introduce Gaussian noise to the auto-encoder model.

Really deep learning with stacked denoising autoencoders – pretraining for classification

With this exercise, you will set yourself apart from the many people who talk about deep learning and the few who actually do it! Now we will apply an autoencoder to the mini version of the famous MNIST dataset, which can conveniently be loaded from within Scikit-learn. The dataset consists of pixel intensities of 28 x 28 images of handwritten digits. The training set has 1,797 training items with 64 features with a label for each record containing the target label *digits* from 0 to 9. So we have 64 features with a target variable consisting of 10 classes (digits from 0-9) to predict.

First, we train the stacked denoising autoencoder model with a sparsity of .9 and inspect the reconstruction error. We will use the results from deep learning research papers as a guideline for the settings. You can read this paper for more information (http://arxiv.org/pdf/1312.5663.pdf). However, we have some limitations because of the enormous computational load for these types of models. So, for this autoencoder, we use five layers with ReLU activation and compress the data from 64 features to 45 features:

```
model =
theanets.Autoencoder([64,(45,'relu'),(45,'relu'),(45,'relu'),(45,'rel
u'),(45,'relu'),64])
dAE_model=model.train([X_train],algo='rmsprop',input_noise=0.1,hidden
_l1=.001,sparsity=0.9,num_updates=1000)
X_dAE=model.encode(X_train)
X_dAE=np.asarray(X_dAE, 'float32')
:OUTPUT:
I 2016-04-20 05:13:37 downhill.base:232 RMSProp 2639 loss=0.660185
err=0.645118
I 2016-04-20 05:13:37 downhill.base:232 RMSProp 2640 loss=0.660031
```

```
err=0.644968
I 2016-04-20 05:13:37 downhill.base:232 validation 264 loss=0.660188
err=0.645123
I 2016-04-20 05:13:37 downhill.base:414 patience elapsed!
I 2016-04-20 05:13:37 theanets.graph:447 building computation graph
I 2016-04-20 05:13:37 theanets.losses:67 using loss: 1.0 *
MeanSquaredError (output out:out)
I 2016-04-20 05:13:37 theanets.graph:551 compiling feed_forward
function
```

Now we have the output from our autoencoder that we created from a new set of compressed features. Let's look closer at this new dataset:

```
X_dAE.shape
Output: (1437, 45)
```

Here, we can actually see that we have compressed the data from 64 to 45 features. The new dataset is less sparse (meaning fewer zeroes) and numerically more continuous. Now that we have our pretrained data from the autoencoder, we can apply a deep neural network to it for supervised learning:

```
#By default, hidden layers use the relu transfer function so we
don't need to specify #them. Relu is the best option for
auto-encoders.
# Theanets classifier also uses softmax by default so we don't need
to specify them.
net = theanets.Classifier(layers=(45,45,45,10))
autoe=net.train([X_dAE, y_train],
algo='rmsprop',learning_rate=.0001,batch_size=110,min_improvement=.00
01,momentum=.9,
nesterov=True,num_updates=1000)
## Enjoy the rare pleasure of 100% accuracy on the training set.
OUTPUT:
I 2016-04-19 10:33:07 downhill.base:232 RMSProp 14074 loss=0.000000
err=0.000000 acc=1.000000
I 2016-04-19 10:33:07 downhill.base:232 RMSProp 14075 loss=0.000000
err=0.000000 acc=1.000000
I 2016-04-19 10:33:07 downhill.base:232 RMSProp 14076 loss=0.000000
err=0.000000 acc=1.000000
```

Before we predict this neural network on the test set, it is important that we apply the autoencoder model that we have trained to the test set:

```
dAE_model=model.train([X_test],algo='rmsprop',input_noise=0.1,hidden_
l1=.001,sparsity=0.9,num_updates=100)
X_dAE2=model.encode(X_test)
X_dAE2=np.asarray(X_dAE2, 'float32')
```

Now let's check the performance on the test set:

```
final=net.predict(X_dAE2)
from sklearn.metrics import accuracy_score
print accuracy_score(final,y_test)
OUTPUT: 0.972222222222
```

We can see that the final accuracy of the model with auto-encoded features (.9722) outperforms the model without it (.9611).

Summary

In this chapter, we looked at the most important concepts behind deep learning together with scalable solutions.

We took away some of the black-boxiness by learning how to construct the right architecture for any given task and worked through the mechanics of forward propagation and backpropagation. Updating the weights of a neural network is a hard task, regular stochastic gradient descent can result in getting stuck in global minima or overshooting. More sophisticated algorithms like momentum, ADAGRAD, RPROP and RMSProp can provide solutions. Even though neural networks are harder to train than other machine learning methods, they have the power of transforming feature representations and can learn any given function (universal approximation theorem). We also dived into large scale deep learning with H2O, and even utilized the very hot topic of parameter optimization for deep learning.

Unsupervised pre-training with auto-encoders can increase accuracy of any given deep network and we walked through a practical example within the theanets framework to get there.

In this chapter, we primarily worked with packages built on top of the Theano framework. In the next chapter, we will cover deep learning techniques with packages built on top of the new open source framework Tensorflow.

Chapter 5. Deep Learning with TensorFlow

In this chapter, we will focus on TensorFlow and cover the following topics:

- Basic TensorFlow operations
- Machine learning from scratch with TensorFlow—regression, SGD classifier, and neural network
- Deep learning with SkFlow
- Incremental deep learning with large files
- Convolutional Neural Networks with Keras

The TensorFlow framework was introduced at the time of writing this book and already has proven to be a great addition to the machine learning landscape.

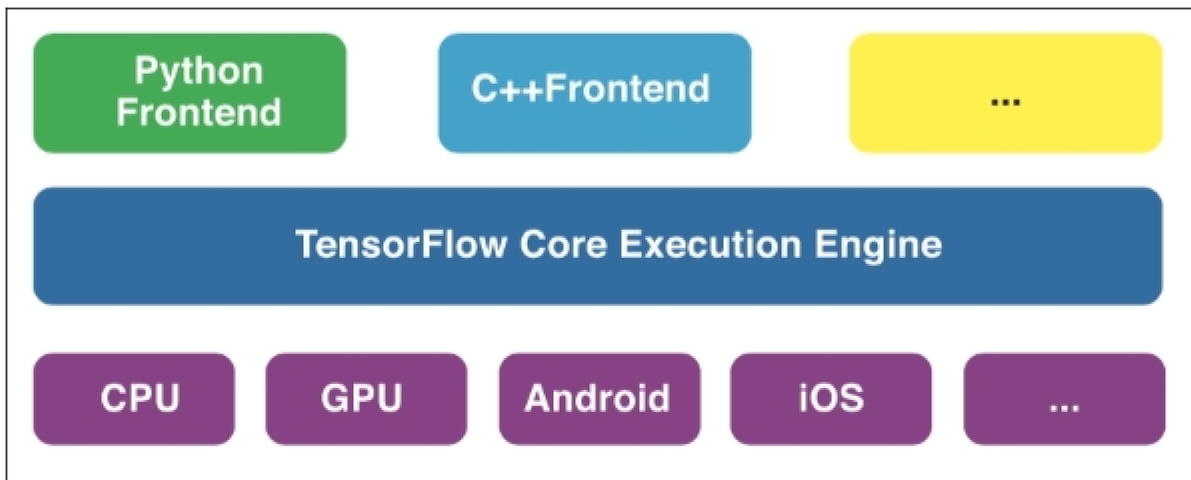
TensorFlow was started by the Google Brain Team consisting of most of the researchers that worked on important developments in deep learning in the recent decade (Geoffrey Hinton, Samy Bengio, and others). It is basically a next-generation development of an earlier generation of frameworks called DistBelief, a platform for distributed deep neural networks. Contrary to TensorFlow, **DistBelief** is not open source. Interesting examples of successful DistBelief projects are the reversed image search engine, Google deep dream, and speech recognition in Google apps. DistBelief enabled Google developers to utilize thousands of cores (both CPU and GPU) for distributed training.

TensorFlow is an improvement over DistBelief in that it is now completely open source and its programming language is less abstract. TensorFlow claims to be more flexible and has a wider range of applications. At the time of writing (late 2015), the TensorFlow framework is in its infancy and interesting lightweight packages built on top of TensorFlow have already emerged, as we will see later in this chapter.

Similarly to Theano, TensorFlow operates with symbolic computations on tensors; this means that most of its computations are based on vector-and matrix multiplications.

Regular programming languages define **variables** that contain values or characters to which operations can be applied to.

In symbolic programming languages such as Theano or TensorFlow, operations are structured around graphs instead of variables. This has computational advantages because they can be distributed and parallelized across computational units (GPU and CPU):

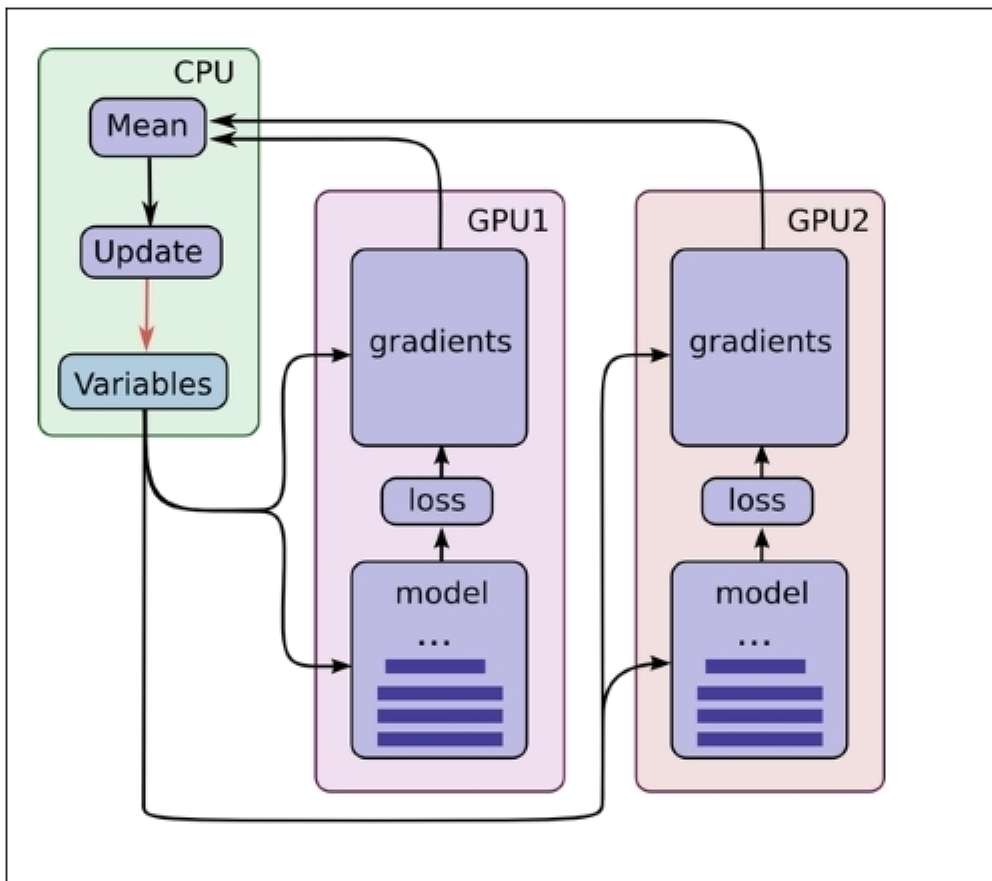


TensorFlow's architecture as introduced in November 2015

Tensorflow has the following features and applications:

- TensorFlow can be parallelized (horizontally) with multiple GPUs
- A development framework is also available for mobile deployment
- **TensorBoard** is a dashboard for visualizations (at a premature stage)
- It's a frontend for several programming languages (Python Go, Java, Lua, JavaScript, R, C++, and soon Julia)
- It provides integration for large scale solutions such as Spark and Google Cloud Platform (https://cloud.google.com/ml/)

The idea that tensor operations in a graph-like structure provide new ways of parallelized computations (so Google claims) can become quite clear with the following image:



A distributed processing architecture with TensorFlow

We can see from this image that each model can be assigned to separate GPUs. After which the average of the predictions is calculated from each model. Among other methods, this approach has been the central idea to train very large distributed neural networks on GPU clusters.

TensorFlow installation

The version of TensorFlow that we will use in this chapter is 0.8 so make sure that you install this version. As TensorFlow is in heavy development, small changes are due. We can install TensorFlow quite easily with `pip install`, independent of which operating system you use:

```
pip install tensorflow
```

If you already have previous versions installed, you can upgrade according to your operating system:

```
# Ubuntu/Linux 64-bit, CPU only:
$ sudo pip install --upgrade https://storage.googleapis.com/
tensorflow/linux/cpu/tensorflow-0.8.1-cp27-none-linux_x86_64.whl
```

```
# Ubuntu/Linux 64-bit, GPU enabled:
$ sudo pip install --upgrade https://storage.googleapis.com/
tensorflow/linux/gpu/tensorflow-0.8.1-cp27-none-linux_x86_64.whl

# Mac OS X, CPU only:
$ sudo easy_install --upgrade six
$ sudo pip install --upgrade https://storage.googleapis.com/
tensorflow/mac/tensorflow-0.8.1-cp27-none-any.whl
```

Now that TensorFlow is installed, you can test it in the terminal:

```
$python
import tensorflow as tf
hello = tf.constant('Hello, TensorFlow!')
sess = tf.Session()
print(sess.run(hello))
Output Hello, TensorFlow!
```

TensorFlow operations

Let's walk through some simple examples to get a feel for how it works.

An important distinction is that with TensorFlow, we first need to initialize the variables before we can apply operations to them. TensorFlow operates on a C++ backend to perform computations so, in order to connect to this backend, we need to instantiate a session first:

```
x = tf.constant([22,21,32], name='x')
d=tf.constant([12,23,43],name='x')
y = tf.Variable(x * d, name='y')
print(y)
```

Instead of providing the output vector of *x*d*, you will see something like this:

```
OUTPUT ]
<tensorflow.python.ops.variables.Variable object at 0x114a95710>
```

To actually produce the provided results of a computation from the C++ backend, we instantiate the session in the following way:

```
x = tf.constant([22,21,32], name='x')
d=tf.constant([12,23,43],name='d')
y = tf.Variable(x * d, name='y')

model = tf.initialize_all_variables()

with tf.Session() as session:
    session.run(model)
    print(session.run(y))
```

```
Output [ 264  483 1376]
```

Up until now, we have used variables directly, but to be more flexible with tensor operations, it can be convenient if we can assign data to a prespecified container. This way, we can perform operations on the computation graph without loading the data in-memory beforehand. In TensorFlow terminology, we then feed data into the graph through what are called *placeholders*. This is exactly where the resemblance with the Theano language becomes clear (see the Appendix, *Introduction to GPUs and Theano*).

These TensorFlow placeholders are simply containers of objects with certain prespecified settings and classes. So in order to perform operations on an object, we first create a placeholder for that object, together with its corresponding class (an integer in this case):

```
a = tf.placeholder(tf.int8)
b = tf.placeholder(tf.int8)
sess = tf.Session()
sess.run(a+b, feed_dict={a: 111, b: 222})
```

```
Output  77
```

Matrix multiplications will work like this:

```
matrix1 = tf.constant([[1, 2,32], [3, 4,2],[3,2,11]])

matrix2 = tf.constant([[21,3,12], [3, 56,2],[35,21,61]])

product = tf.matmul(matrix1, matrix2)

with tf.Session() as sess:
    result = sess.run(product)
    print result
```

```
OUTPUT

[[1147  787 1968]
 [ 145  275  166]
 [ 454  352  711]]
```

It is interesting to note that the output of the object `result` is a NumPy `ndarray` object that we can apply operations to outside of TensorFlow.

GPU computing

If we want to perform TensorFlow operations on a GPU, we only need to specify a device. Be warned; this only works with a properly installed, CUDA-compatible, NVIDIA GPU unit:

```
with tf.device('/gpu:0'):
    product = tf.matmul(matrix1, matrix2)
with tf.Session() as sess:
    result = sess.run(product)
    print result
```

If we want to utilize multiple GPUs, we need to assign a GPU device to a specific task:

```
matrix3 = tf.constant([[13, 21,53], [4, 3,6],[3,1,61]])
matrix4 = tf.constant([[13,23,32], [23, 16,2],[35,51,31]])

with tf.device('/gpu:0'):
    product = tf.matmul(matrix1, matrix2)
with tf.Session() as sess:
    result = sess.run(product)
    print result



with tf.device('/gpu:1'):
    product = tf.matmul(matrix3, matrix4)
with tf.Session() as sess:
    result = sess.run(product)
    print result
```

Linear regression with SGD

Now that we have covered the basics, we can start writing our first machine learning algorithm from scratch within the TensorFlow framework. Later, we will use more practical lightweight applications in higher abstractions on top of TensorFlow.

We will perform a very simple linear regression with stochastic gradient descent in order to get a sense of how training and evaluation works in TensorFlow. First, we will create some variables to work with in order to parse them in placeholders to contain those variables. We then feed `x` and `y` to a `cost` function and train the model with gradient descent:

```
import tensorflow as tf
import numpy as np


X = tf.placeholder("float") # create symbolic variables
Y = tf.placeholder("float")
X_train = np.asarray([1,2.2,3.3,4.1,5.2])
Y_train =  np.asarray([2,3,3.3,4.1,3.9,1.6])

def model(X, w):
    return tf.mul(X, w)
```

```
w = tf.Variable(0.0, name="weights")
y_model = model(X, w) # our predicted values

cost = (tf.pow(Y-y_model, 2)) # squared error cost

train_op = tf.train.GradientDescentOptimizer(0.01).minimize(cost)
#sgd optimization
sess = tf.Session()
init = tf.initialize_all_variables()
sess.run(init)

for trials in range(50):  #
    for (x, y) in zip(X_train, Y_train):
        sess.run(train_op, feed_dict={X: x, Y: y})

print(sess.run(w))

OUTPUT ]
0.844732
```

To summarize, we perform linear regression with SGD in the following way: first, we initialize the regression weights (coefficients), then in the second step, we set up the cost function to later train and optimize the function with gradient descent. In the end, we need to write a `for` loop in order to specify the amount of training rounds we want and calculate the final predictions. The same basic structure will become apparent in neural networks.

A neural network from scratch in TensorFlow

Now let's perform a neural network in the TensorFlow language and dissect the process.

We will also use the Iris dataset and some Scikit-learn applications to preprocess in this case:

```
import tensorflow as tf
import numpy as np
from sklearn import cross_validation
from sklearn.cross_validation import train_test_split
from sklearn.preprocessing import OneHotEncoder
from sklearn.utils import shuffle
from sklearn import preprocessing
import os
import pandas as pd
from datetime import datetime as dt
import logging

iris = datasets.load_iris()
X = np.asarray(iris.data, 'float32')
```

```
Y = iris.target

from sklearn import preprocessing
X= preprocessing.scale(X)
min_max_scaler = preprocessing.MinMaxScaler()
X = min_max_scaler.fit_transform(X)

lb = preprocessing.LabelBinarizer()
Y=lb.fit_transform(iris.target)
```

This is an important step. Neural networks in TensorFlow cannot work with target labels within a singular vector. Target labels need to be transformed into binarized features (some will know this as dummy variables) so that the neural network will work with a one versus all output:

```
X_train, x_test, y_train, y_test =
train_test_split(X,Y,test_size=0.3,random_state=22)


def init_weights(shape):
    return tf.Variable(tf.random_normal(shape, stddev=0.01))
```

Here, we can see the feedforward pass:

```
def model(X, w_h, w_o):
    h = tf.nn.sigmoid(tf.matmul(X, w_h))
    return tf.matmul(h, w_o)



X = tf.placeholder("float", [None, 4])
Y = tf.placeholder("float", [None, 3])
```

Here, we set up our layer architecture with one hidden layer:

```
w_h = init_weights([4, 4])
w_o = init_weights([4, 3])
py_x = model(X, w_h, w_o)


cost = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(py_x,
Y)) # compute costs
train_op =
tf.train.GradientDescentOptimizer(learning_rate=0.01).minimize(cost)
# construct an optimizer

predict_op = tf.argmax(py_x, 1)
```

```
sess = tf.Session()
init = tf.initialize_all_variables()
sess.run(init)


for i in range(500):
    for start, end in zip(range(0, len(X_train),1 ), range(1,
len(X_train),1)):
        sess.run(train_op, feed_dict={X: X_train[start:end], Y:
y_train[start:end]})
    if i % 100 == 0:
        print i, np.mean(np.argmax(y_test, axis=1) ==
                     sess.run(predict_op, feed_dict={X: x_test, Y:
y_test}))
```

```
OUTPUT:]
0 0.288888888889
100 0.666666666667
200 0.933333333333
300 0.977777777778
400 0.977777777778
```

The accuracy of this neural network is around .977% but can yield slightly different results across runs. It is more or less the benchmark for a neural network with a single hidden layer and vanilla SGD.

Like we saw in the previous examples, it is quite intuitive to implement an optimization method and set up the tensors. It is a lot more intuitive than when we do the same in NumPy. (See Chapter 4, *Neural Networks and Deep Learning*.) The downside at this moment is that evaluation and prediction requires a sometimes tedious `for` loop, whereas packages such as Scikit-learn can provide these methods with a simple line of script. Luckily, there are higher-level packages developed on top of TensorFlow that make training and evaluation a lot easier. One of those packages is SkFlow; as the name implies, it is a wrapper based on a scripting style that works just like Scikit-learn.

Machine learning on TensorFlow with SkFlow

Now that we have seen the basic operations of TensorFlow, let's dive into the higher-level applications built on top of TensorFlow to make machine learning a little more practical. SkFlow is the first application that we will cover. In SkFlow, we don't have to specify types and placeholders. We can load and manage data in the same way that we would do with Scikit-learn and NumPy. Let's install the package with `pip`.

The safest way is to install the package from GitHub directly:

```
$ pip install git+git://github.com/tensorflow/skflow.git
```

SkFlow has three main classes of learning algorithms: linear classifiers, linear regression, and neural networks. A linear classifier is basically a simple SGD (multi) classifier, and neural networks is where SkFlow excels. It provides relatively easy-to-use wrappers for very deep neural networks, recurrent networks, and Convolutional Neural Networks. Unfortunately, other algorithms such as Random Forest, gradient boosting, SVM, and Naïve Bayes are not yet implemented. However, there were discussions on GitHub about implementing a Random Forest algorithm in SkFlow that will probably be named tf_forest, which is an exciting development.

Let's apply our first multiclass classification algorithm in SkFlow. For this example, we will use the wine dataset—a dataset originally from the UCI machine learning repository. It consists of 13 features of continuous chemical metrics such as Magnesium, Alcohol, Malic acid, and so on. It's a light dataset with only 178 instances and a target feature with three classes. The target variable consists of three different cultivars. Wines are classified according to their respective cultivar (type of grapes used for the wine) using the chemical analysis of the thirteen chemical metrics. You can see that we load the data from a URL in the same way that we would do it when we work in a Scikit-learn environment:

```
import numpy as np
from sklearn.metrics import accuracy_score
import skflow
import urllib2
url = 'https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/
multiclass/wine.scale'
set1 = urllib2.Request(url)
wine = urllib2.urlopen(set1)


from sklearn.datasets import load_svmlight_file
X_train, y_train = load_svmlight_file(wine)
X_train=X_train.toarray()

from sklearn.cross_validation import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X_train,
y_train, test_size=0.30, random_state=4)
```

```
classifier =
skflow.TensorFlowLinearClassifier(n_classes=4,learning_rate=0.01,
optimizer='SGD',continue_training=True, steps=1000)
classifier.fit(X_train, y_train)
score = accuracy_score(y_train, classifier.predict(X_train))
d=classifier.predict(X_test)
print("Accuracy: %f" % score)

c=accuracy_score(d,y_test)
print('validation/test accuracy: %f' % c)
```

```
OUTPUT:
Step #1, avg. loss: 1.58672
Step #101, epoch #25, avg. loss: 1.45840
Step #201, epoch #50, avg. loss: 1.09080
Step #301, epoch #75, avg. loss: 0.84564
Step #401, epoch #100, avg. loss: 0.68503
Step #501, epoch #125, avg. loss: 0.57680
Step #601, epoch #150, avg. loss: 0.50120
Step #701, epoch #175, avg. loss: 0.44486
Step #801, epoch #200, avg. loss: 0.40151
Step #901, epoch #225, avg. loss: 0.36760
Accuracy: 0.967742
validation/test accuracy: 0.981481
```

By now, this method will be quite familiar; it is basically the same way a classifier in Scikit-learn would work. However, there are two important things to notice. With SkFlow, we can use NumPy and TensorFlow objects interchangeably so that we don't have to merge and convert objects in and out of tensor frames. This makes working with TensorFlow through a higher-level method like SkFlow much more flexible. The second thing to notice is that we applied the `toarray` method to the main data object. This is because the dataset is quite sparse (lots of zero entries), and TensorFlow is not able to process sparse data well.

Neural networks is where TensorFlow excels and in SkFlow, it is quite easy to train a neural network with multiple layers. Let's perform a neural network on the diabetes dataset. This dataset contains diabetes metrics (binary target) diagnostic features of pregnant females of over 21 years of age and of Pima heritage. The Pima Indians of Arizona have the highest reported prevalence of diabetes of any population in the world and therefore this ethnic group has been a voluntary subject of diabetes research. The dataset consists of the following features:

- Number of times pregnant
- Plasma glucose concentration at two hours in an oral glucose tolerance test
- Diastolic blood pressure (mm Hg)
- Triceps skin fold thickness (mm)
- 2-hour serum insulin (mu U/ml)

- Body mass index (weight in kg/(height in m)^2)
- Diabetes pedigree function
- Age (years)
- Class variable (0 or 1)

In this example, we first load and scale the data:

```
import tensorflow
import tensorflow as tf
import numpy as np
import urllib
import skflow
from sklearn.preprocessing import Normalizer
from sklearn import datasets, metrics, cross_validation
from sklearn.cross_validation import train_test_split
# Pima Indians Diabetes dataset (UCI Machine Learning Repository)
url = "http://archive.ics.uci.edu/ml/machine-learning-databases/
pima-indians-diabetes/pima-indians-diabetes.data"
# download the file
raw_data = urllib.urlopen(url)
dataset = np.loadtxt(raw_data, delimiter=",")
print(dataset.shape)
X = dataset[:,0:7]
y = dataset[:,8]
X_train, X_test, y_train, y_test = train_test_split(X, y,
                                                    test_size=0.2,
                                                    random_state=0)


from sklearn import preprocessing
X= preprocessing.scale(X)
min_max_scaler = preprocessing.MinMaxScaler()
X = min_max_scaler.fit_transform(X)
```

This step is very interesting; for neural networks to converge better, we can use more flexible decay rates. While training multilayer neural networks, it is usually helpful to decrease the learning rate over time. Generally speaking, when we have a too high learning rate, we might overshoot the optimum. On the other hand, when the learning rate is too low, we will waste computational resources and get stuck in local minima. Exponential decay is a method to dampen the learning rate over time so that it becomes more sensitive when it starts to approach a minimum. There are three common ways of implementing the learning rate decay; namely, step decay, 1/t decay, and exponential decay:

Exponential decay: $a=a_0e^{-kt}$

In this case, a is the learning rate, k is the hyperparameter, and t is the iteration.

In this example, we will use exponential decay because it seemed to have worked very well for this dataset. This is how we implement an exponential decay function (with TensorFlow's built-in `tf.train.exponential_decay` function):

```
def exp_decay(global_step):
    return tf.train.exponential_decay(
        learning_rate=0.01, global_step=global_step,
        decay_steps=steps, decay_rate=0.01)
```

We can now pass the decay function in the TensorFlow neural network model. For this neural network, we will provide a two-layer network, with the first layer consisting of five units and the second layer of four units. By default, SkFlow implements the ReLU activation function as we prefer it over the other ones (tanh, sigmoid, and so on) and so we stick with it.

Following this example, we can also implement optimization algorithms other than stochastic gradient descent. Let's implement an adaptive algorithm called Adam based on an article by Diederik Kingma and Jimmy Ba (http://arxiv.org/abs/1412.6980).

Adam, developed in the University of Amsterdam, stands for adaptive moment estimation. In the previous chapter, we saw how ADAGRAD works—by lowering the gradients over time as they move toward the (hopefully) global minimum. Adam also uses adaptive methods, but in combination with the idea of momentum training where previous gradient updates are taken into account:

```
steps = 5000
classifier = skflow.TensorFlowDNNClassifier(
    hidden_units=[5,4],
    n_classes=2,
    batch_size=300,
    steps=steps,
    optimizer='Adam',#SGD  #RMSProp
    learning_rate=exp_decay #here is the decay function
     )
classifier.fit(X_train,y_train)
score1a = metrics.accuracy_score(y_train,
classifier.predict(X_train))
print("Accuracy: %f" % score1a)
score1b = metrics.accuracy_score(y_test, classifier.predict(X_test))
print("Validation Accuracy: %f" % score1b)
```

```
OUTPUT
(768, 9)
Step #1, avg. loss: 12.83679
Step #501, epoch #167, avg. loss: 0.69306
Step #1001, epoch #333, avg. loss: 0.56356
Step #1501, epoch #500, avg. loss: 0.54453
Step #2001, epoch #667, avg. loss: 0.54554
Step #2501, epoch #833, avg. loss: 0.53300
```

```
Step #3001, epoch #1000, avg. loss: 0.53266
Step #3501, epoch #1167, avg. loss: 0.52815
Step #4001, epoch #1333, avg. loss: 0.52639
Step #4501, epoch #1500, avg. loss: 0.52721
Accuracy: 0.754072
Validation Accuracy: 0.740260
```

The accuracy is not so convincing; we might improve the accuracy by applying **Principal Component Analysis** (**PCA**) to the input. In this article by Stavros J Perantonis and Vassilis Virvilis from 1999 (http://rexa.info/paper/dc4f2babc5ca4534b435280aec32f5816ddb53b0), it has been proposed that this diabetes dataset benefits well from a PCA dimension reduction before passing in the neural network. We will use the Scikit-learn pipeline method for this dataset:

```
from sklearn.decomposition import PCA
from sklearn import linear_model, decomposition, datasets
from sklearn.pipeline import Pipeline
from sklearn.metrics import accuracy_score

pca = PCA(n_components=4,whiten=True)

lr = pca.fit(X)
classifier = skflow.TensorFlowDNNClassifier(
    hidden_units=[5,4],
    n_classes=2,
    batch_size=300,
    steps=steps,
    optimizer='Adam',#SGD  #RMSProp
    learning_rate=exp_decay
     )


pipe = Pipeline(steps=[('pca', pca), ('NNET', classifier)])

X_train, X_test, Y_train, Y_test = train_test_split(X, y,
                                                     test_size=0.2,
                                                     random_state=0)
pipe.fit(X_train, Y_train)



score2 = metrics.accuracy_score(Y_test, pipe.predict(X_test))
print("Accuracy Validation, with pca: %f" % score2)


OUTPUT:
Step #1, avg. loss: 1.07512
Step #501, epoch #167, avg. loss: 0.54236
```

```
Step #1001, epoch #333, avg. loss: 0.50186
Step #1501, epoch #500, avg. loss: 0.49243
Step #2001, epoch #667, avg. loss: 0.48541
Step #2501, epoch #833, avg. loss: 0.46982
Step #3001, epoch #1000, avg. loss: 0.47928
Step #3501, epoch #1167, avg. loss: 0.47598
Step #4001, epoch #1333, avg. loss: 0.47464
Step #4501, epoch #1500, avg. loss: 0.47712
Accuracy Validation, with pca: 0.805195
```

We have been able to improve the performance of the neural network quite a bit with that simple PCA preprocessing step. We went from seven features to a reduction of four dimensions, thus four features. PCA generally smoothens the signal by zero-centering the features, reducing the feature space using only the vectors containing the highest *eigenvalue*. **Whitening** makes sure that the features are transformed into zero-correlated ones. This results in a smoother signal and smaller feature set enabling the neural network to converge faster. See the Chapter 7, *Unsupervised Learning at Scale* for a more detailed explanation of PCA.

Deep learning with large files – incremental learning

Until now, we have dealt with some TensorFlow operations and machine learning techniques on SkFlow on relatively small datasets. However, this book is about large scale and scalable machine learning; what has the TensorFlow framework to offer us in that regard?

Until recently, parallel computation was in its infancy and not stable enough to be covered in this book. Multi-GPU computing is not accessible to readers without CUDA-compatible NVIDIA cards. Large scale cloud services (https://cloud.google.com/products/machine-learning/) or Amazon EC2 come with a considerable fee. This leaves only one way we can scale our project—by incremental learning.

Generally speaking, any file size exceeding about 25% of the available RAM of a computer will cause memory overload problems. So if you have a 2 GB computer and want to apply machine learning solutions to a 500 MB file, it is time to start thinking about ways to bypass memory consumption.

In order to prevent memory overload, we advise an out-of-core learning method that breaks the data down into smaller chunks to incrementally train and update models. The partial fit methods in Scikit-learn that we covered in Chapter 2, *Scalable Learning in Scikit-learn*, are examples of this.

SkFlow also provides a great incremental learning method for all its machine learning models just like the partial fit method in Scikit-learn. In this section, we are going to use a deep learning classifier incrementally because we think it is the most exciting one.

In this section, we will use two strategies for our scalable and out-of-core deep learning project; namely, incremental learning and random subsampling.

First, we generate some data, then we build a subsample function where we can draw random subsamples from that dataset and incrementally train a deep learning model on these subsets:

```
import numpy as np
import pandas as pd
import skflow
from sklearn.datasets import make_classification
import random
from sklearn.cross_validation import train_test_split
import gc
import tensorflow as tf
from sklearn.metrics import accuracy_score
```

First, we are going to generate some example data and write it to disk:

```
X, y = make_classification(n_samples=5000000,n_features=10,
n_classes=4,n_informative=6,random_state=222,n_clusters_per_class=1)
X_train, X_test, y_train, y_test = train_test_split(X,y,
test_size=0.2, random_state=22)

Big_trainm=pd.DataFrame(X_train,y_train)
Big_testm = pd.DataFrame(X_test,y_test)


Big_trainm.to_csv('lsml-Bigtrainm', sep=',')
Big_testm.to_csv('lsml-Bigtestm', sep=',')
```

Let's free up memory by deleting all the objects that we created.

With `gc.collect` we force Python's garbage collector to empty memory:

```
del(X,y,X_train,y_train,X_test)
gc.collect
```

Here, we create a function that draws random subsamples from disk. Note that we use a sample fraction of ⅓. We could use smaller fractions but we also need to adjust two important things if we do so. First, we need to match the batch size of the deep learning model so that the batch size never exceeds the sample size. Second, we need to adjust our amount of epochs in our `for` loop in such a way that we make sure that the largest portion of the training data is used to train the model:

```
import pandas as pd
import random
def sample_file():
    global skip_idx
    global train_data
    global X_train
    global y_train
    big_train='lsml-Bigtrainm'
```

Count the number of rows in the entire set:

```
num_lines = sum(1 for i in open(big_train))
```

We use one-third fraction of the training set:

```
size = int(num_lines / 3)
```

Skip indexes and keep indices:

```
skip_idx = random.sample(range(1, num_lines), num_lines - size)
train_data = pd.read_csv(big_train, skiprows=skip_idx)
X_train=train_data.drop(train_data.columns[[0]], axis=1)
y_train = train_data.ix[:,0]
```

We saw weight decay in a previous section; we will use it again here:

```
def exp_decay(global_step):
    return tf.train.exponential_decay(
        learning_rate=0.01, global_step=global_step,
        decay_steps=steps, decay_rate=0.01)
```

Here, we set up our neural network DNN classifier with three hidden layers with `5`, `4`, and `4` units respectively. Note that we set the batch size to `300`, which means that we use 300 training cases in each epoch. This also helps prevent memory from overloading:

```
steps = 5000
clf = skflow.TensorFlowDNNClassifier(
    hidden_units=[5,4,4],
    n_classes=4,
    batch_size=300,
    steps=steps,
    optimizer='Adam',
    learning_rate=exp_decay
     )
```

Here, we set our amount of subsamples to three (`epochs=3`). This means that we incrementally train our deep learning model on three consecutive subsamples:

```
epochs=3
for i in range(epochs):
    sample_file()
    clf.partial_fit(X_train,y_train)


test_data = pd.read_csv('lsml-Bigtestm',sep=',')
X_test=test_data.drop(test_data.columns[[0]], axis=1)
y_test = test_data.ix[:,0]
score = accuracy_score(y_test, clf.predict(X_test))
```

```
print score
```

OUTPUT

```
Step #501, avg. loss: 0.55220
Step #1001, avg. loss: 0.31165
Step #1501, avg. loss: 0.27033
Step #2001, avg. loss: 0.25250
Step #2501, avg. loss: 0.24156
Step #3001, avg. loss: 0.23438
Step #3501, avg. loss: 0.23113
Step #4001, avg. loss: 0.23335
Step #4501, epoch #1, avg. loss: 0.23303
Step #1, avg. loss: 2.57968
Step #501, avg. loss: 0.57755
Step #1001, avg. loss: 0.33215
Step #1501, avg. loss: 0.27509
Step #2001, avg. loss: 0.26172
Step #2501, avg. loss: 0.24883
Step #3001, avg. loss: 0.24343
Step #3501, avg. loss: 0.24265
Step #4001, avg. loss: 0.23686
Step #4501, epoch #1, avg. loss: 0.23681
0.929022
```

We managed to get an accuracy of `.929` on the test set within a very manageable training time and without overloading our memory, considerably faster than if we would have trained the same model on the entire dataset at once.

Keras and TensorFlow installation

Previously, we have seen practical examples of the SkFlow wrapper for TensorFlow applications. For a more sophisticated approach to neural networks and deep learning where we have more control over parameters, we propose Keras (http://keras.io/). This package was originally developed within the Theano framework, but recently is also adapted to TensorFlow. This way, we can use Keras as a higher abstract package on top of TensorFlow. Keep in mind though that Keras is slightly less straightforward than SkFlow in its methods. Keras can run on both GPU and CPU, which makes this package really flexible when porting it to different environments.

Let's first install Keras and make sure that it utilizes the TensorFlow backend.

Installation works simply using `pip` in the command line:

```
$pip install Keras
```

Keras is originally built on top of Theano, so we need to specify Keras to utilize TensorFlow instead. In order to do this, we first need to run Keras once on its default platform, Theano.

First, we need to run some Keras code to make sure that all the library items are properly installed. Let's train a basic neural network and get introduced to some key concepts.

Out of convenience, we will make use of generated data with Scikit-learn consisting of four features and a target variable consisting of three classes. These dimensions are very important because we need them to specify the architecture of the neural network:

```
import numpy as np
import keras
from sklearn.datasets import make_classification
from sklearn.cross_validation import train_test_split
from sklearn.preprocessing import OneHotEncoder
from keras.utils import np_utils, generic_utils
from keras.models import Sequential
from keras.layers import Dense, Dropout, Activation
from keras.optimizers import SGD

nb_classes=3
X, y = make_classification(n_samples=1000, n_features=4,
n_classes=nb_classes,n_informative=3, n_redundant=0,
random_state=101)
```

Now that we have specified the variables, it is important to convert the target variable to a one-hot encoding array (just like we did in TensorFlow). Otherwise, Keras won't be able to compute the one versus all target outputs. For Keras, we want to use `np_utils` instead of sklearn's one-hot encoder. This is how we will use it:

```
y=np_utils.to_categorical(y,nb_classes)
print y
```

Our array of `y` will look like this:

```
OUTPUT]
array([[ 1.,  0.,  0.],
       [ 0.,  0.,  1.],
       [ 0.,  0.,  1.],
       …,
```

Now let's split the data into test and train:

```
x_train, x_test, y_train, y_test = train_test_split(X,
y,test_size=0.30, random_state=222)
```

This is where we start to give form to the neural network architecture that we have in mind. Let's start a two-hidden layer neural network with `relu` activation and three units in each hidden layer. Our first layer has four inputs because we have four features in this case. After that, we add the hidden layers with three units, hence (`model.add(dense(3))`).

Like we have seen before, we will use a `softmax` function to pass the network to the output layer:

```
model = Sequential()
model.add(Dense(4, input_shape=(4,)))
model.add(Activation('relu'))
model.add(Dense(3))
model.add(Activation('relu'))
model.add(Dense(3))
model.add(Activation('softmax'))
```

First, we specify our SGD function, where we implement the most important parameters that are familiar to us by now, namely:

- **lr**: The learning rate.
- **decay**: The decay function to decay the learning rate. Do not confuse this with weight decay, which is a regularization parameter.
- **momentum**: We use this to prevent from getting stuck in local minima.
- **nesterov**: This is a Boolean that specifies whether we want to use nesterov momentum and is only applicable if we have specified an integer for the momentum parameter. (Refer to Chapter 4, *Neural Networks and Deep Learning*, for a more detailed explanation.)
- **optimizer**: Here, we will specify our optimization algorithm of choice (consisting of SGD, RMSProp, ADAGRAD, Adadelta, and Adam).

Let us see the following code snippet:

```
#We use this for reproducibility
seed = 22
```

```
np.random.seed(seed)

model = Sequential()
model.add(Dense(4, input_shape=(4,)))
model.add(Activation('relu'))
model.add(Dense(3))
model.add(Activation('relu'))
model.add(Dense(3))
model.add(Activation('softmax'))

sgd = SGD(lr=0.01, decay=1e-6, momentum=0.9, nesterov=True)
model.compile(loss='categorical_crossentropy', optimizer=sgd)
model.fit(x_train, y_train, verbose=1, batch_size=100,
nb_epoch=50,show_accuracy=True,validation_data=(x_test, y_test))
time.sleep(0.1)
```

In this case, we have used `batch_size` of `100`, which means that we have used minibatch gradient descent with 100 training examples in each epoch. In this model, we have used `50` training epochs. This will give you the following output:

```
OUTPUT:

acc: 0.8129 - val_loss: 0.5391 - val_acc: 0.8000
Train on 700 samples, validate on 300 samples
```

In the last model where we used SGD with nesterov, we couldn't improve our score no matter how many epochs we used for its training.

In order to increase accuracy. it is advisable to try out other optimization algorithms. We have already used the Adam optimization method successfully before, so let's use it again here and see if we can increase the accuracy. As adaptive learning rates such as Adam, lower the learning rate over time, it requires more epochs to arrive at an optimum solution. Therefore, in this example, we will set the amount of epochs to 200:

```
adam=keras.optimizers.Adam(lr=0.01)
model.compile(loss='categorical_crossentropy', optimizer=adam)
model.fit(x_train, y_train, verbose=1, batch_size=100,
nb_epoch=200,show_accuracy=True,validation_data=(x_test, y_test))
time.sleep(0.1)
```

```
OUTPUT:
Epoch 200/200
700/700 [==============================] - 0s - loss: 0.3755 - acc:
0.8657 - val_loss: 0.4725 - val_acc: 0.8200
```

We now have managed to achieve a convincing improvement from 0.8 to 0.82 with the Adam optimization algorithm.

For now, we have covered the most important elements of neural networks in Keras. Let's now proceed setting up Keras so that it will utilize the TensorFlow framework. By default, Keras will use the Theano backend. In order to instruct Keras to work on TensorFlow, we need to first locate the Keras folder in the packages folder:

```
import os
print keras.__file__
```

Your path might look different:

```
Output: /Library/Python/2.7/site-packages/keras/__init__.pyc
```

Now that we have located the package folder of Keras, we need to look for the `~/.keras/keras.json` file.

There is a piece of script in this file that looks like this:

```
{"epsilon": 1e-07, "floatx": "float32", "backend": "theano"}
```

You simply need to change `"backend":"theano"` to `"backend":"tensorflow"`, resulting in the following:

```
{"epsilon": 1e-07, "floatx": "float32", "backend": "tensorflow"}
```

If, for some reason, the `.json` file is not present in the Keras folder, that is, `/Library/Python/2.7/site-packages/keras/`, you can just copy paste this to a text editor:

```
{"epsilon": 1e-07, "floatx": "float32", "backend": "tensorflow"}
```

Save it as a `.json` file and put it in the `keras` folder.

To test if the TensorFlow environment is properly utilized from within TensorFlow, we can type the following:

```
from keras import backend as K
input = K.placeholder(shape=(4, 4, 5))
# also works:
input = K.placeholder(shape=(None, 2, 5))
# also works:
input = K.placeholder(ndim=2)
```

```
OUTPUT:

Using Theano backend.
```

Some users might get no output at all, which is fine. Your TensorFlow backend should be ready to use.

Convolutional Neural Networks in TensorFlow through Keras

Between this and the previous chapter, we have come quite a long way covering the most important topics in deep learning. We now understand how to construct architectures by stacking multiple layers in a neural network and how to discern and utilize backpropagation methods. We also covered the concept of unsupervised pretraining with stacked and denoising autoencoders. The next and really exciting step in deep learning is the rapidly evolving field of **Convolutional Neural Networks** (**CNN**), a method of building multilayered, locally connected networks. CNNs, commonly referred to as **ConvNets**, are so rapidly evolving at the time of writing this book that we literally had to rewrite and update this chapter within a month's timeframe. In this chapter, we will cover the most fundamental and important concepts behind CNNs so that we will be able to run some basic examples without becoming overwhelmed by the sometimes enormous complexity. However, we won't be able to do justice fully to the enormous theoretical and computational background so this paragraph provides a practical starting point.

The best way to understand CNNs conceptually is to go back in history, start with a little bit of cognitive neuroscience, and look at Huber and Wiesel's research on the visual cortex of cats. Huber and Wiesel recorded neuro-activations of the visual cortex in cats while measuring neural activity by inserting microelectrodes in the visual cortex of the brain. (Poor cats!) They did this while the cats were watching primitive images of shapes projected on a screen. Interestingly, they found that certain neurons responded only to contours of specific orientation or shape. This led to the theory that the visual cortex is composed of local and orientation-specific neurons. This means that specific neurons respond only to images of specific orientation and shape (triangles, circles, or squares). Considering that cats and other mammals can perceive complex and evolving shapes into a coherent whole, we can assume that perception is an aggregate of all these locally and hierarchically organized neurons. By that time, the first multilayer perceptrons were already fully developed so it didn't take too long before this idea of locality and specific sensitivity in neurons was modeled in perceptron architectures. From a computational neuroscience perspective, this idea was developed into maps of local receptive regions in the brain with the addition of selectively connected layers. This was adopted by the already ongoing field of neural networks and artificial intelligence. The first *reported* scientist who applied this notion of local specific computationally to a multilayer perceptron was Fukushima with his so-called neocognitron (1982).

Yann LeCun developed the idea of the neocognitron into his version called LeNet. It added the gradient descent backpropagation algorithm. This LeNet architecture is still the foundation for many more evolved CNN architectures introduced recently. A basic CNN like LeNet learns to detect edges from raw pixels in the first layer, then use these edges to detect simple shapes in the second layer, and later in the process uses these shapes to detect higher-level features, such as objects in an environment in higher layers. The layer further down the neural sequence is then a final classifier that uses these higher-level features. We can see the feedforward pass in a CNN like this: we move from a matrix input to pixels, we detect edges from pixels, then shapes from edges, and detect increasingly distinctive and more abstract and complex features from shapes.

Note

Each convolution or layer in the network is receptive for a specific feature (such as shape, angle, or color).

Deeper layers will combine these features into a more complex aggregate. This way, it can process complete images without burdening the network with the full input space of the image at step.

Up until now, we have only worked with fully connected neural networks where each layer is connected to each neighboring layer. These networks have proven to be quite effective, but have the downside of dramatically increasing the number of parameters that we have to train. On a side note, we might imagine that when we train a small image (28 x 28) in size, we can get away with a fully connected network. However, training a fully connected network on larger images that span across the entire image would be tremendously computationally expansive.

Note

To summarize, we can state that CNNs have the following benefits over fully connected neural networks:

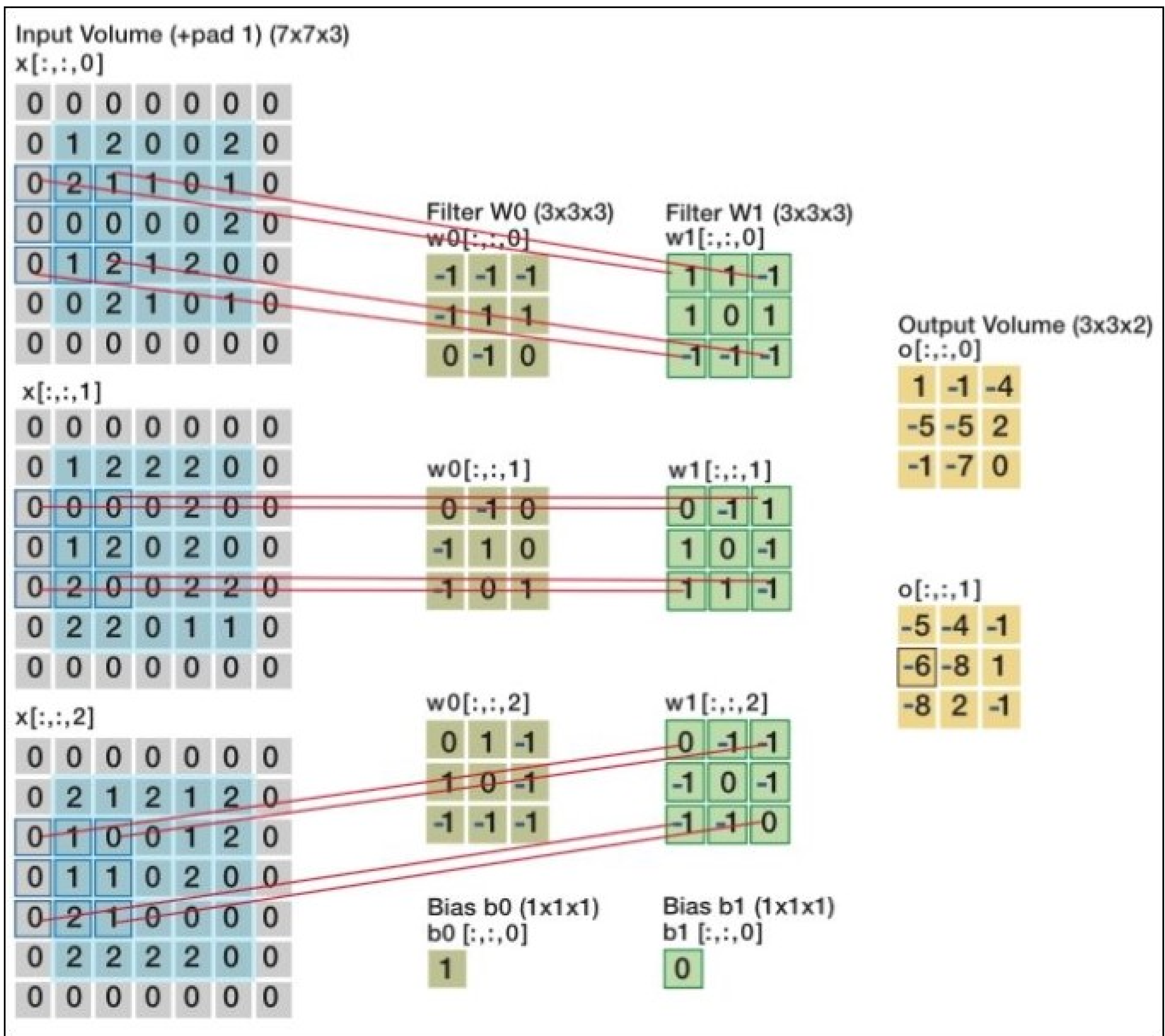
- They reduce the parameter space and thus prevent overtraining and computational load
- CNNs are invariant to object orientation (think of face recognition classifying faces with different locations)
- CNNs capable of learning and generalizing complex multidimensional features
- CNNs can be usefull in speech recogntion, image classification, and lately complex recommendation engines

CNNs utilize so-called receptive fields to connect the input to a feature map. The best way to understand CNNs is to dive deeper into the architecture, and of course get hands-on experience. So let's run through the types of layers that make up CNNs. The architecture of a CNN consists of three types of layers; namely, convolutional layer, pooling layer, and a fully-connected layer, where each layer accepts an input 3D volume (h, w, d) and transforms it into a 3D output through a differentiable function.

The convolution layer

We can understand the concept of convolution by imagining a spotlight of a certain size sliding over the input (pixel-values and RGB color dimensions), conveniently after which we compute a dot product between the filtered values (also referred to as patches) and the true input. This does two important things: first, it compresses the input, and more importantly, second, the network learns filters that only activate when they see some specific type of feature spatial position in the input.

Look at the following image to see how this works:



Two convolutional layers processing image input [7x7x3] input volume: an image of width 7, height 7, and with three color channels R,G,B

We can see from this image that we have two levels of filters (W0 and W1) and three dimensions (in the form of arrays) of inputs, all resulting in the dot product of the sliding spotlight/window over the input matrix. We refer to the size of this spotlight as *stride*, which means that the larger the stride, the smaller the output.

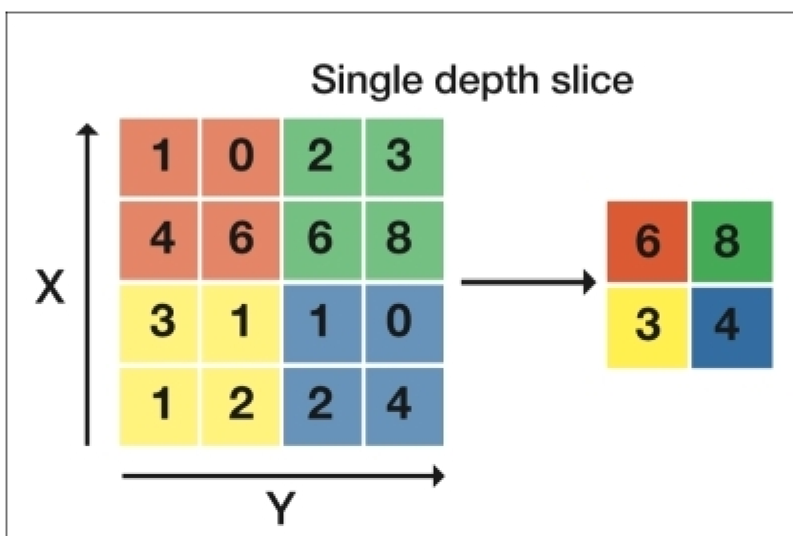
As you can see, when we apply a 3 x 3 filter, the full scope of the filter is processed at the center of the matrix, but once we move close to or past the edges, we start to lose out on the edges of the input. In this case, we apply what is called *zero-padding*. All elements that fall outside of the input dimensions are set

to zero in this case. Zero-padding became more or less a default setting for most CNN applications recently.

The pooling layer

The next type of layer that is often placed in between the filter layers is called a pooling layer or *subsampling* layer. What this does is it performs a downsampling operation along the spatial dimensions (width, height), which in turn helps with overfitting and reducing computational load. There are several ways to perform this downsampling, but recently Max Pooling turned out to be the most effective method.

Max-pooling is a simple method that compresses features by taking the maximum value of a patch of the neighboring feature. The following image will clarify this idea; each color box within the matrix represents a subsample of stride size 2:



A max-pooling layer with a stride of 2

The pooling layers are used mainly for the following reasons:

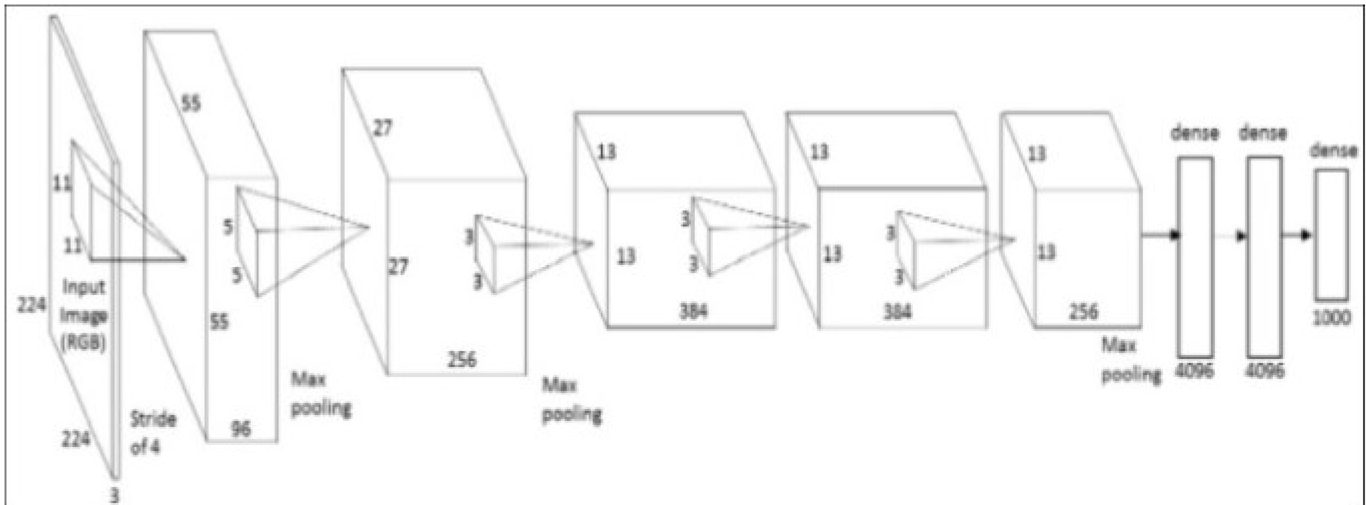
- Reduce the amount of parameters and thus computational load
- Regularization

Interestingly, the latest research findings suggest to leave out the pooling layer altogether, which will result in better accuracy (although at the expense of more strain on the CPU or GPU).

The fully connected layer

There is not much to explain about this type of layer. The final output where the classifications are computed (mostly with softmax) is a fully connected layer. However, in between convolutional layers, there (although rarely) are fully connected layers as well.

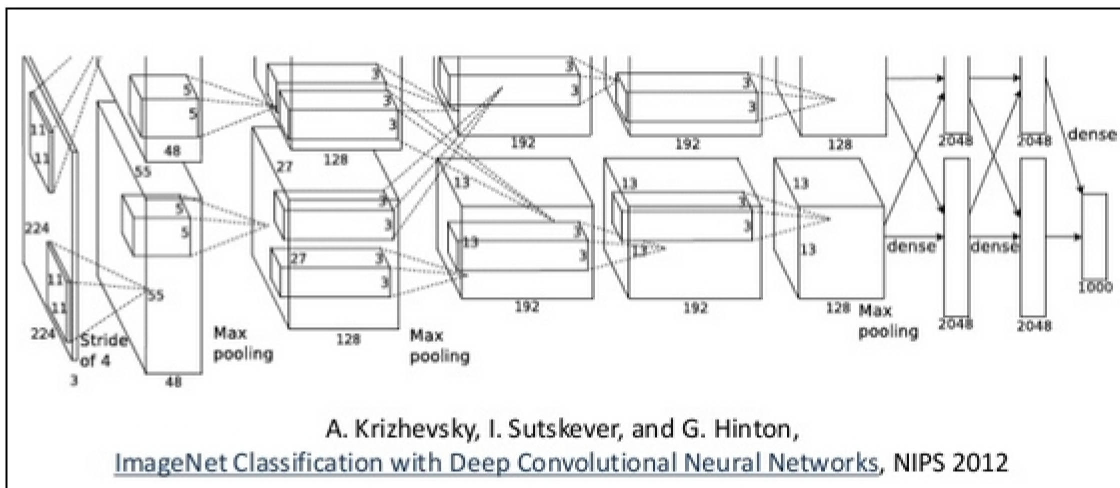
Before we apply a CNN ourselves, let's take what you have learned so far and inspect a CNN architecture to check our understanding. When we look at the ConvNet architecture in the following image, we can already get a sense of what a ConvNet will do to the input. The example is an effective convolutional neural network called AlexNet aimed at classifying 1.2 million images into 1,000 classes. It was used for the ImageNet contest in 2012. ImageNet is the most important image classification and localization competition in the world, which is held each year. AlexNet refers to Alex Krizhevsky (together with Vinod Nair and Geoffrey Hinton).



AlexNet architecture

When we look at the architecture, we can immediately see the input dimension 224 by 224 with three-dimensional depth. The stride size of four in the input, where max pooling layers are stacked, reduces the dimensionality of the input. In turn, this is followed by the convolutional layer. The two dense layers of size 4,096 are the fully connected layers leading to the final output that we mentioned before.

On a side note, we mentioned in a previous paragraph that TensorFlow's graph computation allows parallelization across GPUs. AlexNet did the same thing; look at the following image to see how they parallelized the architecture across GPUs:



A. Krizhevsky, I. Sutskever, and G. Hinton, ImageNet Classification with Deep Convolutional Neural Networks, NIPS 2012

The preceding image is taken from http://www.cs.toronto.edu/~fritz/absps/imagenet.pdf.

AlexNet let different models utilize GPUs by splitting up the architecture vertically to later be merged into the final classification output. That CNNs are more suitable for distributed processing is one of the biggest advantages of locally connected networks over fully connected ones. This model trained a set of 1.2 million images and took five days to complete on two NVIDIA GTX 580 3GB GPUs. Two multiple GPU units (a total of six GPUs) were used for this project.

CNN's with an incremental approach

Now that we have a decent understanding of the architectures of CNNs, let's get our hands dirty in Keras and apply a CNN.

For this example, we will use the famous CIFAR-10 face image dataset, which is conveniently available within the Keras domain. The dataset consists of 60,000, 32 x 32 color images with 10 target classes consisting of an airplane, automobile, bird, cat, deer, dog, frog, horse, ship, and truck. This is a smaller dataset than the one that was used for the AlexNet example. For more information, you can refer to https://www.cs.toronto.edu/~kriz/cifar.html.

In this CNN, we will use the following architecture to classify the image according to the 10 classes that we specified:

```
input->convolution 1 (32,3,3)->convolution 2(32,3,3)->pooling-
>dropout -> Output (Fully connected layer and softmax)
```

GPU Computing

If you have a CUDA compatible graphics card installed, you can utilize your GPU for this CNN example by placing the following piece of code on top of your IDE:

```
import os
os.environ['THEANO_FLAGS'] = 'device=gpu0, assert_no_cpu_op=raise,
on_unused_input=ignore, floatX=float32'
```

We do recommend however to first try this example on your regular CPU.

Let's first import and prepare the data.

We use a 32 x 32 input size considering this is the actual size of the image:

```
from keras.datasets import cifar10
from keras.preprocessing.image import ImageDataGenerator
from keras.models import Sequential
from keras.layers.core import Dense, Dropout, Activation, Flatten
from keras.layers.convolutional import Convolution2D, MaxPooling2D
from keras.optimizers import SGD
from keras.utils import np_utils

batch_size = 32
nb_classes = 10
nb_epoch = 5 #these are the number of epochs, watch out because it
might set your #cpu/gpu on fire.


# input image dimensions
img_rows, img_cols = 32, 32
# the CIFAR10 images are RGB
img_channels = 3

# the data, shuffled and split between train and test sets
(X_train, y_train), (X_test, y_test) = cifar10.load_data()
print('X_train shape:', X_train.shape)
print(X_train.shape[0], 'train samples')
print(X_test.shape[0], 'test samples')

#remember we need to encode the target variable
Y_train = np_utils.to_categorical(y_train, nb_classes)
Y_test = np_utils.to_categorical(y_test, nb_classes)
```

Now let's setup our CNN architecture and construct the model according to the architecture we have in mind.

For this example, we will train our CNN model with vanilla SGD and Nesterov momentum:

```
model = Sequential()

#this is the first convolutional layer, we set the filter size
model.add(Convolution2D(32, 3, 3, border_mode='same',
                            input_shape=(img_channels, img_rows,
img_cols)))
model.add(Activation('relu'))
#the second convolutional layer
model.add(Convolution2D(32, 3, 3))
model.add(Activation('relu'))
#here we specify the pooling layer
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))


#first we flatten the input towards the fully connected layer into
the softmax function
model.add(Flatten())
model.add(Dense(512))
model.add(Activation('relu'))
model.add(Dropout(0.2))
model.add(Dense(nb_classes))
model.add(Activation('softmax'))

# let's train the model using SGD + momentum like we have done
before.
sgd = SGD(lr=0.01, decay=1e-6, momentum=0.9, nesterov=True)
model.compile(loss='categorical_crossentropy', optimizer=sgd)

X_train = X_train.astype('float32')
X_test = X_test.astype('float32')

#Here we apply scaling to the features
X_train /= 255
X_test /= 255
```

This step is very important because here we specify the CNN to train incrementally. We saw in previous chapters (refer to Chapter 2, *Scalable Learning in Scikit-learn*), and in a previous paragraph, the computational efficiency of online and incremental learning. We can mimic some of its properties and apply it to CNNs by using a very small epoch size with a smaller `batch_size` (fraction of the training set in each epoch), and train them incrementally in a for loop. This way, we can given the same amount of epochs and train our CNN in a much shorter time with also a lower burden on main memory. We can implement this very powerful idea with a simple for loop as follows:

```
for epoch in xrange(nb_epoch):
    model.fit(X_train, Y_train, batch_size=batch_size,
nb_epoch=1,show_accuracy=True
              ,validation_data=(X_test, Y_test), shuffle=True)

OUTPUT:]

X_train shape: (50000, 3, 32, 32)
50000 train samples
10000 test samples
Train on 50000 samples, validate on 10000 samples
Epoch 1/1
50000/50000 [==============================] - 1480s - loss: 1.4464
- acc: 0.4803 - val_loss: 1.1774 - val_acc: 0.5785
Train on 50000 samples, validate on 10000 samples
Epoch 1/1
50000/50000 [==============================] - 1475s - loss: 1.0701
- acc: 0.6212 - val_loss: 0.9959 - val_acc: 0.6525
Train on 50000 samples, validate on 10000 samples
Epoch 1/1
50000/50000 [==============================] - 1502s - loss: 0.8841
- acc: 0.6883 - val_loss: 0.9395 - val_acc: 0.6750
Train on 50000 samples, validate on 10000 samples
Epoch 1/1
50000/50000 [==============================] - 1555s - loss: 0.7308
- acc: 0.7447 - val_loss: 0.9138 - val_acc: 0.6920
Train on 50000 samples, validate on 10000 samples
Epoch 1/1
50000/50000 [==============================] - 1587s - loss: 0.5972
- acc: 0.7925 - val_loss: 0.9351 - val_acc: 0.6820
```

We can see our CNN train to finally arrive at a validation accuracy approaching 0.7. Considering we have trained a complex model on a high-dimensional dataset with 50,000 training examples and 10 target classes, this is already satisfying. The maximum possible score that can be achieved with a CNN on this dataset requires at least 200 epochs. The method proposed in this example is by no means final. This is quite a basic implementation to get you started with CNNs. Feel free to experiment by adding or removing layers, adjusting the batch size, and so on. Play with the parameters to get a feel of how this works.

If you want to learn more about the latest developments in convolutional layers, take a look at **residual network** (**ResNet**), which is one of the latest improvements on CNN architectures.

Kaiming He and others were the winners of ImageNet 2015 (ILSVRC). It features an interesting architecture that uses a method called batch normalization, a method that normalizes the feature transformation between layers. There is a batch normalization function in Keras that you might want to experiment with (http://keras.io/layers/normalization/).

To give you an overview of the latest generation of ConvNets, you might want to familiarize yourself with the following parameter settings for CNNs that have been found to be more effective:

- Small stride
- Weight decay (regularization instead of dropout)
- No dropout
- Batch normalization between mid-level layers
- Less to no pre-training (Autoencoders and Boltzman machines slowly fall out of fashion for image classification)

Another interesting notion is that recently convolutional networks are used for applications besides image detection. They are used for language and text classification, sentence completion, and even recommendation systems. An interesting example is Spotify's music recommendation engine, which is based on Convolutional Neural Networks. You can take a look here for further information:

- http://benanne.github.io/2014/08/05/spotify-cnns.html
- http://machinelearning.wustl.edu/mlpapers/paper_files/NIPS2013_5004.pdf

Currently, convolutional networks are used for the following actions:

- Face detection (Facebook)
- Film classification (YouTube)
- Speech and text
- Generative art (Google DeepDream, for instance)
- Recommendation engines (music recommendation—Spotify)

Summary

In this chapter, we have come quite a long way covering the TensorFlow landscape and its corresponding methods. We got acquainted with how to set up basic regressors, classifiers, and single-hidden layer neural networks. Even though the programming TensorFlow operations are relatively straightforward, for off-the-shelf machine learning tasks, TensorFlow might be a little bit too tedious. This is exactly where SkFlow comes in, a higher-level library with an interface quite similar to Scikit-learn. For incremental or even out-of-core solutions, SkFlow provides a partial fit method, which can easily be set up. Other large scale solutions are either restricted to GPU applications or are at a premature stage. So for now, we have to settle for incremental learning strategies when it comes to scalable solutions.

We also provided an introduction to Convolutional Neural Networks and saw how they can be set up in Keras.

Chapter 6. Classification and Regression Trees at Scale

In this chapter, we will focus on scalable methods for classification and regression trees. The following topics will be covered:

- Tips and tricks for fast random forest applications in Scikit-learn
- Additive random forest models and subsampling
- GBM gradient boosting
- XGBoost together with streaming methods
- Very fast GBM and random forest in H2O

The aim of a decision tree is to learn a series of decision rules to infer the target labels based on the training data. Using a recursive algorithm, the process starts at the tree root and splits the data on the feature that results in the lowest impurity. Currently, the most widely applicable scalable tree-based applications are based on CART. Introduced by Breiman, Friedman Stone, and Ohlson in 1984, **CART** is an abbreviation of **Classification and Regression Trees**. CART is different from other decision tree models (such as ID3, C4.5/C5.0, CHAID, and MARS) in two ways. First, CART is applicable to both classification and regression problems. Second, it builds binary trees (at each split, resulting in a binary split). This enables CART trees to operate recursively on given features and optimize in a greedy fashion on an error metric in the form of impurity. These binary trees together with scalable solutions are the focus of this chapter.

Let's look closely at how these trees are constructed. We can see a decision tree as a graph with nodes, passing information down from top to bottom. Each decision within the tree is made by binary splits for either classes (Boolean) or continuous variables (a threshold value) resulting in a final prediction.

Trees are constructed and learned by the following procedure:

- Recursively finding the variable that best splits the target label from root to terminal node. This is measured by the impurity of each feature that we minimize based on the target outcome. In this chapter, the relevant impurity measures are Gini impurity and cross entropy.

Gini impurity

$$Gini(S) = 1 - \sum_{i=1}^{k} p_i^2$$

Gini impurity is a metric that measures the divergence between the probability *pi* of the target classes (*k*) so that an equal spread of probability values over the target classes result in a high Gini impurity.

Cross entropy

$$D = -\sum_{k=1}^{k} \hat{p}_{mk} \, log \, \hat{p}_{mk}.$$

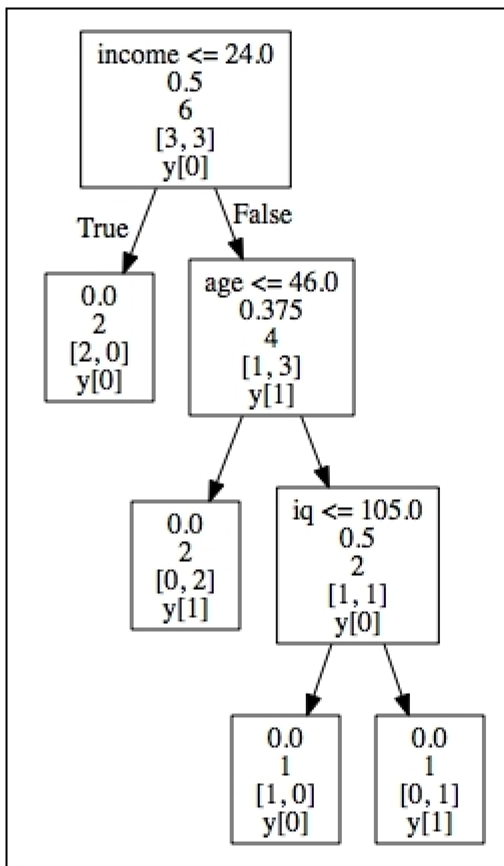
With cross entropy, we look at the log probability of misclassification. Both metrics are proven to yield quite similar results. However, Gini impurity is computationally more efficient because it does not require calculating the logs.

We do this until a stopping criterion is met. This criteria can roughly mean two things: one, adding new variables no longer improves the target outcome, and second, a maximum tree depth or tree complexity threshold is reached. Note that very deep and complex trees with many nodes can easily lead to overfitting. In order to prevent this, we generally prune the tree by limiting the tree depth.

To get an intuition for how this process works, let's build a decision tree with Scikit-learn and visualize it with graphviz. First, create a toy dataset to see if we can predict who is a smoker and who is not based on IQ (numeric), age (numeric), annual income (numeric), business owner (Boolean), and university degree (Boolean). You need to download the software from http://www.graphviz.org in order to load a visualization of the `tree.dot` file that we are going to create with Scikit-learn:

```
import numpy as np
from sklearn import tree
iq=[90,110,100,140,110,100]
age=[42,20,50,40,70,50]
anincome=[40,20,46,28,100,20]
businessowner=[0,1,0,1,0,0]
univdegree=[0,1,0,1,0,0]
smoking=[1,0,0,1,1,0]
ids=np.column_stack((iq, age, anincome,businessowner,univdegree))
names=['iq','age','income','univdegree']
dt = tree.DecisionTreeClassifier(random_state=99)
dt.fit(ids,smoking)
dt.predict(ids)
tree.export_graphviz(dt,out_file='tree2.dot',feature_names=names,labe
l=all,max_depth=5,class_names=True)
```

You can now find the `tree.dot` file in your working directory. Once you find this file, you can open it with the graphviz software:



- Root node (Income): This is the starting node that represents the feature with the highest information gain and lowest impurity (Gini=.5)
- Internal nodes (Age and IQ):This is each node between the root node and terminal. Parent nodes pass down decision rules to the receiving end—child nodes (left and right)
- Terminal nodes (leaf nodes): The target labels partitioned by the tree structure

Tree-depth is the number of edges from the root node to the terminal nodes. In this case, we have a tree depth of 3.

We can now see all the binary splits resulting from the generated tree. At the top in the root node, we can see that a person with an income lower than 24k is not a smoker (income < 24). We can also see the corresponding Gini impurity (.5) for that split at each node. There are no left child nodes because the decision is final. The path simply ends there because it fully divides the target class. However, in the right child node (age) of income, the tree branches out. Here, if age is less than or equal to 46, then that person is not a smoker, but with an age older than 46 and an IQ lower than 105, that person is a smoker. Just as important are the few features we created that are not part of the tree—degree and business owner. This is because the variables in the tree are able to classify the target labels without them. These omitted features simply don't contribute to decreasing the impurity level of the tree.

Single trees have their drawbacks because they easily overtrain and therefore don't generalize well to unseen data. The current generation of these techniques are trained with ensemble methods where single

trees are aggregated into much more powerful models. These CART ensemble techniques are one of the most used methods for machine learning because of their accuracy, ease of use, and capability to handle heterogeneous data. These techniques have been successfully applied in recent datascience competitions such as Kaggle and KDD-cup. As ensemble methods for classification and regression trees are currently the norm in the world of AI and data science, scalable solutions for CART-ensemble methods will be the main topic for this chapter.

Generally, we can discern two classes of ensemble methods that we use with CART models, namely bagging and boosting. Let's work through these concepts to form a better understanding of how the process of ensemble formation works.

Bootstrap aggregation

Bagging is an abbreviation of **bootstrap aggregation**. The bootstrapping technique originated in a context where analysts had to deal with a scarcity of data. With this statistical approach, subsamples were used to estimate population parameters when a statistical distribution couldn't be figured out a priori. The goal of bootstrapping is to provide a more robust estimate for population parameters where more variability is introduced to a smaller dataset by random subsampling with replacement. Generally, bootstrapping follows the following basic steps:

1. Randomly sample a batch of size *x* with replacement from a given dataset.
2. Calculate a metric or parameter from each sample to estimate the population parameters.
3. Aggregate the results.

In recent years, bootstrap methods have been used for parameters of machine learning models as well. An ensemble is most effective when its classifiers provide highly diverse decision boundaries. This diversity in ensembles can be achieved in the diversity of its underlying models and data these models are trained on. Trees are very well-suited for such diversity among classifiers because the structure of trees can be highly variable. However, the most popular method to ensemble is to use different training datasets to train individual classifiers. Often, such datasets are obtained through subsampling-sampling techniques, such as bootstrapping and bagging. Everything started from the idea that by leveraging more data, we can reduce the variance of the estimates. If it is not possible to have more data at hand, resampling can provide a significant improvement because it allows retraining the algorithm over many versions of the training sample. This is where the idea of bagging comes in; using bagging, we take the original bootstrap idea a bit further by aggregation (averaging, for instance) of the results of many resamples in order to arrive at a final prediction where the errors due to in-sample overfitting are reciprocally smoothed out.

When we apply an ensemble technique such as bagging to tree models, we build multiple trees on each separate bootstrapped sample (or subsampled using sampling without replacement) of the original dataset and then aggregate the results (usually by arithmetic, geometric averaging, or voting).

In such a fashion, a standard bagging algorithm will look as follows:

1. Draw an *n* amount of random samples with size *K* with replacement from the full dataset (*S1*, *S2*, … *Sn*).
2. Train distinct trees on (*S1*, *S2*, … *Sn*).

3. Calculate predictions on samples (*S1*, *S2*, … *Sn*) on new data and aggregate their results.

CART models benefit very well from bagging methods because of the stochasticity and diversity it introduces.

Random forest and extremely randomized forest

Apart from bagging, based on training examples, we can also draw random subsamples based on features. Such a method is referred to as Random Subspaces. Random subspaces are particularly useful for high-dimensional data (data with lots of features) and it is the foundation of the method that we refer to as random forest. At the time of writing this, random forest is the most popular machine learning algorithm because of its ease of use, robustness to messy data, and parallelizability. It found its way into all sorts of applications such as location apps, games, and screening methods for healthcare applications. For instance, the Xbox Kinect uses a random forest model for motion detection purposes. Considering that the random forest algorithm is based on bagging methods, the algorithm is relatively straightforward:

1. Bootstrap *m* samples of size *N* from the available sample.
2. Trees are constructed independently on each subset (*S1*, *S2*, … *Sn*) using a different fraction of the feature set *G* (without replacement) at every node split.
3. Minimize the error of the node splits (based on the Gini index or entropy measure).
4. Have each tree make a prediction and aggregate the results, using voting for classification and averaging for regression.

As bagging relies on multiple subsamples, it's an excellent candidate for parallelization where each CPU unit is dedicated to calculating separate models. In this way, we can speed up the learning using the widespread availability of multiple cores. As a limit in such a scaling strategy, we have to be aware that Python is single-threaded and we will have to replicate many Python instances, each replicating the memory space with the in-sample examples we have to employ. Consequently, we will need to have a lot of RAM memory available in order to fit the training matrix and the number of processes. If the available RAM does not suffice, setting the number of parallel tree computations running at once on our computer will not help scaling the algorithm. In this case, CPU usage and RAM memory are important bottlenecks.

Random forests models are quite easy to employ for machine learning because they don't require a lot of hyperparameter tuning to perform well. The most important parameters of interest are the *amount of trees* and depth of the trees (tree depth) that have the most influence on the performance of the model. When operating on these two hyperparameters, it results in an accuracy/performance trade-off where more trees and more depth lead to higher computational load. Our experience as practitioners suggests not to set the value of the *amount of trees* too high because eventually, the model will reach a plateau in performance and won't improve anymore when adding more trees but will just result in taxing the CPU cores. Under such considerations, although a random forest model with default parameters performs well just out of the box, we can still increase its performance by tuning the number of trees. See the following table for an overview of the hyperparameters for random forests.

The most important parameters for bagging with a random forest:

- `n_estimators`: The number of trees in the model
- `max_features`: The number of features used for tree construction
- `min_sample_leaf`: Node split is removed if a terminal node contains less samples than the minimum

- `max_depth`: The number of nodes that we pass top-down from the root to the terminal node
- `criterion`: The method used to calculate the best split (Gini or entropy)
- `min_samples_split`: The minimum number of samples required to split an internal node

Scikit-learn provides a wide range of powerful CART ensemble applications, some of which are quite computationally efficient. When it comes to random forests, there is an often-overlooked algorithm called extra-trees, better known as **Extremely Randomized Forest**. When it comes to CPU efficiency, extra-trees can deliver a considerable speedup from regular random forests—sometimes even in tenfolds.

In the following table, you can see the computation speed for each method. Extra-trees is considerably faster with a more pronounced difference once we increase the sample size:

samplesize	extratrees	randomforest
100000	25.9 s	164 s
50000	9.95 s	35.1 s
10000	2.11 s	6.3 s

Models were trained with 50 features and 100 estimators for both extra trees and random forest. We used a quad-core MacBook Pro with 16GB RAM for this example. We measured the training time in seconds.

In this paragraph, we will use extreme forests instead of the vanilla random forest method in Scikit-learn. So you might ask: how are they different? The difference is not strikingly complex. In random forests, the node split decision rules are based on a best score resulting from the randomly selected features at each iteration. In extremely randomized forests, a random split is generated on each feature in the random subset (so there are no computations spent on looking for the best split for each feature) and then the best scoring threshold is selected. Such an approach brings about some advantageous properties because the method leads to achieving models with lower variance even if each individual tree is grown until having the greatest possible accuracy in terminal nodes. As more randomness is added to branch splits, the tree learner makes errors, which are consequently less correlated among the trees in the ensemble. This will lead to much more uncorrelated estimations in the ensemble and, depending on the learning problem (there is no free lunch, after all), to a lower generalization error than a standard random forest ensemble. In practice, however, a regular random forest model can provide a slightly higher accuracy.

Given such interesting learning properties, with a more efficient node split calculation and the same parallelism that can be leveraged for random forests, we consider extremely randomized trees as an excellent candidate in the range of ensemble tree algorithms, if we want to speed up in-core learning.

To read a detailed description of the extremely randomized forest algorithm, you can read the following article that started everything:

P. Geurts, D. Ernst, and L. Wehenkel, *Extremely randomized trees*, Machine Learning, 63(1), 3-42, 2006. This article can be freely accessed at https://www.semanticscholar.org/paper/Extremely-randomized-trees-Geurts-Ernst/336a165c17c9c56160d332b9f4a2b403fccbdbfb/pdf.

As an example of how to scale in-core tree ensembles, we will run an example where we apply an efficient random forest method to credit data. This dataset is used to predict credit card clients' default rates. The data consists of 18 features and 30,000 training examples. As we need to import a file in XLS format, you will need to install the `xlrd` package, and we can achieve this by typing the following in the command-line terminal:

```
$ pip install xlrd
import pandas as pd
import numpy as np
import os
import xlrd
import urllib
#set your path here
os.chdir('/your-path-here')

url = 'http://archive.ics.uci.edu/ml/machine-learning-databases/
00350/default%20of%20credit%20card%20clients.xls'
filename='creditdefault.xls'
urllib.urlretrieve(url, filename)

target = 'default payment next month'
data = pd.read_excel('creditdefault.xls', skiprows=1)

target = 'default payment next month'
y = np.asarray(data[target])
features = data.columns.drop(['ID', target])
X = np.asarray(data[features])

from sklearn.ensemble import ExtraTreesClassifier
from sklearn.cross_validation import cross_val_score
from sklearn.datasets import make_classification
from sklearn.cross_validation import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X,
y,test_size=0.30, random_state=101)

clf = ExtraTreesClassifier(n_estimators=500, random_state=101)
clf.fit(X_train,y_train)
scores = cross_val_score(clf, X_train, y_train,
cv=3,scoring='accuracy', n_jobs=-1)
print "ExtraTreesClassifier -> cross validation accuracy: mean =
%0.3f std = %0.3f" % (np.mean(scores), np.std(scores))
```

Output]

ExtraTreesClassifier -> cross validation accuracy: mean = 0.812 std = 0.003

Now that we have some base estimation of the accuracy on the training set, let's see how well it performs on the test set. In this case, we want to monitor the false positives and false negatives and also check for class imbalances on the target variable:

```
y_pred=clf.predict(X_test)
from sklearn.metrics import confusion_matrix
confusionMatrix = confusion_matrix(y_test, y_pred)
print confusionMatrix
from sklearn.metrics import accuracy_score
accuracy_score(y_test, y_pred)
```

```
OUTPUT:
[[6610  448]
[1238  704]]
```

```
Our overall test accuracy:
0.8126666666666665
```

Interestingly, the test set accuracy is equal to our training results. As our baseline model is just using the default setting, we can try to improve performance by tuning the hyperparameters, a task that can be computationally expensive. Recently, more computationally efficient methods have been developed for hyperparameter optimization, which we will cover in the next section.

Fast parameter optimization with randomized search

You might already be familiar with Scikit-learn's gridsearch functionalities. It is a great tool but when it comes to large files, it can ramp up training time enormously depending on the parameter space. For extreme random forests, we can speed up the computation time for parameter tuning using an alternative parameter search method named **randomized search**. Where common gridsearch taxes both CPU and memory by systematically testing all possible combinations of the hyperparameter settings, randomized search selects combinations of hyperparameters at random. This method can lead to a considerable computational speedup when the gridsearch is testing more than 30 combinations (for smaller search spaces, gridsearch is still competitive). The gain achievable is in the same order as we have seen when we switched from random forests to extremely randomized forests (think between a two to tenfold gain, depending on hardware specifications, hyperparameter space, and the size of the dataset).

We can specify the number of hyperparameter settings that are evaluated randomly by the `n_iter` parameter:

```
from sklearn.grid_search import GridSearchCV, RandomizedSearchCV


param_dist = {"max_depth": [1,3, 7,8,12,None],
    "max_features": [8,9,10,11,16,22],
    "min_samples_split": [8,10,11,14,16,19],
    "min_samples_leaf": [1,2,3,4,5,6,7],
    "bootstrap": [True, False]}

#here we specify the search settings, we use only 25 random
parameter
#valuations but we manage to keep training times in check.
rsearch = RandomizedSearchCV(clf, param_distributions=param_dist,
    n_iter=25)

rsearch.fit(X_train,y_train)
rsearch.grid_scores_

bestclf=rsearch.best_estimator_
print bestclf
```

Here, we can see the list of optimal parameter settings for our model.

We can now use this model to make predictions on our test set:

```
OUTPUT:
ExtraTreesClassifier(bootstrap=False, class_weight=None,
criterion='gini',
```

```
    max_depth=12, max_features=11, max_leaf_nodes=None,
    min_samples_leaf=4, min_samples_split=10,
    min_weight_fraction_leaf=0.0, n_estimators=500, n_jobs=1,
    oob_score=False, random_state=101, verbose=0, warm_start=False)

y_pred=bestclf.predict(X_test)
confusionMatrix = confusion_matrix(y_test, y_pred)
print confusionMatrix
accuracy=accuracy_score(y_test, y_pred)
print accuracy

OUT
[[6733  325]
[1244  698]]

Out[152]:
0.82566666666666666
```

We managed to increase the performance of our model within a manageable range of training time and increase accuracy at the same time.

Extremely randomized trees and large datasets

So far, we have looked at solutions to scale up leveraging multicore CPUs and randomization thanks to the specific characteristics of random forest and its more efficient alternative, extremely randomized forest. However, if you have to deal with a large dataset that won't fit in memory or is too CPU-demanding, you may want to try an out-of-core solution. The best solution for an out-of-core approach with ensembles is the one provided by H2O, which will be covered in detail later in this chapter. However, we can exert another practical trick in order to have random forest or extra-trees running smoothly on a large-scale dataset. A second-best solution would be to train models on subsamples of the data and then ensemble the results from each model built on different subsamples of the data (after all, we just have to average or group the results). In Chapter 3, *Fast-Learning SVMs*, we already introduced the concept of reservoir sampling, dealing with sampling on data streams. In this chapter, we will use sampling again, resorting to a larger choice of sampling algorithms. First, let's install a really handy tool called subsample developed by Paul Butler (https://github.com/paulgb/subsample), a command-line tool to sample data from a large, newline-separated dataset (typically, a CSV-like file). This tool provides quick and easy sampling methods such as reservoir sampling.

As seen in Chapter 3, *Fast-Learning SVMs*, reservoir sampling is a sampling algorithm that helps sampling fixed-size samples from a stream. Conceptually simple (we have seen the formulation in Chapter 3), it just needs a simple pass over the data to produce a sample that will be stored in a new file on disk. (Our script in Chapter 3 stored it in memory, instead.)

In the next example, we will use this subsample tool together with a method to ensemble the models trained on these subsamples.

To recap, in this section, we are going to perform the following actions:

1. Create our dataset and split it into test and training data.
2. Draw subsamples of the training data and save them as separate files on our harddrive.
3. Load these subsamples and train extremely randomized forest models on them.
4. Aggregate the models.
5. Check the results.

Let's install this subsample tool with `pip`:

```
$pip install subsample
```

In the command line, set the working directory containing the file that you want to sample:

```
$ cd /yourpath-here
```

At this point, using the `cd` command, you can specify your working directory where you will need to store the file that we will create in the next step.

We do this in the following way:

```
from sklearn.datasets import fetch_covtype
import numpy as np
from sklearn.cross_validation import train_test_split
dataset = fetch_covtype(random_state=111, shuffle=True)
dataset = fetch_covtype()
X, y = dataset.data, dataset.target
X_train, X_test, y_train, y_test = train_test_split(X,y,
test_size=0.3, random_state=0)
del(X,y)
covtrain=np.c_[X_train,y_train]
covtest=np.c_[X_test,y_test]
np.savetxt('covtrain.csv', covtrain, delimiter=",")
np.savetxt('covtest.csv', covtest, delimiter=",")
```

Now that we have split the dataset into test and training sets, let's subsample the training set in order to obtain chunks of data that we can manage to upload in-memory. Considering that the size of the full training dataset is 30,000 examples, we will subsample three smaller datasets, each one made of 10,000 items. If you have a computer equipped with less than 2GB of RAM memory, you may find it more manageable to split the initial training set into smaller files, though the modeling results that you will obtain will likely differ from our example based on three subsamples. As a rule, the fewer the examples in the subsamples, the greater will be the bias of your model. When subsampling, we are actually trading the advantage of working on a more manageable amount of data against an increased bias of the estimates:

```
$ subsample --reservoir -n 10000 covtrain.csv > cov1.csv

$ subsample --reservoir -n 10000 covtrain.csv > cov2.csv
```

```
$ subsample --reservoir -n 10000 covtrain.csv>cov3.csv
```

You can now find these subsets in the folder that you specified in the command line.

Now make sure that you set the same path in your IDE or notebook.

Let's load the samples one by one and train a random forest model on them.

To combine them later for a final prediction, note that we keep a single line-by-line approach so that you can follow the consecutive steps closely.

In order for these examples to be successful, make sure that you are working on the same path set in your IDE or Jupyter Notebook:

```
import os
os.chdir('/your-path-here')
```

At this point, we are ready to start learning from the data and we can load the samples in-memory one by one and train an ensemble of trees on them:

```
import numpy as np
from sklearn.ensemble import ExtraTreesClassifier
from sklearn.cross_validation import cross_val_score
from sklearn.cross_validation import train_test_split
import pandas as pd
import os
```

After reporting a validation score, the code will proceed training our model on all the data chunks, one at a time. As we are learning separately from different data partitions, data chunk after data chunk, we initialize the ensemble learner (in this case, `ExtraTreeClassifier`) using the `warm_start=Trueparameter` and `set_params` method that incrementally adds trees from the previous training sessions as the fit method is called multiple times:

```
#here we load sample 1
df = pd.read_csv('/yourpath/cov1.csv')
y=df[df.columns[54]]
X=df[df.columns[0:54]]




clf1=ExtraTreesClassifier(n_estimators=100,
random_state=101,warm_start=True)
clf1.fit(X,y)
scores = cross_val_score(clf1, X, y, cv=3,scoring='accuracy',
n_jobs=-1)
print "ExtraTreesClassifier -> cross validation accuracy: mean =
```

```
%0.3f std = %0.3f" % (np.mean(scores), np.std(scores))
print scores
print 'amount of trees in the model: %s' % len(clf1.estimators_)

#sample 2
df = pd.read_csv('/yourpath/cov2.csv')
y=df[df.columns[54]]
X=df[df.columns[0:54]]

clf1.set_params(n_estimators=150, random_state=101,warm_start=True)
clf1.fit(X,y)
scores = cross_val_score(clf1, X, y, cv=3,scoring='accuracy',
n_jobs=-1)
print "ExtraTreesClassifier after params -> cross validation
accuracy: mean = %0.3f std = %0.3f" % (np.mean(scores),
np.std(scores))
print scores
print 'amount of trees in the model: %s' % len(clf1.estimators_)

#sample 3
df = pd.read_csv('/yourpath/cov3.csv')
y=df[df.columns[54]]
X=df[df.columns[0:54]]
clf1.set_params(n_estimators=200, random_state=101,warm_start=True)
clf1.fit(X,y)
scores = cross_val_score(clf1, X, y, cv=3,scoring='accuracy',
n_jobs=-1)
print "ExtraTreesClassifier after params -> cross validation
accuracy: mean = %0.3f std = %0.3f" % (np.mean(scores),
np.std(scores))
print scores
print 'amount of trees in the model: %s' % len(clf1.estimators_)

# Now let's predict our combined model on the test set and check our
score.

df = pd.read_csv('/yourpath/covtest.csv')
X=df[df.columns[0:54]]
y=df[df.columns[54]]
pred2=clf1.predict(X)
scores = cross_val_score(clf1, X, y, cv=3,scoring='accuracy',
n_jobs=-1)
print "final test score %r" % np.mean(scores)

OUTPUT:]
ExtraTreesClassifier -> cross validation accuracy: mean = 0.803 std
= 0.003
```

```
[ 0.805997    0.79964007  0.8021021 ]
amount of trees in the model: 100
ExtraTreesClassifier after params -> cross validation accuracy: mean
= 0.798 std = 0.003
[ 0.80155875  0.79651861  0.79465626]
amount of trees in the model: 150
ExtraTreesClassifier after params -> cross validation accuracy: mean
= 0.798 std = 0.006
[ 0.8005997   0.78974205  0.8033033 ]
amount of trees in the model: 200
final test score 0.92185447181058278
```

Note

Warning: This method looks un-Pythonic but is quite effective.

We have improved the score on the final prediction now; we went from an accuracy of around .8 to an accuracy of .922 on the test set. This is because we have a final combined model containing all the tree information combined of the previous three random forest models. In the code's output, you can also notice the number of trees that are added to the initial model.

From here, you might want to try such an approach on even larger datasets leveraging more subsamples, or apply randomized search to one of the subsamples for better tuning.

CART and boosting

We started this chapter with bagging; now we will complete our overview with boosting, a different ensemble method. Just like bagging, boosting can be used for both regression and classification and has recently overshadowed random forest for higher accuracy.

As an optimization process, boosting is based on the stochastic gradient descent principle that we have seen in other methods, namely optimizing models by minimizing error according to gradients. The most familiar boosting methods to date are **AdaBoost** and Gradient Boosting (GBM and recently XGBoost). The AdaBoost algorithm comes down to minimizing the error of those cases where the prediction is slightly wrong so that cases that are harder to classify get more attention. Recently, AdaBoost fell out of favor as other boosting methods were found to be generally more accurate.

In this chapter, we will cover the two most effective boosting algorithms available to date to Python users: **Gradient Boosting Machine** (**GBM**) found in the Scikit-learn package and **extreme gradient boosting** (**XGBoost**). As GBM is sequential in nature, the algorithm is hard to parallelize and thus harder to scale than random forest, but some tricks will do the job. Some tips and tricks to speed up the algorithm will be covered with a nice out-of-memory solution for H2O.

Gradient Boosting Machines

As we have seen in the previous sections, random forests and extreme trees are efficient algorithms and both work quite well with minimum effort. Though recognized as a more accurate method, GBM is not very easy to use and it is always necessary to tune its many hyperparameters in order to achieve the best results. Random forest, on the other hand, can perform quite well with only a few parameters to consider (mostly tree depth and number of trees). Another thing to pay attention to is overtraining. Random forests are less sensitive to overtraining than GBM. So, with GBM, we also need to think about regularization strategies. Above all, random forests are much easier to perform parallel operations where as GBM is sequential and thus slower to compute.

In this chapter, we will apply GBM in Scikit-learn, look at the next generation of tree boosting algorithms named XGBoost, and implement boosting on a larger scale on H2O.

The GBM algorithm that we use in Scikit-learn and H2O is based on two important concepts: **additive expansion** and gradient optimization by the **steepest descent** algorithm. The general idea of the former is to generate a sequence of relatively simple trees (weak learners), where each successive tree is added along a gradient. Let's assume that we have M trees that aggregate the final predictions in the ensemble. The tree in each iteration fk is now part of a much broader space of all the possible trees in the model (ϕ) (in Scikit-learn, this parameter is better known as `n_estimators`):

$$\hat{y} = \sum_{m=1}^{M} f_k(x_i), f_k \in \phi$$

The additive expansion will add new trees to previous trees in a stage-wise manner:

$\hat{y}_i^{(0)}=0$

$\hat{y}_i^{(1)}= f_1(x_i) = \hat{y}_i^{(0)} + f_1(x_i)$ *(this is our first tree)*

$\hat{y}_i^{(2)}=f_1(x_i)+f_2(x_i)=\hat{y}_i^{(1)} + f_2(x_i)$ *(our second tree added to the previous)*

And so on... till stopping criteria is reached

The prediction of our gradient boosting ensemble is simply the sum of the predictions of all the previous trees and the newly added tree $(\hat{y}_i^{(t-1)}) + f_1(x_i)$, more formally leading to the following:

$$\hat{y}_i^{(t)} = \sum_{m=1}^{M} f_k\left(x_i\right) = \hat{y}_i^{(t-1)} + f_i\left(x_i\right)$$

The second important and yet quite tricky part of the GBM algorithm is gradient optimization by **steepest descent**. This means that we add increasingly more powerful trees to the additive model. This is achieved by applying gradient optimization to the new trees. How do we perform gradient updates with trees as there are no parameters like we have seen with traditional learning algorithms? First, we parameterize the trees; we do this by recursively upgrading the node split values along a gradient where the nodes are represented by a vector. This way, the steepest descent direction is the negative gradient of the loss function and the node splits will be upgraded and learned, leading to:

- λ : A shrinkage parameter (also referred to as learning rate in this context) that will cause the ensemble to learn slowly with the addition of more trees
- γ_{mi} : The gradient upgrade parameter also referred to as the step length

The prediction score for each leaf is thus the final score for the new tree that then simply is the sum over each leaf:

$$\hat{y}_i^{(t)} = \sum_{m=1}^{M} f_k\left(x_i\right) = \hat{y}_i^{(t-1)} + \lambda\gamma_{mi} f_i\left(x_i\right)$$

So to summarize, GBM works by incrementally adding more accurate trees learned along the gradient.

Now that we understand the core concepts, let's run a GBM example and look at the most important parameters. For GBM, these parameters are extra important because when we set this number of trees too high, we are bound to strain our computational resources exponentially. So be careful with these parameters. Most parameters in Scikit-learn's GBM application are the same as in random forest, which we have covered in the previous paragraph. There are three parameters that we need to take into account that need special attention.

max_depth

Contrary to random forests, which perform better when they build their tree structures to their maximum extension (thus building and ensembling predictors with high variance), GBM tends to work better with smaller trees (thus leveraging predictors with a higher bias, that is, weak learners). Working with smaller decision trees or just with stumps (decision trees with only one single branch) can alleviate the training time, trading off the speediness of execution against a larger bias (because a smaller tree can hardly intercept more complex relationships in data).

learning_rate

Also known as shrinkage λ, this is a parameter related to the gradient descent optimization and how each tree will contribute to the ensemble. Smaller values of this parameter can improve the optimization in the training process, though it will require more estimators to converge and thus more computational time. As it affects the weight of each tree in the ensemble, smaller values imply that each tree will contribute a small part to the optimization process and you will need more trees before reaching a good solution. Consequently, when optimizing this parameter for performance, we should avoid too large values that may lead to sub-optimal models; we also have to avoid using too low values because this will affect the computation time heavily (more trees will be needed for the ensemble to converge to a solution). In our experience, a good starting point is to use a learning rate in the range <0.1 and >.001.

Subsample

Let's recall the principles of bagging and pasting, where we take random samples and construct trees on those samples. If we apply subsampling to GBM, we randomize the tree constructions and prevent overfitting, reduce memory load, and even sometimes increase accuracy. We can apply this procedure to GBM as well, making it more stochastic and thus leveraging the advantages of bagging. We can randomize the tree construction in GBM by setting the subsample parameter to .5.

Faster GBM with warm_start

This parameter allows storing new tree information after each iteration is added to the previous one without generating new trees. This way, we can save memory and speed up computation time extensively.

Using the GBM available in Scikit-learn, there are two actions that we can take in order to increase memory and CPU efficiency:

- Warm start for (semi) incremental learning

- We can use parallel processing during cross-validation

Let's run through a GBM classification example where we use the spam dataset from the UCI machine learning library. We will first load the data, preprocess it, and look at the variable importance of each feature:

```
import pandas
import urllib2
import urllib2
from sklearn import ensemble
columnNames1_url = 'https://archive.ics.uci.edu/ml/
machine-learning-databases/spambase/spambase.names'
columnNames1 = [
   line.strip().split(':')[0]
   for line in urllib2.urlopen(columnNames1_url).readlines()[33:]]

columnNames1
n = 0
for i in columnNames1:
   columnNames1[n] = i.replace('word_freq_','')
   n += 1
print columnNames1

spamdata = pandas.read_csv(
   'https://archive.ics.uci.edu/ml/machine-learning-databases/
spambase/spambase.data',
   header=None, names=(columnNames1 + ['spam'])
)

X = spamdata.values[:,:57]
y=spamdata['spam']

spamdata.head()

import numpy as np
from sklearn import cross_validation
from sklearn.metrics import classification_report
from sklearn.cross_validation import cross_val_score
from sklearn.cross_validation import cross_val_predict
from sklearn.cross_validation import train_test_split
from sklearn.metrics  import recall_score, f1_score
from sklearn.cross_validation import cross_val_predict
import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix
from sklearn.metrics import accuracy_score
from sklearn.ensemble import GradientBoostingClassifier
```

```
X_train, X_test, y_train, y_test = train_test_split(X,y,
test_size=0.3, random_state=22)

clf =
ensemble.GradientBoostingClassifier(n_estimators=300,random_state=222
,max_depth=16,learning_rate=.1,subsample=.5)
scores=clf.fit(X_train,y_train)
scores2 = cross_val_score(clf, X_train, y_train, cv=3,
scoring='accuracy',n_jobs=-1)
print scores2.mean()

y_pred = cross_val_predict(clf, X_test, y_test, cv=10)
print 'validation accuracy %s' % accuracy_score(y_test, y_pred)

OUTPUT:]
validation accuracy 0.928312816799

confusionMatrix = confusion_matrix(y_test, y_pred)
print confusionMatrix

from sklearn.metrics import accuracy_score
accuracy_score(y_test, y_pred)

clf.feature_importances_

def featureImp_order(clf, X, k=5):
    return X[:,clf.feature_importances_.argsort()[::-1][:k]]
newX = featureImp_order(clf,X,2)
print newX

# let's order the features in amount of importance

print sorted(zip(map(lambda x: round(x, 4),
clf.feature_importances_), columnNames1),
    reverse=True)
OUTPUT]

0.945030177548
    precision    recall  f1-score   support

    0       0.93      0.96      0.94       835
    1       0.93      0.88      0.91       546

avg / total       0.93      0.93      0.93      1381

[[799  36]
 [ 63 483]]
```

```
Feature importance:

[(0.2262, 'char_freq_;'),
(0.0945, 'report'),
(0.0637, 'capital_run_length_average'),
(0.0467, 'you'),
(0.0461, 'capital_run_length_total')
(0.0403, 'business')
(0.0397, 'char_freq_!')
(0.0333, 'will')
(0.0295, 'capital_run_length_longest')
(0.0275, 'your')
(0.0259, '000')
(0.0257, 'char_freq_(')
(0.0235, 'char_freq_$')
(0.0207, 'internet')
```

We can see that the character `;` is the most discriminative in classifying spam.

Note

Variable importance shows us how much splitting each feature reduces the relative impurity across all the splits in the tree.

Speeding up GBM with warm_start

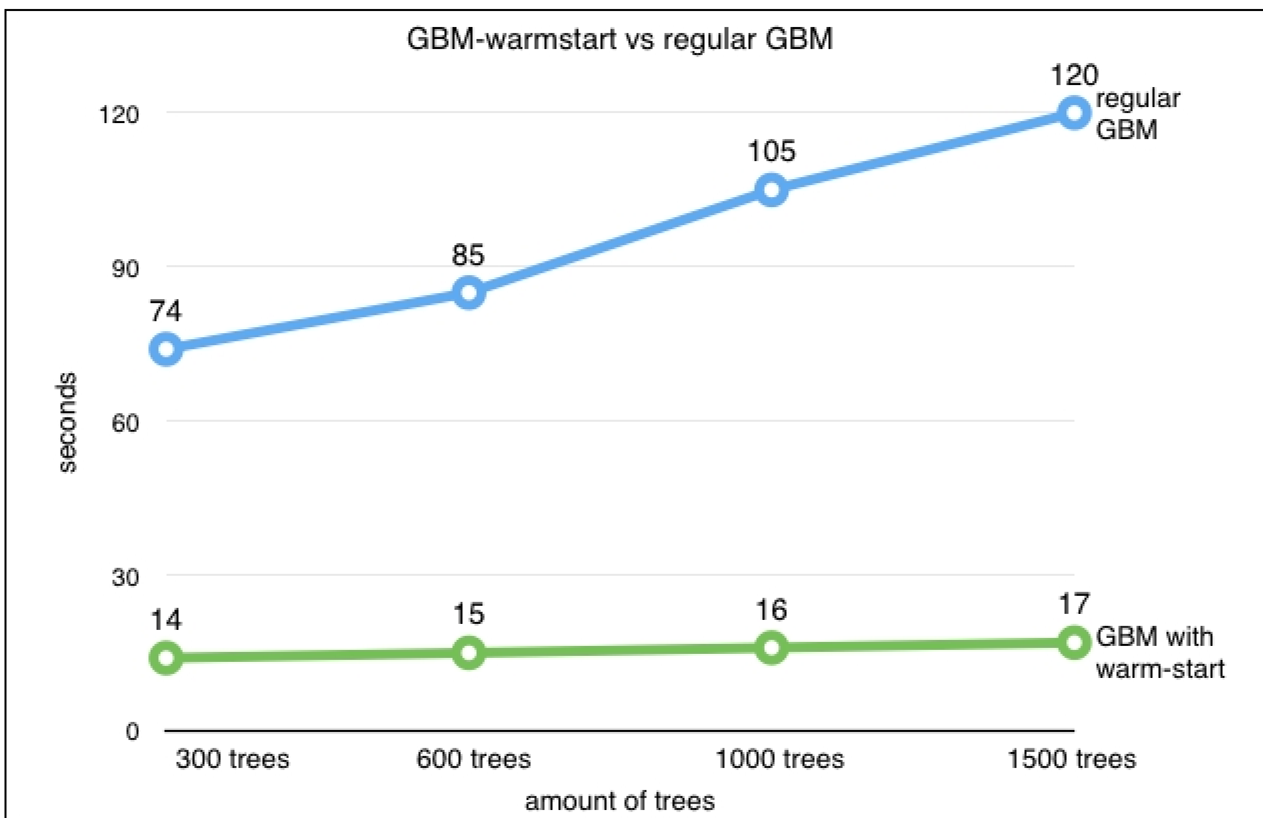
Unfortunately, there is no parallel processing for GBM in Scikit-learn. Only cross-validation and gridsearch can be parallelized. So what can we do to make it faster? We saw that GBM works with the principle of additive expansion where trees are added incrementally. We can utilize this idea in Scikit-learn with the `warm-start` parameter. We can model this with Scikit-learn's GBM functionalities by building tree models incrementally with a handy for loop. So let's do this with the same dataset and inspect the computational advantage that it provides:

```
gbc = GradientBoostingClassifier(warm_start=True, learning_rate=.05,
max_depth=20,random_state=0)
for n_estimators in range(1, 1500, 100):
    gbc.set_params(n_estimators=n_estimators)
    gbc.fit(X_train, y_train)
y_pred = gbc.predict(X_test)
print(classification_report(y_test, y_pred))
print(gbc.set_params)
OUTPUT:
 precision    recall  f1-score   support

    0       0.93      0.95      0.94       835
    1       0.92      0.89      0.91       546
```

```
avg / total       0.93      0.93      0.93      1381

<bound method GradientBoostingClassifier.set_params of
GradientBoostingClassifier(init=None, learning_rate=0.05,
loss='deviance',
    max_depth=20, max_features=None, max_leaf_nodes=None,
    min_samples_leaf=1, min_samples_split=2,
    min_weight_fraction_leaf=0.0, n_estimators=1401,
    presort='auto', random_state=0, subsample=1.0, verbose=0,
    warm_start=True)>
```

It's adviable to pay special attention to the output of the settings of the tree (`n_estimators=1401`). You can see that the size of the tree we used is 1401. This little trick helped us reduce the training time a lot (think half or even less) when we compare it to a similar GBM model, which we would have trained with 1401 trees at once. Note that we can use this method for random forest and extreme random forest as well. Nevertheless, we found this useful specifically for GBM.

Let's look at the figure that shows the training time for a regular GBM and our `warm_start` method. The computation speedup is considerable with the accuracy remaining relatively the same:



Training and storing GBM models

Ever thought of training a model on three computers at the same time? Or training a GBM model on an EC2 instance? There might be a case where you train a model and want to store it to use it again later on. When you have to wait for two days for a full training round to complete, we don't want to have to go through that process again. A case where you have trained a model in the cloud on an Amazon EC2 instance, you can store that model and reuse it later on another computer using Scikit-learn's `joblib`. So let's walk through this process as Scikit-learn has provided us with a handy tool to manage it.

Let's import the right libraries and set our directories for our file locations:

```
import errno
import os
#set your path here
path='/yourpath/clfs'
clfm=os.makedirs(path)
os.chdir(path)
```

Now let's export the model to the specified location on our harddrive:

```
from sklearn.externals import joblib
joblib.dump( gbc,'clf_gbc.pkl')
```

Now we can load the model and reuse it for other purposes:

```
model_clone = joblib.load('clf_gbc.pkl')
zpred=model_clone.predict(X_test)
print zpred
```

XGBoost

We have just discussed that there are no options for parallel processing when using GBM from Scikit-learn, and this is exactly where XGBoost comes in. Expanding on GBM, XGBoost introduces more scalable methods leveraging multithreading on a single machine and parallel processing on clusters of multiple servers (using sharding). The most important improvement of XGBoost over GBM lies in the capability of the latter to manage sparse data. XGBoost automatically accepts sparse data as input without storing zero values in memory. A second benefit of XGBoost lies in the way in which the best node split values are calculated while branching the tree, a method named quantile sketch. This method transforms the data by a weighting algorithm so that candidate splits are sorted based on a certain accuracy level. For more information read the article at http://arxiv.org/pdf/1603.02754v3.pdf.

XGBoost stands for Extreme Gradient Boosting, an open source gradient boosting algorithm that has gained a lot of popularity in data science competitions such as Kaggle (https://www.kaggle.com/) and KDD-cup 2015. (The code is available on GitHub at https://github.com/dmlc/XGBoost, as we described in Chapter 1, *First Steps to Scalability*.) As the authors (Tianqui Chen, Tong He, and Carlos Guestrin) report on papers that they wrote on their algorithm, XGBoost, among 29 challenges held on Kaggle during 2015, 17 winning solutions used XGBoost as a standalone or part of some kind of ensemble of multiple models. In their paper *XGBoost: A Scalable Tree Boosting System* (which can be found at http://learningsys.org/papers/LearningSys_2015_paper_32.pdf), the authors report that, in the recent KDD-cup 2015, XGBoost was used by every team that ended in the top ten of the competition. Apart from successful performances in both accuracy and computational efficiency, our principal concern in this book is scalability, and XGBoost is indeed a scalable solution from different points of view. XGBoost is a new generation of GBM algorithms with important tweaks to the initial tree boost GBM algorithm. XGBoost provides parallel processing; the scalability offered by the algorithm is due to quite a few new tweaks and additions developed by its authors:

- An algorithm that accepts sparse data, which can leverage sparse matrices, saving both memory (no need for dense matrices) and computation time (zero values are handled in a special way)
- An approximate tree learning (weighted quantile sketch), which bears similar results but in much less time than the classical complete explorations of possible branch cuts
- Parallel computing on a single machine (using multithreading in the phase of the search for the best split) and similarly distributed computations on multiple ones
- Out-of-core computations on a single machine leveraging a data storage solution called Column Block, which arranges data on disk by columns, thus saving time by pulling data from disk as the optimization algorithm (which works on column vectors) expects it

From a practical point of view, XGBoost features mostly the same parameters as GBM. XGBoost is also quite capable of dealing with missing data. Other tree ensembles, based on standard decisions trees, require missing data first to be imputed using an off-scale value (such as a large negative number) in order to develop an appropriate branching of the tree to deal with missing values. XGBoost, instead, first fits all the non-missing values and, after having created the branching for the variable, it decides which branch is better for the missing values to take in order to minimize the prediction error. Such an approach leads to trees that are more compact and an effective imputation strategy leading to more predictive power.

The most important XGBoost parameters are as follows:

- `eta` (default=0.3): This is the equivalent of the learning rate in Scikit-learn's GBM
- `min_child_weight` (default=1): Higher values prevent overfitting and tree complexity
- `max_depth` (default=6): This is the number of interactions in the trees
- `subsample` (default=1): This is a fraction of samples of the training data that we take in each iteration
- `colsample_bytree` (default=1): This is the fraction of features in each iteration
- `lambda` (default=1): This is the L2 regularization (Boolean)
- `seed` (default=0): This is the equivalent of Scikit-learn's `random_state` parameter, allowing reproducibility of learning processes across multiple tests and different machines

Now that we know XGBoost's most important parameters, let's run an XGBoost example on the same dataset that we used for GBM with the same parameter settings (as much as possible). XGBoost is a little less straightforward to use than the Scikit-learn package. So we will provide some basic examples that you can use as a starting point for more complex models. Before we dive deeper into XGBoost applications, let's compare it to the GBM method in `sklearn` on the spam dataset; we have already loaded the data in-memory:

```
import xgboost as xgb
import numpy as np
from sklearn.metrics import classification_report
from sklearn import cross_validation

clf = xgb.XGBClassifier(n_estimators=100,max_depth=8,
    learning_rate=.1,subsample=.5)

clf1 = GradientBoostingClassifier(n_estimators=100,max_depth=8,
    learning_rate=.1,subsample=.5)

%timeit xgm=clf.fit(X_train,y_train)
%timeit gbmf=clf1.fit(X_train,y_train)

y_pred = xgm.predict(X_test)
y_pred2 = gbmf.predict(X_test)

print 'XGBoost results %r' % (classification_report(y_test, y_pred))
print 'gbm results %r' % (classification_report(y_test, y_pred2))

OUTPUT:
1 loop, best of 3: 1.71 s per loop
1 loop, best of 3: 2.91 s per loop
XGBoost results '             precision    recall  f1-score
support\n\n          0       0.95      0.97      0.96       835\
n          1       0.95      0.93      0.94       546\n\navg /
total       0.95      0.95      0.95      1381\n'
gbm results '             precision    recall  f1-score   support\n\
n          0       0.95      0.97      0.96       835\n
```

```
1       0.95      0.92      0.93       546\n\navg / total
0.95      0.95      0.95      1381\n
```

We can clearly see that XGBoost is quite faster than GBM (1.71s versus 2.91s) even though we didn't use parallelization for XGBoost. Later, we can even arrive at a greater speedup when we use parallelization and out-of-core methods for XGBoost when we apply out-of-memory streaming. In some cases, the XGBoost model results in a higher accuracy than GBM, but (almost) never the other way around.

XGBoost regression

Boosting methods are often used for classification but can be very powerful for regression tasks as well. As regression is often overlooked, let's run a regression example and walk through the key issues. Let's fit a boosting model on the California housing set with gridsearch. The California house dataset has recently been added to Scikit-learn, which saves us some preprocessing steps:

```
import numpy as np
import scipy.sparse
import xgboost as xgb
import os
import pandas as pd
from sklearn.cross_validation import train_test_split
import numpy as np
from sklearn.datasets import fetch_california_housing
from sklearn.metrics import mean_squared_error
pd=fetch_california_housing()

#because the y  variable is highly skewed we apply the log
transformation
y=np.log(pd.target)
X_train, X_test, y_train, y_test = train_test_split(pd.data,
    y,
    test_size=0.15,
    random_state=111)
names = pd.feature_names
print names

import xgboost as xgb
from xgboost.sklearn import XGBClassifier
from sklearn.grid_search import GridSearchCV

clf=xgb.XGBRegressor(gamma=0,objective= "reg:linear",nthread=-1)


clf.fit(X_train,y_train)
y_pred = clf.predict(X_test)
print 'score before gridsearch %r' % mean_squared_error(y_test,
```

```
y_pred)

params = {
 'max_depth':[4,6,8],
 'n_estimators':[1000],
'min_child_weight':range(1,3),
'learning_rate':[.1,.01,.001],
'colsample_bytree':[.8,.9,1]
,'gamma':[0,1]}

#with the parameter nthread we specify XGBoost for parallelisation
cvx = xgb.XGBRegressor(objective= "reg:linear",nthread=-1)
clf=GridSearchCV(estimator=cvx,param_grid=params,n_jobs=-1,scoring='m
ean_absolute_error',verbose=True)

clf.fit(X_train,y_train)
print clf.best_params_
y_pred = clf.predict(X_test)
print 'score after gridsearch %r' %mean_squared_error(y_test, y_pred)


#Your output might look a little different based on your hardware.

OUTPUT
['MedInc', 'HouseAge', 'AveRooms', 'AveBedrms', 'Population',
'AveOccup', 'Latitude', 'Longitude']
score before gridsearch 0.07110580252173157
Fitting 3 folds for each of 108 candidates, totalling 324 fits
[Parallel(n_jobs=-1)]: Done  34 tasks      | elapsed:  1.9min
[Parallel(n_jobs=-1)]: Done 184 tasks      | elapsed: 11.3min
[Parallel(n_jobs=-1)]: Done 324 out of 324 | elapsed: 22.3min
finished
{'colsample_bytree': 0.8, 'learning_rate': 0.1, 'min_child_weight':
1, 'n_estimators': 1000, 'max_depth': 8, 'gamma': 0}
score after gridsearch 0.04987829413796254
```

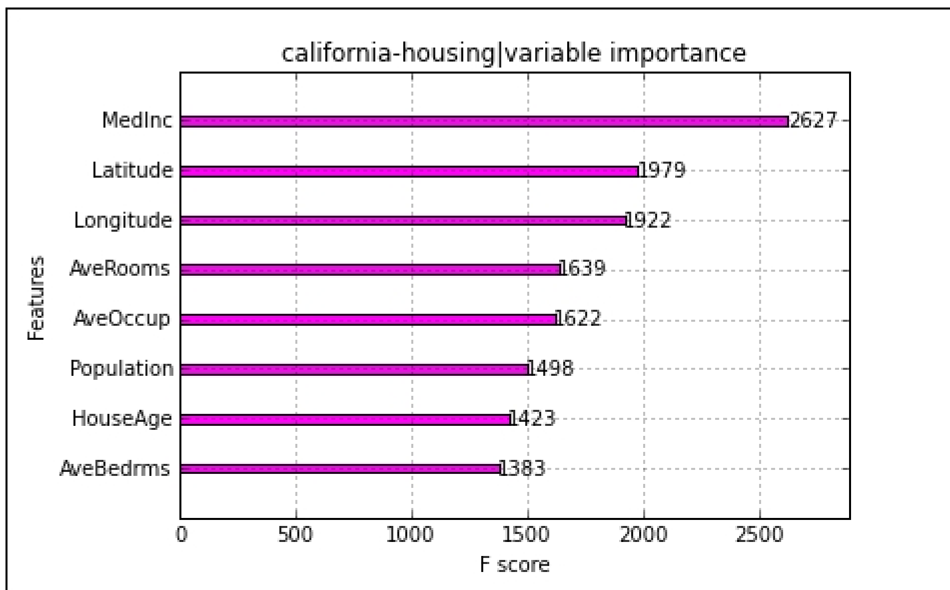
We have been able to improve our score quite a bit with gridsearch; you can see the optimal parameters of our gridsearch. You can see its resemblance with regular boosting methods in sklearn. However, XGBoost by default parallelizes the algorithm over all available cores. You can improve the performance of the model by increasing the `n_estimators` parameter to around 2,500 or 3,000. However, we found that the training time would be a little too long for readers with less powerful computers.

XGBoost and variable importance

XGBoost has some very practical built-in functionalities to plot the variable importance. First, there's an handy tool for feature selection relative to the model at hand. As you probably know, variable

importance is based on the relative influence of each feature at the tree construction. It provides practical methods for feature selection and insight into the nature of the predictive model. So let's see how we can plot importance with XGBoost:

```
import numpy as np
import os
from matplotlib import pylab as plt
# %matplotlib inline   <- this only works in jupyter notebook

#our best parameter set
# {'colsample_bytree': 1, 'learning_rate': 0.1, 'min_child_weight':
1, 'n_estimators': 500, #'max_depth': 8, 'gamma': 0}

params={'objective': "reg:linear",
        'eval_metric': 'rmse',
        'eta': 0.1,
       'max_depth':8,
       'min_samples_leaf':4,
        'subsample':.5,
        'gamma':0
       }

dm = xgb.DMatrix(X_train, label=y_train,
                 feature_names=names)
regbgb = xgb.train(params, dm, num_boost_round=100)
np.random.seed(1)
regbgb.get_fscore()


regbgb.feature_names
regbgb.get_fscore()
xgb.plot_importance(regbgb,color='magenta',title='california-housing|
variable importance')
```



Feature importance should be used with some caution (for GBM and random forest as well). The feature importance metrics are purely based on the tree structure built on the specific model trained with the parameters of that model. This means that if we change the parameters of the model, the importance metrics, and some of the rankings, will change as well. Therefore, it's important to note that for any importance metric, they should not be taken as a generic variable conclusion that generalizes across models.

XGBoost streaming large datasets

In terms of accuracy/performance trade-off, this is simply the best desktop solution. We saw that, with the previous random forest example, we needed to perform subsampling in order to prevent overloading our main memory.

An often-overlooked capability of XGBoost is the method of streaming data through memory. This method parses data through the main memory in a stage-wise fashion to subsequently be parsed into XGBoost model training. This method is a prerequisite to train models on large datasets that are impossible to fit in the main memory. Streaming with XGBoost only works with LIBSVM files, which means that we first have to parse our dataset to the LIBSVM format and import it in the memory cache preserved for XGBoost. Another thing to note is that we use different methods to instantiate XGBoost models. The Scikit-learn-like interface for XGBoost only works on regular NumPy objects. So let's look at how this works.

First, we need to load our dataset in the LIBSVM format and split it into train and test sets before we proceed with preprocessing and training. Parameter tuning with gridsearch is unfortunately not possible with this XGBoost method. If you want to tune parameters, we need to transform the LIBSVM file into a Numpy object, which will dump the data from the memory cache to the main memory. This is unfortunately not scalable, so if you want to perform tuning on large datasets, I would suggest using the reservoir sampling tools we previously introduced and apply tuning to subsamples:

```
import urllib
from sklearn.datasets import dump_svmlight_file
from sklearn.datasets import load_svmlight_file
trainfile = urllib.URLopener()
trainfile.retrieve("http://www.csie.ntu.edu.tw/~cjlin/libsvmtools/
datasets/multiclass/poker.bz2", "pokertrain.bz2")
X,y = load_svmlight_file('pokertrain.bz2')
dump_svmlight_file(X, y,'pokertrain', zero_based=True,query_id=None,
multilabel=False)
testfile = urllib.URLopener()
testfile.retrieve("http://www.csie.ntu.edu.tw/~cjlin/libsvmtools/
datasets/multiclass/poker.t.bz2", "pokertest.bz2")
X,y = load_svmlight_file('pokertest.bz2')
dump_svmlight_file(X, y,'pokertest', zero_based=True,query_id=None,
multilabel=False)
del(X,y)
from sklearn.metrics import classification_report
import numpy as np
import xgboost as xgb
dtrain = xgb.DMatrix('/yourpath/pokertrain#dtrain.cache')
dtest = xgb.DMatrix('/yourpath/pokertest#dtestin.cache')

# For parallelisation it is better to instruct "nthread" to match
the exact amount of cpu cores you want #to use.
param =
{'max_depth':8,'objective':'multi:softmax','nthread':2,'num_class':10
,'verbose':True}
num_round=100
watchlist = [(dtest,'eval'), (dtrain,'train')]
bst = xgb.train(param, dtrain, num_round,watchlist)
print bst
OUTPUT:
[89]    eval-merror:0.228659    train-merror:0.016913
[90]    eval-merror:0.228599    train-merror:0.015954
[91]    eval-merror:0.227671    train-merror:0.015354
[92]    eval-merror:0.227777    train-merror:0.014914
[93]    eval-merror:0.226247    train-merror:0.013355
[94]    eval-merror:0.225397    train-merror:0.012155
[95]    eval-merror:0.224070    train-merror:0.011875
[96]    eval-merror:0.222421    train-merror:0.010676
[97]    eval-merror:0.221881    train-merror:0.010116
[98]    eval-merror:0.221922    train-merror:0.009676
[99]    eval-merror:0.221733    train-merror:0.009316
```

We can really experience a great speedup from in-memory XGBoost. We would need a lot more training time if we had used the internal memory version. In this example, we already included the test set as a

validation round in the *watchlist*. However, if we want to predict values on unseen data, we can simply use the same prediction procedure as with any other model in Scikit-learn and XGBoost:

```
bst.predict(dtest)
OUTPUT:
array([ 0.,  0.,  1., ...,  0.,  0.,  1.], dtype=float32)
```

XGBoost model persistence

In the previous chapter, we covered how to store a GBMmodel to disk to later import and use it for predictions. XGBoost provides the same functionalities. So let's see how we can store and import the model:

```
import pickle
bst.save_model('xgb.model')
```

Now you can import the saved model from the directory that you previously specified:

```
imported_model = xgb.Booster(model_file='xgb.model')
```

Great, now you can use this model for predictions:

```
imported_model.predict(dtest)
OUTPUT array([ 9.,  9.,  9., ...,  1.,  1.,  1.], dtype=float32)
```

Out-of-core CART with H2O

Up until now, we have only dealt with desktop solutions for CART models. In Chapter 4, *Neural Networks and Deep Learning*, we introduced H2O for deep learning out of memory that provided a powerful scalable method. Luckily, H2O also provides tree ensemble methods utilizing its powerful parallel Hadoop ecosystem. As we covered GBM and random forest extensively in previous sections, let's get to it right away. For this exercise, we will use the spam dataset that we used before.

Random forest and gridsearch on H2O

Let's implement a random forest with gridsearch hyperparameter optimization. In this section, we first load the spam dataset from the URL source:

```
import pandas as pd
import numpy as np
import os
import xlrd
import urllib
import h2o

#set your path here
os.chdir('/yourpath/')

url = 'https://archive.ics.uci.edu/ml/machine-learning-databases/
spambase/spambase.data'
filename='spamdata.data'
urllib.urlretrieve(url, filename)
```

Now that we have loaded the data, we can initialize the H2O session:

```
h2o.init(max_mem_size_GB = 2)

OUTPUT:
```

H2O cluster uptime:	1 days 1 hours 33 minutes 47 seconds 112 milliseconds
H2O cluster version:	3.6.0.8
H2O cluster name:	H2O_started_from_python
H2O cluster total nodes:	1
H2O cluster total memory:	1.78 GB
H2O cluster total cores:	4
H2O cluster allowed cores:	4
H2O cluster healthy:	True
H2O Connection ip:	127.0.0.1
H2O Connection port:	54321

Here, we preprocess the data where we split the data into train, validation, and test sets. We do this with an H2O function (`.split_frame`). Also note the important step where we convert the target vector `C58` to a factor variable:

```
spamdata = h2o.import_file(os.path.realpath("/yourpath/"))
spamdata['C58']=spamdata['C58'].asfactor()
train, valid, test= spamdata.split_frame([0.6,.2], seed=1234)
spam_X = spamdata.col_names[:-1]
spam_Y = spamdata.col_names[-1]
```

In this part, we will set up the parameters that we will optimize with gridsearch. First of all, we set the number of trees in the model to a single value of 300. The parameters that are iterated with gridsearch are as follows:

- `max_depth`: The maximum depth of the tree
- `balance_classes`: Each iteration uses balanced classes for the target outcome
- `sample_rate`: This is the fraction of the rows that are sampled for each iteration

Now let's pass these parameters into a Python list to be used in our H2O gridsearch model:

```
hyper_parameters={'ntrees':[300],
'max_depth':[3,6,10,12,50],'balance_classes':['True','False'],'sample
_rate':[.5,.6,.8,.9]}
grid_search = H2OGridSearch(H2ORandomForestEstimator,
```

```
hyper_params=hyper_parameters)
grid_search.train(x=spam_X, y=spam_Y,training_frame=train)
print 'this is the optimum solution for hyper parameters search %s'
% grid_search.show()
OUTPUT:
```

```
drf Grid Build Progress: [##################################################] 100%

Grid Search Results for H2ORandomForestEstimator:
```

Model Id	Hyperparameters: [ntrees, sample_rate, max_depth, balance_classes]	mse
Grid_DRF_py_87_model_python_1466382079157_49_model_19	[300, 0.9, 50, True]	0.0249340
Grid_DRF_py_87_model_python_1466382079157_49_model_18	[300, 0.8, 50, True]	0.0258412
Grid_DRF_py_87_model_python_1466382079157_49_model_17	[300, 0.6, 50, True]	0.0289790
Grid_DRF_py_87_model_python_1466382079157_49_model_16	[300, 0.5, 50, True]	0.0314358
Grid_DRF_py_87_model_python_1466382079157_49_model_38	[300, 0.8, 50, False]	0.0417964
---	---	---
Grid_DRF_py_87_model_python_1466382079157_49_model_23	[300, 0.9, 3, False]	0.0914980
Grid_DRF_py_87_model_python_1466382079157_49_model_1	[300, 0.6, 3, True]	0.1040888
Grid_DRF_py_87_model_python_1466382079157_49_model_0	[300, 0.5, 3, True]	0.1042337
Grid_DRF_py_87_model_python_1466382079157_49_model_2	[300, 0.8, 3, True]	0.1042843
Grid_DRF_py_87_model_python_1466382079157_49_model_3	[300, 0.9, 3, True]	0.1060737

Of all the possible combinations, the model with a row sample rate of .9, a tree depth of 50, and balanced classes yields the highest accuracy. Now let's train a new random forest model with optimal parameters resulting from our gridsearch and predict the outcome on the test set:

```
final = H2ORandomForestEstimator(ntrees=300,
max_depth=50,balance_classes=True,sample_rate=.9)
final.train(x=spam_X, y=spam_Y,training_frame=train)
print final.predict(test)
```

The final output of the predictions in H2O results in an array with the first column containing the actual predicted classes and columns containing the class probabilities of each target label:

OUTPUT:

predict	p0	p1
1	0.531042	0.468958
1	0.510856	0.489144
1	0.51637	0.48363
1	0.542997	0.457003
1	0.544576	0.455424
1	0.560277	0.439723
1	0.544576	0.455424
1	0.5408	0.4592
1	0.535741	0.464259
1	0.498822	0.501178

Stochastic gradient boosting and gridsearch on H2O

We have seen in previous examples that most of the time, a well-tuned GBM model outperforms random forest. So now let's perform a GBM with gridsearch in H2O and see if we can improve our score. For this session, we introduce the same random subsampling method that we used for the random forest model in H2O (`sample_rate`). Based on Jerome Friedman's article (https://statweb.stanford.edu/~jhf/ftp/stobst.pdf) from 1999, a method named **stochastic gradient boosting** was introduced. This stochasticity added to the model utilizes random subsampling without replacement from the data at each tree iteration that is considered to prevent overfitting and increase overall accuracy. In this example, we take this idea of stochasticity further by introducing random subsampling based on the features at each iteration.

This method of randomly subsampling features is also referred to as the **random subspace method**, which we have already seen in the *Random forest and extremely randomized forest* section of this chapter. We achieve this with the `col_sample_rate` parameter. So to summarize, in this GBM model, we are going to perform gridsearch optimization on the following parameters:

- `max_depth`: Maximum tree depth
- `sample_rate`: Fraction of the rows used at each iteration
- `col_sample_rate`: Fraction of the features used at each iteration

We use exactly the same spam dataset as the previous section so we can get right down to it:

```
hyper_parameters={'ntrees':[300],'max_depth':[12,30,50],'sample_rate'
:[.5,.7,1],'col_sample_rate':[.9,1],
'learn_rate':[.01,.1,.3],}
grid_search = H2OGridSearch(H2OGradientBoostingEstimator,
hyper_params=hyper_parameters)
grid_search.train(x=spam_X, y=spam_Y, training_frame=train)
print 'this is the optimum solution for hyper parameters search %s'
% grid_search.show()
```

```
gbm Grid Build Progress: [####################################################] 100%

Grid Search Results for H2OGradientBoostingEstimator:
```

Model Id	Hyperparameters: [learn_rate, col_sample_rate, ntrees, sample_rate, max_depth]	mse
Grid_GBM_py_87_model_python_1466382079157_52_model_23	[0.3, 0.9, 300, 1.0, 30]	0.0001859
Grid_GBM_py_87_model_python_1466382079157_52_model_20	[0.3, 0.9, 300, 1.0, 12]	0.0001859
Grid_GBM_py_87_model_python_1466382079157_52_model_47	[0.3, 1.0, 300, 1.0, 12]	0.0001859
Grid_GBM_py_87_model_python_1466382079157_52_model_26	[0.3, 0.9, 300, 1.0, 50]	0.0001859
Grid_GBM_py_87_model_python_1466382079157_52_model_53	[0.3, 1.0, 300, 1.0, 50]	0.0001859
---	---	---
Grid_GBM_py_87_model_python_1466382079157_52_model_33	[0.01, 1.0, 300, 0.5, 50]	0.0196867
Grid_GBM_py_87_model_python_1466382079157_52_model_6	[0.01, 0.9, 300, 0.5, 50]	0.0197013

```
gbm Grid Build Progress:
[##################################################] 100%
```

The upper part of our gridsearch output shows that we should use an exceptionally high learning rate of .3, a column sample rate of .9, and a maximum tree depth of 30. Random subsampling based on rows didn't increase performance, but subsampling based on features with a fraction of .9 was quite effective

in this case. Now let's train a new GBM model with the optimal parameters resulting from our gridsearch optimization and predict the outcome on the test set:

```
spam_gbm2 = H2OGradientBoostingEstimator(
  ntrees=300,
  learn_rate=0.3,
  max_depth=30,
  sample_rate=1,
  col_sample_rate=0.9,
  score_each_iteration=True,
  seed=2000000
)
spam_gbm2.train(spam_X, spam_Y, training_frame=train,
validation_frame=valid)

confusion_matrix = spam_gbm2.confusion_matrix(metrics="accuracy")
print confusion_matrix
OUTPUT:
```

```
gbm Model Build Progress: [##################################################] 100%

Confusion Matrix (Act/Pred) for max accuracy @ threshold = 0.99983413575:
```

	0	1	Error	Rate
0	1639.0	0.0	0.0	(0.0/1639.0)
1	1.0	1050.0	0.001	(1.0/1051.0)
Total	1640.0	1050.0	0.0004	(1.0/2690.0)

This delivers interesting diagnostics of the model's performance such as `accuracy`, `rmse`, `logloss`, and `AUC`. However, its output is too large to include here. Look at the output of your IPython notebook for the complete output.

You can utilize this with the following:

```
print spam_gbm2.score_history()
```

Of course, the final predictions can be achieved as follows:

```
print spam_gbm2.predict(test)
```

Great, we have been able to improve the accuracy of our model close to 100%. As you can see, in H2O, you might be less flexible in terms of modeling and munging your data, but the speed of processing and accuracy that can be achieved is unrivaled. To round off this session, you can do the following:

```
h2o.shutdown(prompt=False)
```

Summary

We saw that CART methods trained with ensemble routines are powerful when it comes to predictive accuracy. However, they can be computationally expensive and we have covered some techniques in speeding them up in sklearn's applications. We noticed that using extreme randomized forests tuned with randomized search could speed up by tenfold when used properly. For GBM, however, there is no parallelization implemented in sklearn, and this is exactly where XGBoost comes in.

XGBoost comes with an effective parallelized boosting algorithm speeding up the algorithm nicely. When we use larger files (>100k training examples), there is an out-of-core method that makes sure we don't overload our main memory while training models.

The biggest gains in speed and memory can be found with H2O; we saw powerful tuning capabilities together with an impressive training speed.

Chapter 7. Unsupervised Learning at Scale

In the previous chapters, the focus of the problem was on predicting a variable, which could have been a number, class, or category. In this chapter, we will change the approach and try to create new features and variables at scale, hopefully better for our prediction purposes than the ones already included in the observation matrix. We will first introduce the unsupervised methods and illustrate three of them, which are able to scale to big data:

- **Principal Component Analysis** (**PCA**), an effective way to reduce the number of features
- **K-means**, a scalable algorithm for clustering
- **Latent Dirichlet Allocation** (**LDA**), a very effective algorithm able to extract topics from a series of text documents

Unsupervised methods

Unsupervised learning is a branch of machine learning whose algorithms reveal inferences from data without an explicit label (unlabeled data). The goal of such techniques is to extract hidden patterns and group similar data.

In these algorithms, the unknown parameters of interests of each observation (the group membership and topic composition, for instance) are often modeled as latent variables (or a series of hidden variables), hidden in the system of observed variables that cannot be observed directly, but only deduced from the past and present outputs of the system. Typically, the output of the system contains noise, which makes this operation harder.

In common problems, unsupervised methods are used in two main situations:

- With labeled datasets to extract additional features to be processed by the classifier/regressor down to the processing chain. Enhanced by additional features, they may perform better.
- With labeled or unlabeled datasets to extract some information about the structure of the data. This class of algorithms is commonly used during the **Exploratory Data Analysis** (**EDA**) phase of the modeling.

First of all, before starting with our illustration, let's import the modules that will be necessary along the chapter in our notebook:

```
In : import matplotlib
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from matplotlib import pylab
%matplotlib inline
import matplotlib.cm as cm
import copy
import tempfile
import os
```

Feature decomposition – PCA

PCA is an algorithm commonly used to decompose the dimensions of an input signal and keep just the *principal* ones. From a mathematical perspective, PCA performs an orthogonal transformation of the observation matrix, outputting a set of linear uncorrelated variables, named principal components. The output variables form a basis set, where each component is orthonormal to the others. Also, it's possible to rank the output components (in order to use just the principal ones) as the first component is the one containing the largest possible variance of the input dataset, the second is orthogonal to the first (by definition) and contains the largest possible variance of the residual signal, and the third is orthogonal to the first two and it's built on the residual variance, and so on.

A generic transformation with PCA can be expressed as a projection to a space. If just the principal components are taken from the transformation basis, the output space will have a smaller dimensionality than the input one. Mathematically, it can be expressed as follows:

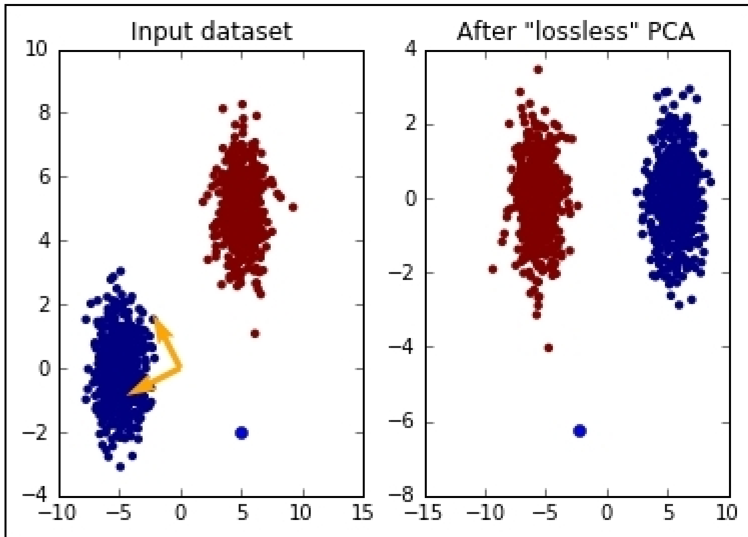
$$\hat{X} = X \cdot T$$

Here, X is a generic point of the training set of dimension N, T is the transformation matrix coming from PCA, and $\hat{X}$ is the output vector. Note that the symbol indicates a dot product in this matrix equation. From a practical perspective, also note that all the features of X must be zero-centered before doing this operation.

Let's now start with a practical example; later, we will explain math PCA in depth. In this example, we will create a dummy dataset composed of two blobs of points—one cantered in (-5, 0) and the other one in (5, 5).Let's use PCA to transform the dataset and plot the output compared to the input. In this simple example, we will use all the features, that is, we will not perform feature reduction:

```
In:from sklearn.datasets.samples_generator import make_blobs
from sklearn.decomposition import PCA

X, y = make_blobs(n_samples=1000, random_state=101, \
centers=[[-5, 0], [5, 5]])
pca = PCA(n_components=2)
X_pca = pca.fit_transform(X)
pca_comp = pca.components_.T

test_point = np.matrix([5, -2])
test_point_pca = pca.transform(test_point)

plt.subplot(1, 2, 1)
plt.scatter(X[:, 0], X[:, 1], c=y, edgecolors='none')
plt.quiver(0, 0, pca_comp[:,0], pca_comp[:,1], width=0.02, \
```

```
            scale=5, color='orange')
plt.plot(test_point[0, 0], test_point[0, 1], 'o')
plt.title('Input dataset')

plt.subplot(1, 2, 2)
plt.scatter(X_pca[:, 0], X_pca[:, 1], c=y, edgecolors='none')
plt.plot(test_point_pca[0, 0], test_point_pca[0, 1], 'o')
plt.title('After "lossless" PCA')

plt.show()
```



As you can see, the output is more organized than the original features' space and, if the next task is a classification, it would require just one feature of the dataset, saving almost 50% of the space and computation needed. In the image, you can clearly see the core of PCA: it's just a projection of the input dataset to the transformation basis drawn in the image on the left in orange. Are you unsure about this? Let's test it:

```
In:print "The blue point is in", test_point[0, :]
print "After the transformation is in", test_point_pca[0, :]
print "Since (X-MEAN) * PCA_MATRIX = ", np.dot(test_point - \
pca.mean_, pca_comp)

Out:The blue point is in [[ 5 -2]]
After the transformation is in [-2.34969911 -6.2575445 ]
Since (X-MEAN) * PCA_MATRIX =  [[-2.34969911 -6.2575445 ]
```

Now, let's dig into the core problem: how is it possible to generate T from the training set? It should contain orthonormal vectors, and the vectors should be ranked according to the quantity of variance (that is, the energy or information carried by the observation matrix) that they can explain. Many solutions

have been implemented, but the most common implementation is based on **Singular Value Decomposition** (**SVD**).

SVD is a technique that decomposes any matrix *M* into three matrixes (U, Σ, W) with special properties and whose multiplication gives back *M* again:

$$M = U \cdot \Sigma \cdot W^T$$

Specifically, given *M*, a matrix of *m* rows and *n* columns, the resulting elements of the equivalence are as follows:

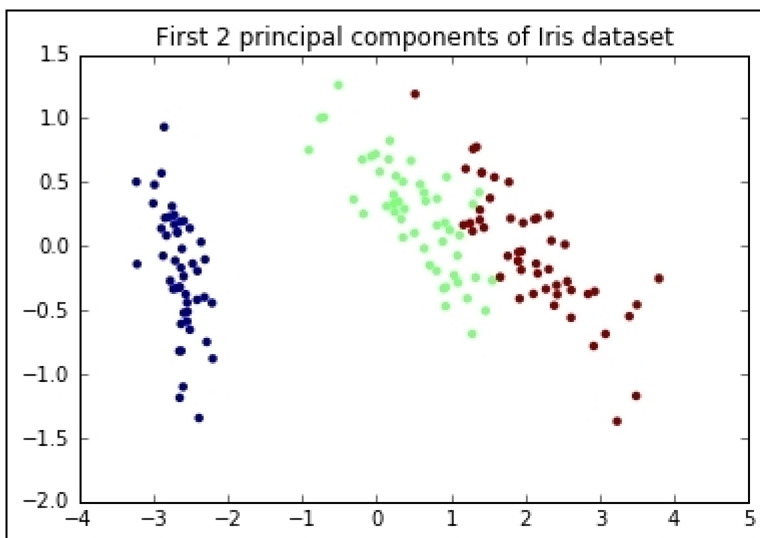
- *U* is a matrix *m x m* (square matrix), it's unitary, and its columns form an orthonormal basis. Also, they're named left singular vectors, or input singular vectors, and they're the eigenvectors of the matrix product $M \cdot M^T$.
- Σ is a matrix *m x n*, which has only non-zero elements on its diagonal. These values are named singular values, are all non-negative, and are the eigenvalues of both $M \cdot M^T$ and $M^T \cdot M$.
- W is a unitary matrix *n x n* (square matrix), its columns form an orthonormal basis, and they're named right (or output) singular vectors. Also, they are the eigenvectors of the matrix product $M^T \cdot M$.

Why is this needed? The solution is pretty easy: the goal of PCA is to try and estimate the directions where the variance of the input dataset is larger. For this, we first need to remove the mean from each feature and then operate on the covariance matrix $X^T \cdot X$.

Given that, by decomposing the matrix *X* with SVD, we have the columns of the matrix *W* that are the principal components of the covariance (that is, the matrix *T* we are looking for), the diagonal of Σ that contains the variance explained by the principal components, and the columns of *U* the principal components. Here's why PCA is always done with SVD.

Let's see it now on a real example. Let's test it on the Iris dataset, extracting the first two principal components (that is, passing from a dataset composed by four features to one composed by two):

```
In:from sklearn import datasets

iris = datasets.load_iris()
X = iris.data
y = iris.target

print "Iris dataset contains", X.shape[1], "features"

pca = PCA(n_components=2)
X_pca = pca.fit_transform(X)

print "After PCA, it contains", X_pca.shape[1], "features"
print "The variance is [% of original]:", \
        sum(pca.explained_variance_ratio_)


plt.scatter(X_pca[:, 0], X_pca[:, 1], c=y, edgecolors='none')
plt.title('First 2 principal components of Iris dataset')

plt.show()

Out:Iris dataset contains 4 features
After PCA, it contains 2 features
The variance is [% of original]: 0.977631775025
```



This is the analysis of the outputs of the process:

- The explained variance is almost 98% of the original variance from the input. The number of features has been halved, but only 2% of the information is not in the output, hopefully just noise.
- From a visual inspection, it seems that the different classes, composing the Iris dataset, are separated from each other. This means that a classifier working on such a reduced set will have comparable performance in terms of accuracy, but will be faster to train and run prediction.

As a proof of the second point, let's now try to train and test two classifiers, one using the original dataset and another using the reduced set, and print their accuracy:

```
In:from sklearn.linear_model import SGDClassifier
from sklearn.cross_validation import train_test_split
from sklearn.metrics import accuracy_score

def test_classification_accuracy(X_in, y_in):
    X_train, X_test, y_train, y_test = \
        train_test_split(X_in, y_in, random_state=101, \
        train_size=0.50)

    clf = SGDClassifier('log', random_state=101)
clf.fit(X_train, y_train)

    return accuracy_score(y_test, clf.predict(X_test))

print "SGDClassifier accuracy on Iris set:", \
            test_classification_accuracy(X, y)
print "SGDClassifier accuracy on Iris set after PCA (2
components):", \
            test_classification_accuracy(X_pca, y)

Out:SGDClassifier accuracy on Iris set: 0.586666666667
SGDClassifier accuracy on Iris set after PCA (2 components): 0.72
```

As you can see, this technique not only reduces the complexity and space of the learner down in the chain, but also helps achieve generalization (exactly as a Ridge or Lasso regularization).

Now, if you are unsure how many components should be in the output, typically as a rule of thumb, choose the minimum number that is able to explain at least 90% (or 95%) of the input variance. Empirically, such a choice usually ensures that only the noise is cut off.

So far, everything seems perfect: we found a great solution to reduce the number of features, building some with very high predictive power, and we also have a rule of thumb to guess the right number of them. Let's now check how scalable this solution is: we're investigating how it scales when the number of observations and features increases. The first thing to note is that the SVD algorithm, the *core* piece of PCA, is not stochastic; therefore, it needs the whole matrix in order to be able to extract its principal components. Now, let's see how scalable PCA is in practice on some synthetic datasets with an increasing number of features and observations. We will perform a full (lossless) decomposition (the

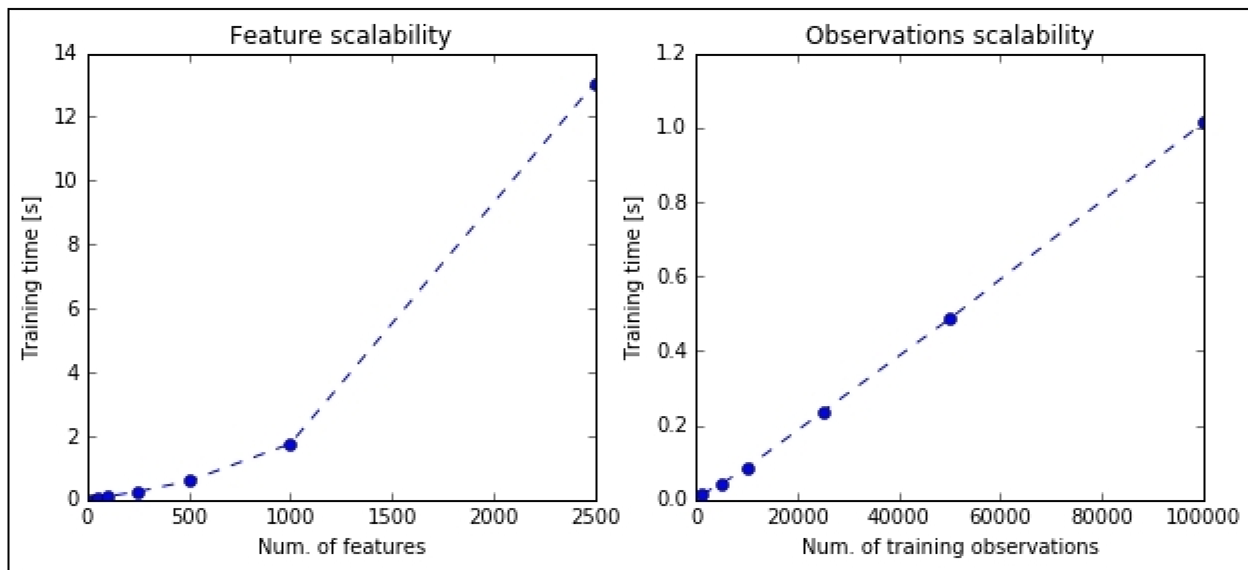
argument while instantiating the object PCA is `None`), as asking for a lower number of features doesn't impact the performance (it's just a matter of slicing the output matrixes of SVD).

In the following code, we first create matrixes with 10,000 points and 20, 50, 100, 250, 1,000, and 2,500 features to be processed by PCA. Then, we create matrixes with 100 features and 1, 5, 10, 25, 50, and 100 thousand observations to be processed with PCA:

```
In:import time

def check_scalability(test_pca):
    pylab.rcParams['figure.figsize'] = (10, 4)

    # FEATURES
    n_points = 10000
    n_features = [20, 50, 100, 250, 500, 1000, 2500]
    time_results = []

    for n_feature in n_features:
        X, _ = make_blobs(n_points, n_features=n_feature, \
random_state=101)

        pca = copy.deepcopy(test_pca)
        tik = time.time()
        pca.fit(X)
        time_results.append(time.time()-tik)

    plt.subplot(1, 2, 1)
    plt.plot(n_features, time_results, 'o--')
    plt.title('Feature scalability')
    plt.xlabel('Num. of features')
    plt.ylabel('Training time [s]')

    # OBSERVATIONS
    n_features = 100
    n_observations = [1000, 5000, 10000, 25000, 50000, 100000]
    time_results = []

    for n_points in n_observations:
        X, _ = make_blobs(n_points, n_features=n_features, \
random_state=101)
        pca = copy.deepcopy(test_pca)
        tik = time.time()
        pca.fit(X)
        time_results.append(time.time()-tik)

    plt.subplot(1, 2, 2)
    plt.plot(n_observations, time_results, 'o--')
```

```
    plt.title('Observations scalability')
    plt.xlabel('Num. of training observations')
    plt.ylabel('Training time [s]')



    plt.show()

check_scalability(PCA(None))
```

Out:



As you can clearly see, PCA based on SVD is not scalable: if the number of features increases linearly, the time needed to train the algorithm increases exponentially. Also, the time needed to process a matrix with a few hundred observations becomes too high and (not shown in the image) the memory consumption makes the problem unfeasible for a domestic computer (with 16 or fewer GB of RAM). It seems clear that a PCA based on SVD is not the solution for big data; fortunately, in recent years, many workarounds have been introduced. In the following sections, you'll find a short introduction for each of them.

Randomized PCA

The correct name of this technique should be *PCA based on randomized SVD*, but it has become popular with the name **Randomized PCA**. The core idea behind the randomization is the redundancy of all the principal components; in fact, if the goal of the method is dimension reduction, one should expect that only a few vectors are needed in the output (the K principal ones). By focusing on the problem of finding the best K principal vectors, the algorithm scales more. Note that in this algorithm, *K*—the number of principal components to be outputted—is a key parameter: set it too large and the

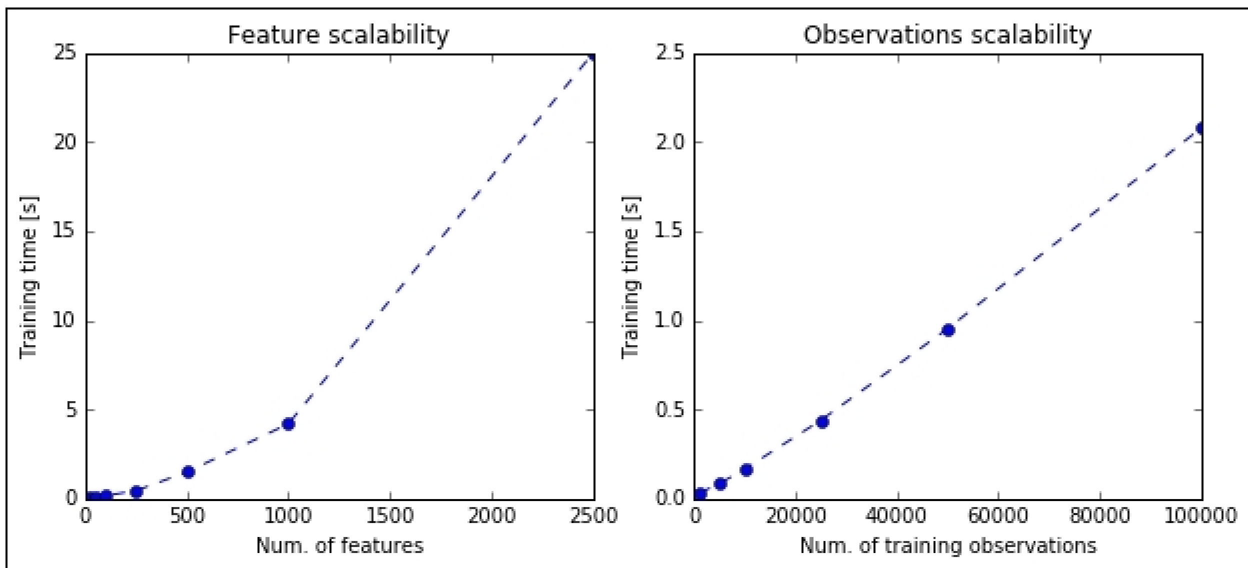
performance won't be better than PCA; set it too low and the variance explained with the resulting vectors will be too low.

As in PCA, we want to find an approximation of the matrix containing observations *X*, such that $X \approx Q \cdot Q^T \cdot X$; we also want the matrix *Q* with *K* orthonormal columns (they will be called principal components). With SVD, we can now compute the decomposition of the *small* matrix $Q^T \cdot X = U \cdot \Sigma \cdot W^T$. As we proved, it won't take a long time. As $X \approx Q \cdot Q^T \cdot X = Q \cdot U \cdot \Sigma \cdot W^T$, by taking $Q \cdot U = S$, we now have a truncated approximation of *X* based on a low rank SVD, $X \approx S \cdot \Sigma \cdot W^T$.

Mathematically, it seems perfect, but there are still two missing points: what role does the randomization have? How to get the matrix *Q*? Both questions are answered here: a Gaussian random matrix is extracted, Ω, and it's computed *Y* as $Y = X \cdot \Omega$. Then, *Y* is QR-decomposed, creating $Y = Q \cdot R$, and here is *Q*, the matrix of *K* orthonormal columns that we are looking for.

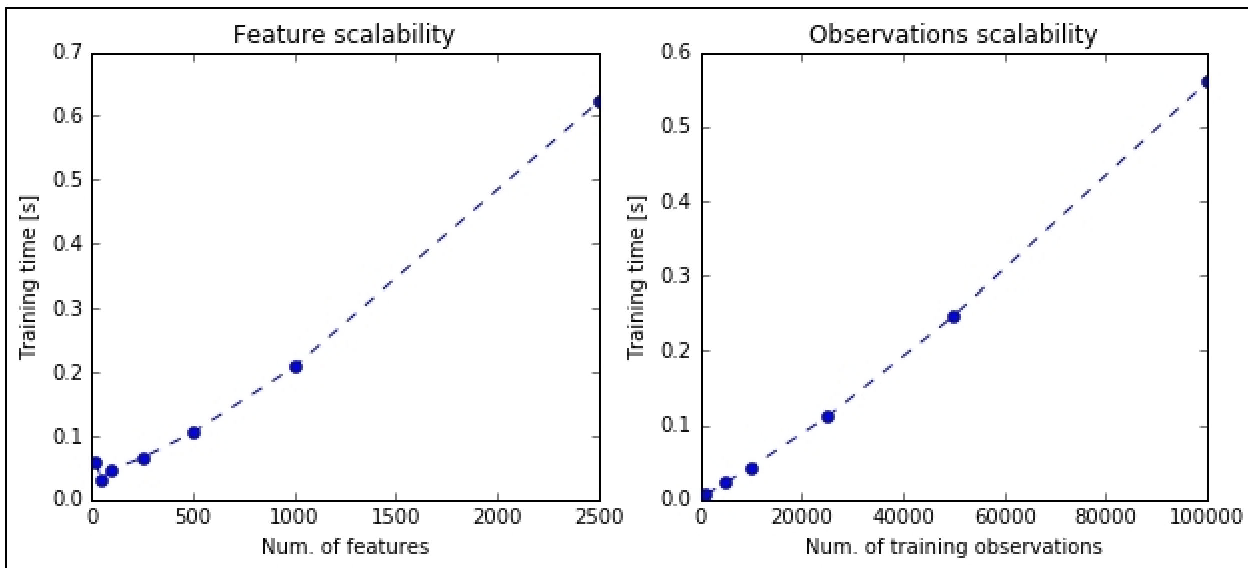
The math underneath this decomposition is pretty heavy, and fortunately everything has been implemented in Scikit-learn, so you don't need to figure out how to deal with Gaussian random variables and so on. Let's initially see how bad it performs when a full (lossless) decomposition is computed with randomized PCA:

```
In:from sklearn.decomposition import RandomizedPCA
check_scalability(RandomizedPCA(None))
```



Performances are worse than the classic PCA; in fact, this transformation works very well when a reduced set of components is asked. Let's now see the performance when *K=20*:

```
In:check_scalability(RandomizedPCA(20))
```



As expected, the computations are very quick; in less than a second, the algorithm is able to perform the most complex factorizations.

Checking the results and the algorithm, we still notice something odd: the training dataset, *X*, must all fit in-memory in order to be decomposed, even in the case of the randomized PCA. Does an online version

of PCA exist that is able to incrementally fit the principal vectors without having the whole dataset in-memory? Yes, there is—incremental PCA.

Incremental PCA

Incremental PCA, or mini-batch PCA, is the online version of Principal Component Analysis. The core of the algorithm is very simple: the batch of data is initially split into mini-batches with the same number of observations. (The only limitation is that the number of observations per mini-batch should be greater than the number of features.) Then, the first mini-batch is centered (the mean is removed) and its SVD decomposition is performed, storing the principal components. Then, when the next mini-batch enters the process, it's first centered and then stacked with the principal components extracted from the previous mini-batch (they're inserted as additional observations). Now, another SVD is performed and the principal components are overwritten with the new ones. The process goes till the last mini-batch: for each of them, first there is the centering, then the stacking, and finally the SVD. Doing so, instead of a *big* SVD, we're performing as many *small* SVD as the number of mini-batches.

As you can understand, this technique doesn't outperform randomized PCA, but its goal is to offer a solution (or the only solution) when PCA is needed on a dataset that doesn't fit in-memory. Incremental PCA doesn't run to win a speed challenge, but to limit memory consumption; the memory usage is constant throughout the training and can be tuned by setting the mini-batch size. As a rule of thumb, the memory footprint is approximately the same order of magnitude as the square of the size of the mini-batch.

As a code example, let's now check how incremental PCA is able to cope with a large dataset, which is, in our exemplification, composed of 10 million observations and 100 features. None of the previous algorithms are able to do so, unless you want to crash your computer (or witness an enormous amount of swap between memory and the storage disk). With incremental PCA, such a task turns into a piece of cake and, all things considered, the process is not all that slow (Note that we're doing a complete lossless decomposition with a stable amount of memory consumed):

```
In:from sklearn.decomposition import IncrementalPCA

X, _ = make_blobs(100000, n_features=100, random_state=101)
pca = IncrementalPCA(None, batch_size=1000)

tik = time.time()
for i in range(100):
    pca.partial_fit(X)
print "PCA on 10M points run with constant memory usage in ", \
    time.time() - tik, "seconds"

Out:PCA on 10M points run with constant memory usage in
155.642718077 seconds
```

Sparse PCA

Sparse PCA operates in a different way than the previous algorithms; instead of operating a feature reduction using an SVD applied on the covariance matrix (after being centered), it operates a feature selection-like operation of that matrix, finding the set of sparse components that best reconstruct the data. As in Lasso regularization, the amount of sparseness is controllable by a penalty (or constraint) on the coefficients.

With respect to PCA, sparse PCA doesn't guarantee that the resulting components will be orthogonal, but the result is more interpretable as principal vectors are actually a portion of the input dataset. Moreover, it's scalable in terms of the number of features: if PCA and its scalable versions are stuck when the number of features is getting larger (let's say, more than 1,000), sparse PCA is still an optimal solution in terms of speed thanks to the internal method to solve the Lasso problem, typically based on Lars or Coordinate Descent. (Remember that Lasso tries to minimize the L1 norm of the coefficients.) Moreover, it's great when the number of features is greater than the number of observations as, for example, some image datasets.

Let's now see how it works on a 25,000 observations dataset with 10,000 features. For this example, we're using the mini-batch version of the `SparsePCA` algorithm that ensures constant memory usage and is able to cope with massive datasets, eventually larger than the available memory (note that the batch version is named `SparsePCA` but doesn't support the online training):

```
In:from sklearn.decomposition import MiniBatchSparsePCA

X, _ = make_blobs(25000, n_features=10000, random_state=101)

tik = time.time()
pca = MiniBatchSparsePCA(20, method='cd', random_state=101, \
                    n_iter=1000)
pca.fit(X)
print "SparsePCA on matrix", X.shape, "done in ", time.time() - \
        tik, "seconds"

Out:
SparsePCA on matrix (25000, 10000) done in  41.7692570686 seconds
```

In about 40 seconds, `SparsePCA` is able to produce a solution using a constant amount of memory.

PCA with H2O

We can also use the PCA implementation provided by H2O. (We've already seen H2O in the previous chapter and mentioned it along the book.)

With H2O, we first need to turn on the server with the `init` method. Then, we dump the dataset on a file (precisely, a CSV file) and finally run the PCA analysis. As the last step, we shut down the server.

We're trying this implementation on some of the biggest datasets seen so far—the one with 100K observations and 100 features and the one with 10K observations and 2,500 features:

```
In: import h2o
from h2o.transforms.decomposition import H2OPCA
h2o.init(max_mem_size_GB=4)

def testH2O_pca(nrows, ncols, k=20):
    temp_file = tempfile.NamedTemporaryFile().name
    X, _ = make_blobs(nrows, n_features=ncols, random_state=101)
np.savetxt(temp_file, np.c_[X], delimiter=",")
    del X

pca = H2OPCA(k=k, transform="NONE", pca_method="Power")
    tik = time.time()
    pca.train(x=range(100), \
training_frame=h2o.import_file(temp_file))

    print "H2OPCA on matrix ", (nrows, ncols), \
" done in ", time.time() - tik, "seconds"
os.remove(temp_file)

testH2O_pca(100000, 100)
testH2O_pca(10000, 2500)
h2o.shutdown(prompt=False)

Out:[...]
H2OPCA on matrix  (100000, 100) done in  12.9560530186 seconds
[...]
H2OPCA on matrix  (10000, 2500) done in  10.1429388523 seconds
```

As you can see, in both cases, H2O indeed performs very fast and is well-comparable (if not outperforming) to Scikit-learn.

Clustering – K-means

K-means is an unsupervised algorithm that creates K disjoint clusters of points with equal variance, minimizing the distortion (also named inertia).

Given only one parameter K, representing the number of clusters to be created, the K-means algorithm creates K sets of points $S_1, S_2, \ldots, S_K$, each of them represented by its centroid: $C_1, C_2, \ldots, C_K$. The generic centroid, C_i, is simply the mean of the samples of the points associated to the cluster Si in order to minimize the intra-cluster distance. The outputs of the system are as follows:

1. The composition of the clusters $S_1, S_2, \ldots, S_K$, that is, the set of points composing the training set that are associated to the cluster number 1, 2, …, K.
2. The centroids of each cluster, $C_1, C_2, \ldots, C_K$. Centroids can be used for future associations.
3. The distortion introduced by the clustering, computed as follows:
$$D = \sum_{i=i}^{K} \sum_{x \in S_i} ||x - Ci||^2$$

This equation denotes the optimization intrinsically done in the K-means algorithm: the centroids are chosen to minimize the intra-cluster distortion, that is, the sum of Euclidean norms of the distances between each input point and the centroid of the cluster to which the point is associated to. In other words, the algorithm tries to fit the best vectorial quantization.

The training phase of the K-means algorithm is also called Lloyd's algorithm, named after Stuart Lloyd, who first proposed the algorithm. It's an iterative algorithm composed of two phases iterated over and over till convergence (the distortion reaches a minimum). It's a variant of the generalized **expectation-maximization** (**EM**) algorithm as the first step creates a function for the **expectation** (**E**) of a score, and the **maximization** (**M**) step computes the parameters that maximize the score. (Note that in this formulation, we try to achieve the opposite, that is, the minimization of the distortion.) Here's its formulation:

- The expectation step: In this step, the points in the training set are assigned to the closest centroid:
$$S_i^{(t)} = \{\ x: \|x - Ci\|^2 = \min_j \|x - Cj\|^2\ \}$$

  This step is also named assignment or vectorial quantization.
- The maximization step: The centroid of each cluster is moved to the middle of the cluster by averaging the points composing it:

$$C_i^{(t+1)} = \frac{1}{|S_i^{(t)}|} \cdot \sum_{x \in S_i^{(t)}} x \; for \; i = 1, \ldots, K$$

This step is also named update step.

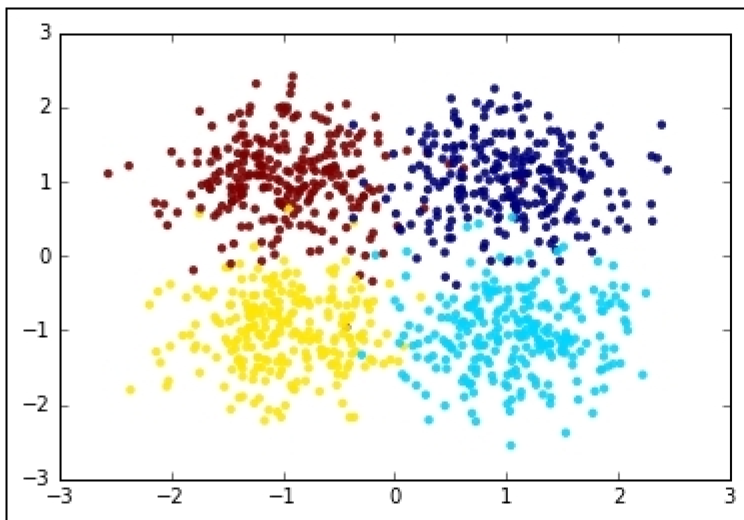
These two steps are performed till convergence (points are stable in their cluster), or till the algorithm reaches a preset number of iterations. Note that, per composition, the distortion cannot increase throughout the training phase (unlike Stochastic Gradient Descent-based methods); therefore, in this algorithm, the more iterations, the better the result.

Let's now see how it looks like on a dummy two-dimensional dataset. We first create a set of 1,000 points concentrated in four locations symmetric with respect to the origin. Each cluster, per construction, has the same variance:

```
In:import matplotlib
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
%matplotlib inline

In:from sklearn.datasets.samples_generator import make_blobs


centers = [[1, 1], [1, -1], [-1, -1], [-1, 1]]
X, y = make_blobs(n_samples=1000, centers=centers,
                  cluster_std=0.5, random_state=101)
```

Let's now plot the dataset. To make things easier, we will color the clusters with different colors:

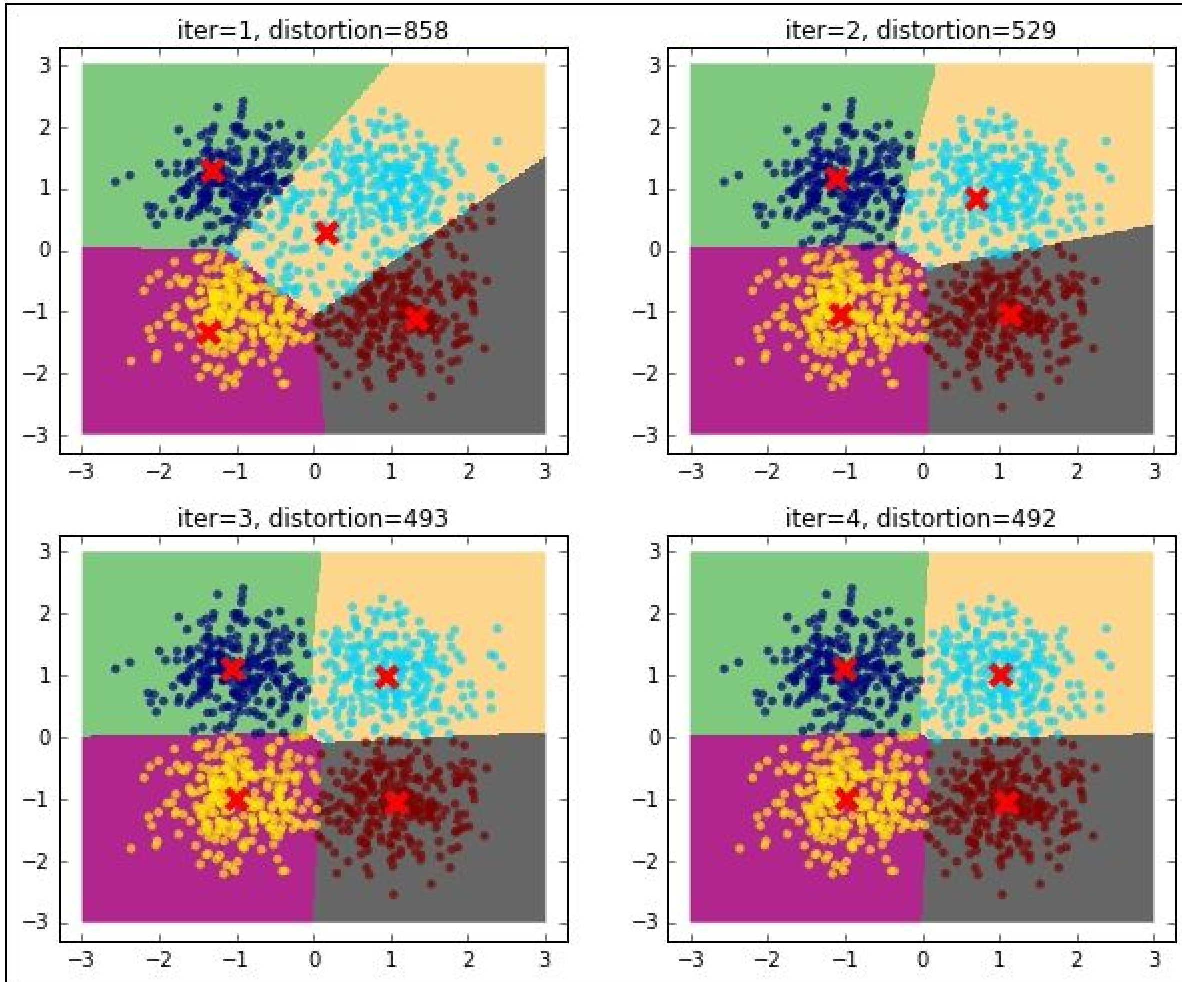
```
In:plt.scatter(X[:,0], X[:,1], c=y, edgecolors='none', alpha=0.9)
plt.show()
```



Let's now run K-means and inspect what's going on at each iteration. For this, we stop the iteration to 1, 2, 3, and 4 iterations and plot the points with their associated cluster (color-coded) as well as the centroid, distortion (in the title), and decision boundaries (also named Voronoi cells). The initial choice of the centroids is at random, that is, four training points are elected centroids in the first iteration during the expectation phase of the training:

```
In:pylab.rcParams['figure.figsize'] = (10.0, 8.0)

from sklearn.cluster import KMeans

for n_iter in range(1, 5):

    cls = KMeans(n_clusters=4, max_iter=n_iter, n_init=1,
                 init='random', random_state=101)
    cls.fit(X)


    # Plot the voronoi cells

    plt.subplot(2, 2, n_iter)
    h=0.02
    xx, yy = np.meshgrid(np.arange(-3, 3, h), np.arange(-3, 3, h))
    Z = cls.predict(np.c_[xx.ravel(), \
        yy.ravel()]).reshape(xx.shape)
    plt.imshow(Z, interpolation='nearest', cmap=plt.cm.Accent, \
               extent=(xx.min(), xx.max(), yy.min(), yy.max()), \
               aspect='auto', origin='lower')


    plt.scatter(X[:,0], X[:,1], c=cls.labels_, \
```

```
edgecolors='none', alpha=0.7)
    plt.scatter(cls.cluster_centers_[:,0], \
                cls.cluster_centers_[:,1], \
                marker='x', color='r', s=100, linewidths=4)
    plt.title("iter=%s, distortion=%s" %(n_iter, \
                int(cls.inertia_)))

plt.show()
```



As you can see, the distortion is getting lower and lower as the number of iterations increases. For this dummy dataset, it seems that using a few iterations (five iterations), we've reached the convergence.

Initialization methods

Finding the global minimum of the distortion in K-means is a NP-hard problem; moreover, exactly as with Stochastic Gradient Descent, this method is prone to converge to local minima especially if the number of dimensions is high. In order to avoid such behavior and limit the maximum number of iterations, you can use the following countermeasures:

- Run the algorithm multiple times using different initial conditions. In Scikit-learn, the `KMeans` class has the `n_init` parameter that controls how many times the K-means algorithm will be run with different centroid seeds. At the end, the model that ensures the lower distortion is selected. If multiple cores are available, this process can be run in parallel by setting the `n_jobs` parameter to the number of desired jobs to spin off. Note that the memory consumption is linearly dependent on the number of parallel jobs.
- Prefer the k-means++ initialization (the `KMeans` class is the default) to the random choice of training points. K-means++ initialization selects points that are *far* among each other; this should ensure that the centroids are able to form clusters in uniform subspaces of the space. It's also proved that this fact ensures that *it's more likely* to find the best solution.

K-means assumptions

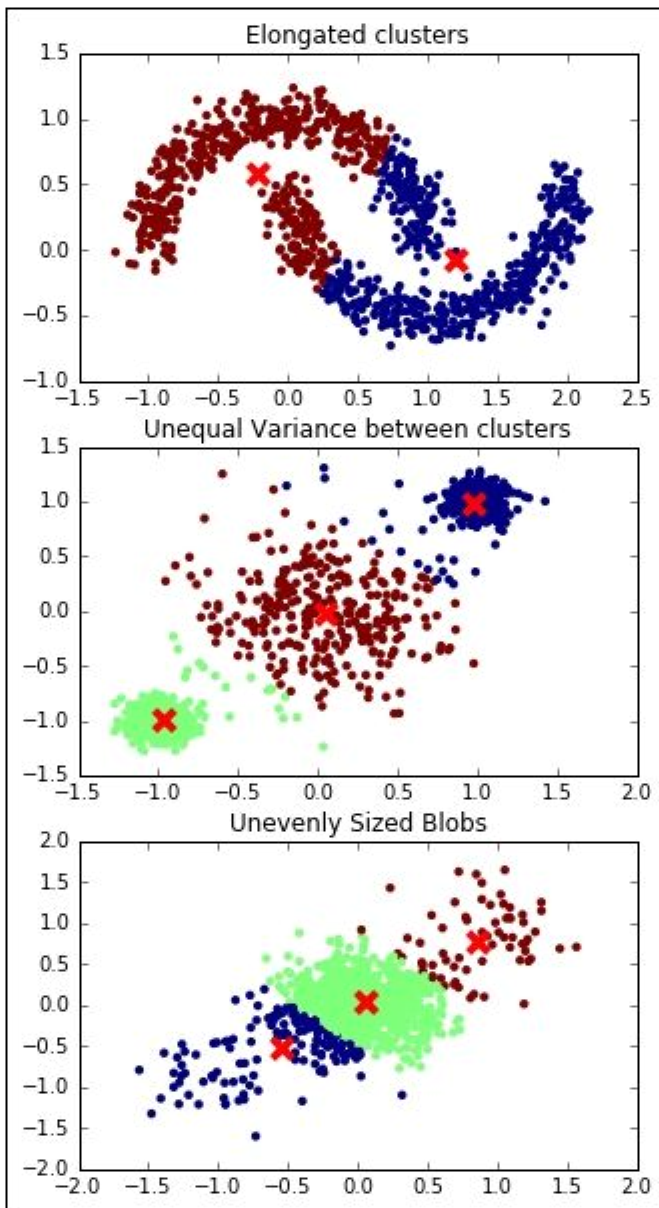
K-means relies on the assumptions that each cluster has a (hyper-) spherical shape, that is, it doesn't have an elongated shape (like an arrow), all the clusters have the same variance internally, and their size is comparable (or they are very far away).

All of these hypotheses can be guaranteed with a strong feature preprocessing step; PCA, KernelPCA, feature normalization, and sampling can be a good start.

Let's now see what happens when the assumptions behind K-means are not met:

```
In:pylab.rcParams['figure.figsize'] = (5.0, 10.0)
from sklearn.datasets import make_moons

# Oblong/elongated sets
X, _ = make_moons(n_samples=1000, noise=0.1, random_state=101)
cls = KMeans(n_clusters=2, random_state=101)
y_pred = cls.fit_predict(X)

plt.subplot(3, 1, 1)
plt.scatter(X[:, 0], X[:, 1], c=y_pred, edgecolors='none')
plt.scatter(cls.cluster_centers_[:,0], cls.cluster_centers_[:,1],
                marker='x', color='r', s=100, linewidths=4)
plt.title("Elongated clusters")

# Different variance between clusters
centers = [[-1, -1], [0, 0], [1, 1]]
X, _ = make_blobs(n_samples=1000, cluster_std=[0.1, 0.4, 0.1],
                  centers=centers, random_state=101)
cls = KMeans(n_clusters=3, random_state=101)
y_pred = cls.fit_predict(X)

plt.subplot(3, 1, 2)
plt.scatter(X[:, 0], X[:, 1], c=y_pred, edgecolors='none')
plt.scatter(cls.cluster_centers_[:,0], cls.cluster_centers_[:,1],
                marker='x', color='r', s=100, linewidths=4)
```

```
plt.title("Unequal Variance between clusters")

# Unevenly sized blobs
centers = [[-1, -1], [1, 1]]
centers.extend([[0,0]]*20)
X, _ = make_blobs(n_samples=1000, centers=centers,
                  cluster_std=0.28, random_state=101)
cls = KMeans(n_clusters=3, random_state=101)
y_pred = cls.fit_predict(X)

plt.subplot(3, 1, 3)
plt.scatter(X[:, 0], X[:, 1], c=y_pred, edgecolors='none')
plt.scatter(cls.cluster_centers_[:,0], cls.cluster_centers_[:,1],
                marker='x', color='r', s=100, linewidths=4)
plt.title("Unevenly Sized Blobs")

plt.show()
```

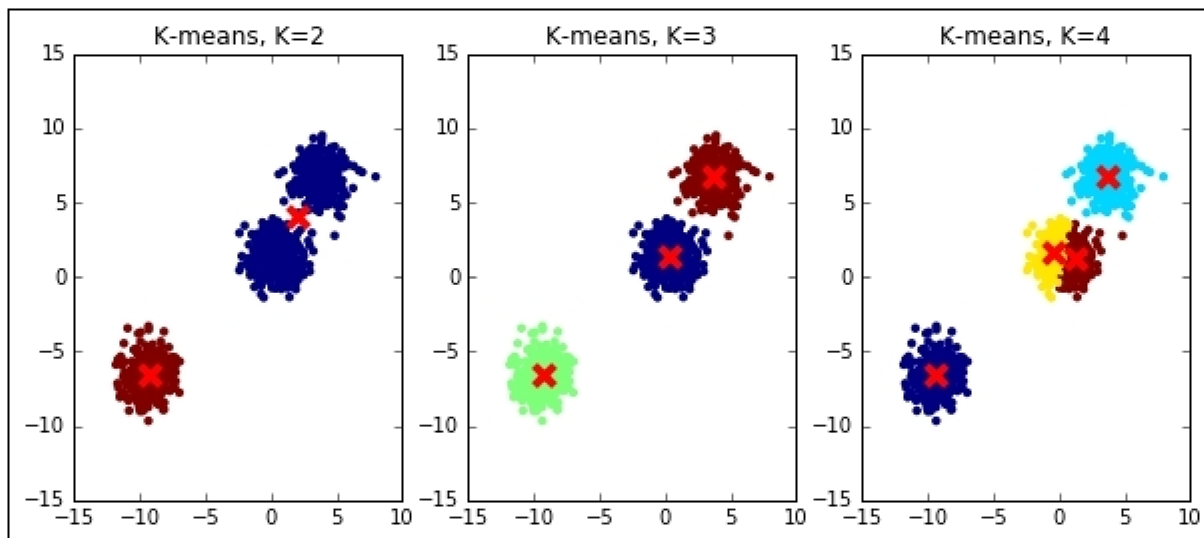


In all the preceding examples, the clustering operation is not perfect, outputting a wrong and unstable result.

So far, we've assumed to know exactly which is the exact K, the number of clusters that we're expecting to use in the clustering operation. Actually, in real-world problems, this is not always true. We often use an unsupervised learning method to discover the underlying structure of the data, including the number of clusters composing the dataset. Let's see what happens when we try to run K-means with a wrong K on a simple dummy dataset; we will try both a lower K and a higher one:

```
In:pylab.rcParams['figure.figsize'] = (10.0, 4.0)
X, _ = make_blobs(n_samples=1000, centers=3, random_state=101)
```

```
for K in [2, 3, 4]:
    cls = KMeans(n_clusters=K, random_state=101)
    y_pred = cls.fit_predict(X)

    plt.subplot(1, 3, K-1)
    plt.title("K-means, K=%s" % K)
    plt.scatter(X[:, 0], X[:, 1], c=y_pred, edgecolors='none')
    plt.scatter(cls.cluster_centers_[:,0],
cls.cluster_centers_[:,1],
                marker='x', color='r', s=100, linewidths=4)

plt.show()
```



As you can see, the results are massively wrong in case the right K is not guessed, even for this simple dummy dataset. In the next section, we will explain some tricks to best select the K.

Selection of the best K

There are several methods to detect the best K if the assumptions behind K-means are met. Some of them are based on cross-validation and metrics on the output; they can be used on all clustering methods, but only when a ground truth is available (they're named supervised metrics). Some others are based on intrinsic parameters of the clustering algorithm and can be used independently by the presence or absence of the ground truth (also named unsupervised metrics). Unfortunately, none of them ensures 100% accuracy to find the correct result.

Supervised metrics require a ground truth (containing the true associations in sets) and they're usually combined with a gridsearch analysis to understand the best K. Some of these metrics are derived from equivalent classification ones, but they allow having a different number of unordered sets as predicted labels. The first one that we're going to see is named **homogeneity**; as you can expect, it gives a measure of how many of the predicted clusters contain just points of one class. It's a measure based on entropy,

and it's the cluster equivalent of the precision in classification. It's a measure bound between 0 (worst) and 1 (best); its mathematical formulation is as follows:

$$h = 1 - \frac{H(C|K)}{H(C)}$$

Here, *H(C|K)* is the conditional entropy of the class distribution given the proposed clustering assignment, and *H(C)* is the entropy of the classes. *H(C|K)* is maximal and equals *H(C)* when the clustering provides no new information; it is zero when each cluster contains only a member of a single class.

Connected to it, as in precision and recall for classification, there is the **completeness** score: it gives a measure about how much all members of a class are assigned to the same cluster. Even this one is bound between 0 (worst) and 1 (best), and its mathematical formulation is deeply based on entropy:

$$c = 1 - \frac{H(K|C)}{H(K)}$$

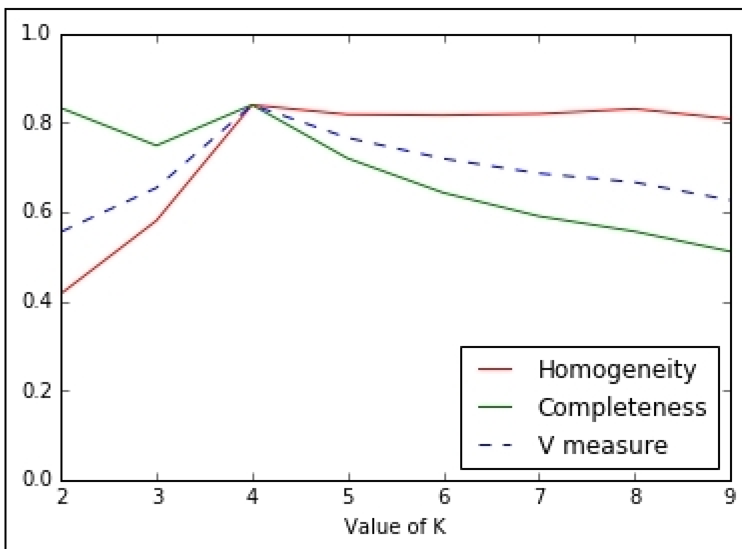
Here, *H(K|C)* is the conditional entropy of the proposed cluster distribution given the class, and *H(K)* is the entropy of the clusters.

Finally, equivalent to the f1 score for the classification task, the V-measure is the harmonic mean of homogeneity and completeness:

$$v = 2 \cdot \frac{h \cdot c}{h + c}$$

Let's get back to the first dataset (four symmetric noisy clusters), and try to see how these scores operate and whether they are able to highlight the best K to use:

```
In:pylab.rcParams['figure.figsize'] = (6.0, 4.0)
from sklearn.metrics import homogeneity_completeness_v_measure


centers = [[1, 1], [1, -1], [-1, -1], [-1, 1]]
X, y = make_blobs(n_samples=1000, centers=centers,
                  cluster_std=0.5, random_state=101)

Ks = range(2, 10)
HCVs = []
for K in Ks:
    y_pred = KMeans(n_clusters=K, random_state=101).fit_predict(X)
    HCVs.append(homogeneity_completeness_v_measure(y, y_pred))

plt.plot(Ks, [el[0] for el in HCVs], 'r', label='Homogeneity')
plt.plot(Ks, [el[1] for el in HCVs], 'g', label='Completeness')
plt.plot(Ks, [el[2] for el in HCVs], 'b', label='V measure')
plt.ylim([0, 1])
plt.legend(loc=4)
plt.show()
```

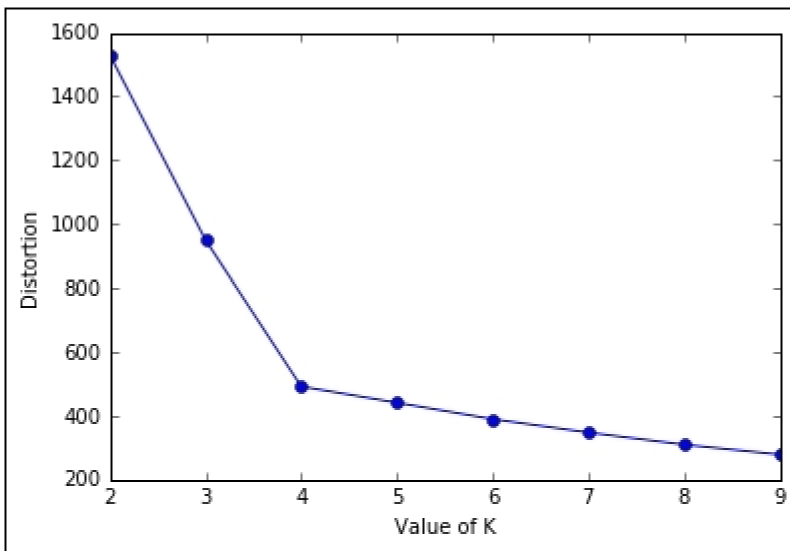


In the plot, initially (*K<4*) completeness is high, but homogeneity is low; for *K>4*, it is the opposite: homogeneity is high, but completeness is low. In both cases, the V-measure is low. For *K=4*, instead, all the measure reaches their maximum, indicating that's the best value for *K*, the number of clusters.

Beyond these metrics that are supervised, there are others named unsupervised that don't require a ground truth, but are just based on the learner itself.

The first that we're going to see in this section is the **Elbow method**, applied to the distortion. It's very easy and doesn't require any math: you just need to plot the distortion of many K-means models with different Ks, then select the one in which increasing K doesn't introduce *much lower* distortion in the solution. In Python, this is very simple to achieve:

```
In:Ks = range(2, 10)
Ds = []
for K in Ks:
    cls = KMeans(n_clusters=K, random_state=101)
    cls.fit(X)
    Ds.append(cls.inertia_)

plt.plot(Ks, Ds, 'o-')
plt.xlabel("Value of K")
plt.ylabel("Distortion")
plt.show()
```

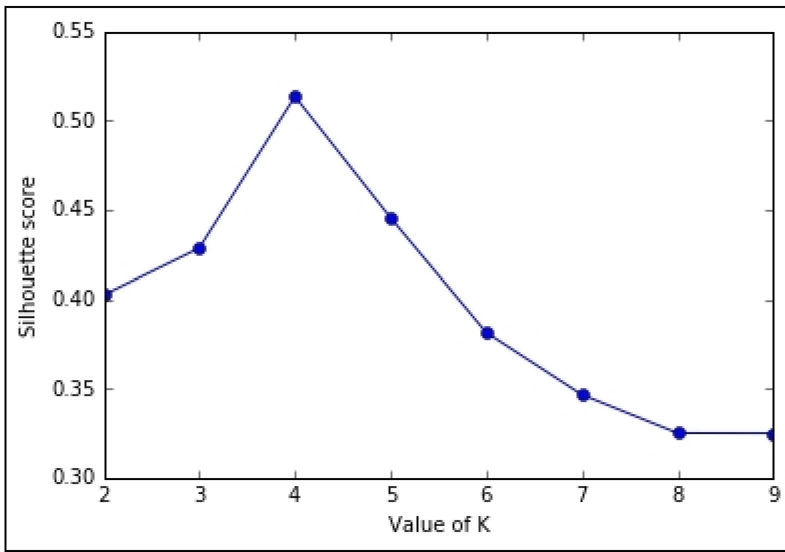


As you can expect, the distortion drops till *K=4*, then it decreases slowly. Here, the best-obtained K is 4.

Another unsupervised metric that we're going to see is the **Silhouette**. It's more complex, but also more powerful than the previous heuristics. At a very high level, it measures how close (similar) an observation is to the assigned cluster and how loosely (dissimilarly) it is matched to the data of nearby clusters. A Silhouette score of 1 indicates that all the data is in the best cluster, and -1 indicates a completely wrong cluster result. To obtain such a measure using Python code is very easy, thanks to the Scikit-learn implementation:

```
In:from sklearn.metrics import silhouette_score

Ks = range(2, 10)
```

```
Ds = []
for K in Ks:
    cls = KMeans(n_clusters=K, random_state=101)
    Ds.append(silhouette_score(X, cls.fit_predict(X)))

plt.plot(Ks, Ds, 'o-')
plt.xlabel("Value of K")
plt.ylabel("Silhouette score")
plt.show()
```



Even in this case, we've arrived at the same conclusion: the best value for K is 4 as the silhouette score is much lower with a lower and higher K.

Scaling K-means – mini-batch

Let's now test the scalability of K-means. From the website of UCI, we've selected an appropriate dataset for this task: the US Census 1990 Data. This dataset contains almost 2.5 million observations and 68 categorical (but already number-encoded) attributes. There is no missing data and the file is in the CSV format. Each observation contains the ID of the individual (to be removed before the clustering) and other information about gender, income, marital status, work, and so on.

Note

Further information about the dataset can be found at http://archive.ics.uci.edu/ml/datasets/US+Census+Data+%281990%29 or in the paper published in The Journal of Machine Learning Research by Meek, Thiesson, and Heckerman (2001) entitled *The Learning Curve Method Applied to Clustering*.

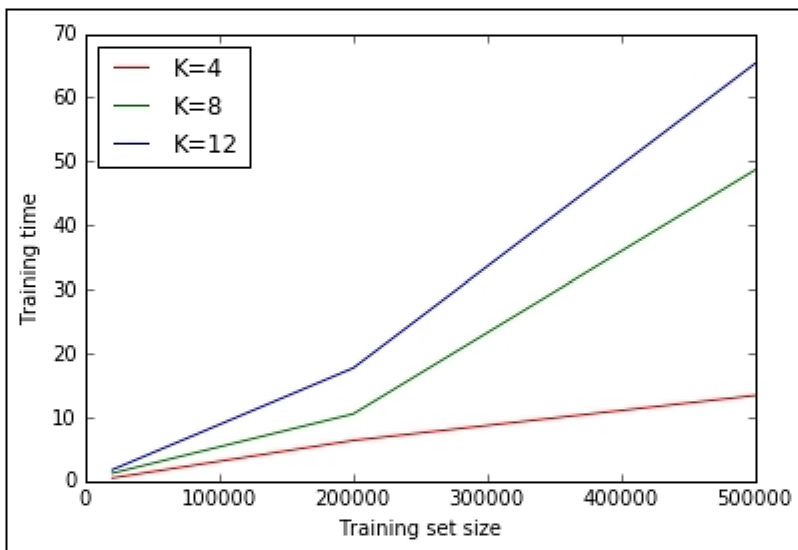
As the first thing to be done, you have to download the file containing the dataset and store it in a temporary directory. Note that it's 345MB in size, therefore its download might require a long time on slow connections:

```
In:import urllib
import os.path

url = "http://archive.ics.uci.edu/ml/machine-learning-databases/
census1990-mld/USCensus1990.data.txt"
census_csv_file = "/tmp/USCensus1990.data.txt"

import os.path
if not os.path.exists(census_csv_file):
    testfile = urllib.URLopener()
    testfile.retrieve(url, census_csv_file)
```

Now, let's run some tests clocking the times needed to train a K-means learner with K equal to 4, 8, and 12, and with a dataset containing 20K, 200K, and 0.5M observations. As we don't want to saturate the memory of the machine, consequently we will just read the first 500K lines and drop the column containing the identifier of the user. Finally, let's plot the training times for a complete performance evaluation:

```
In:piece_of_dataset = pd.read_csv(census_csv_file,
iterator=True).get_chunk(500000).drop('caseid', axis=1).as_matrix()

time_results = {4: [], 8:[], 12:[]}
dataset_sizes = [20000, 200000, 500000]


for dataset_size in dataset_sizes:
    print "Dataset size:", dataset_size
    X = piece_of_dataset[:dataset_size,:]

    for K in [4, 8, 12]:
        print "K:", K
        cls = KMeans(K, random_state=101)
        timeit = %timeit -o -n1 -r1 cls.fit(X)

        time_results[K].append(timeit.best)

plt.plot(dataset_sizes, time_results[4], 'r', label='K=4')
plt.plot(dataset_sizes, time_results[8], 'g', label='K=8')
plt.plot(dataset_sizes, time_results[12], 'b', label='K=12')

plt.xlabel("Training set size")
plt.ylabel("Training time")
plt.legend(loc=0)
```

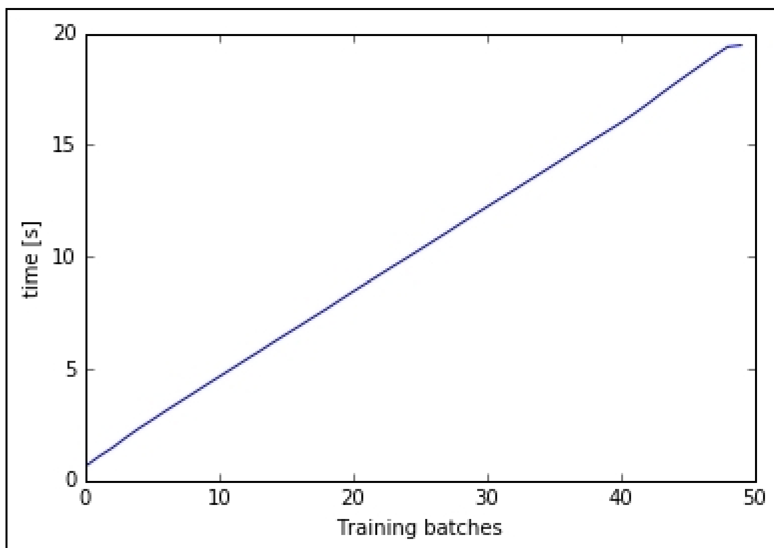
```
plt.show()

Out:Dataset size: 20000
K: 4
1 loops, best of 1: 478 ms per loop
K: 8
1 loops, best of 1: 1.22 s per loop
K: 12
1 loops, best of 1: 1.76 s per loop
Dataset size: 200000
K: 4
1 loops, best of 1: 6.35 s per loop
K: 8
1 loops, best of 1: 10.5 s per loop
K: 12
1 loops, best of 1: 17.7 s per loop
Dataset size: 500000
K: 4
1 loops, best of 1: 13.4 s per loop
K: 8
1 loops, best of 1: 48.6 s per loop
K: 12
1 loops, best of 1: 1min 5s per loop
```



It seems clear that, given the plot and actual timings, the training time increases linearly with K and the training set size, but for large Ks and training sizes, such a relation becomes nonlinear. Doing an exhaustive search with the whole training set for many Ks does not seem scalable.

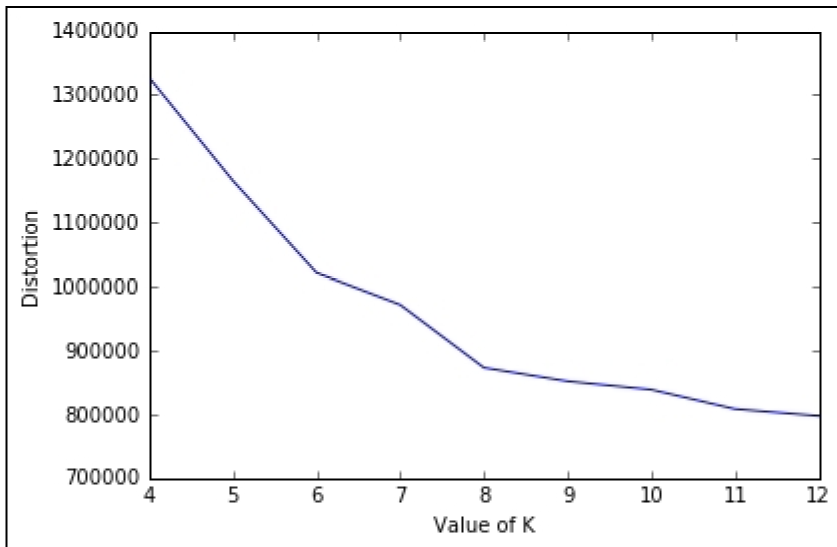
Fortunately, there's an online version of K-means based on mini-batches, already implemented in Scikit-learn and named `MiniBatchKMeans`. Let's try it on the slowest case of the previous cell, that is, with *K=12*. With the classic K-means, the training on 500,000 samples (circa 20% of the full dataset) took more than a minute; let's see the performance of the online mini-batch version, setting the batch size to 1,000 and importing chunks of 50,000 observations from the dataset. As an output, we plot the training time versus the number of chunks already passed thought the training phase:

```
In:from sklearn.cluster import MiniBatchKMeans
import time

cls = MiniBatchKMeans(12, batch_size=1000, random_state=101)
ts = []

tik = time.time()
for chunk in pd.read_csv(census_csv_file, chunksize=50000):
    cls.partial_fit(chunk.drop('caseid', axis=1))
    ts.append(time.time()-tik)

plt.plot(range(len(ts)), ts)
plt.xlabel('Training batches')
plt.ylabel('time [s]')

plt.show()
```



Training time is linear for each chunk, performing the clustering on the full 2.5 million observations dataset in nearly 20 seconds. With this implementation, we can run a full search to select the best K using the elbow method on the distortion. Let's do a gridsearch, with K spanning from 4 to 12, and plot the distortion:

```
In:Ks = list(range(4, 13))
ds = []

for K in Ks:
    cls = MiniBatchKMeans(K, batch_size=1000, random_state=101)

    for chunk in pd.read_csv(census_csv_file, chunksize=50000):
        cls.partial_fit(chunk.drop('caseid', axis=1))
    ds.append(cls.inertia_)

plt.plot(Ks, ds)
plt.xlabel('Value of K')
plt.ylabel('Distortion')

plt.show()

Out:
```



From the plot, the elbow seems in correspondence of K=8. Beyond the value, we would like to point out that in less than a couple of minutes, we've been able to perform this massive operation on a large dataset, thanks to the batch implementation; therefore remember never to use the plain vanilla K-means if the dataset is getting big.

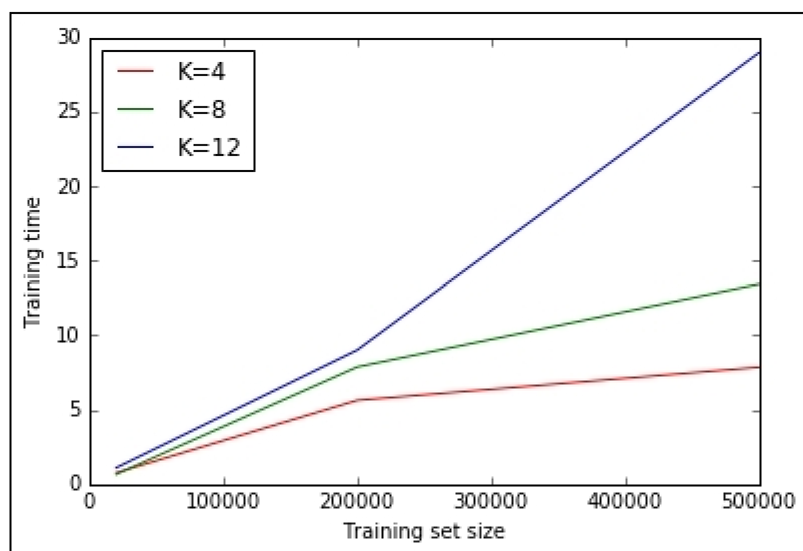
K-means with H2O

Here, we're comparing the K-means implementation of H2O with Scikit-learn. More specifically, we will run the mini-batch experiment using `H2OKMeansEstimator`, the object for K-means available in H2O. The setup is similar to the one shown in the *PCA with H2O* section, and the experiment is the same as seen in the preceding section:

```
In:import h2o
from h2o.estimators.kmeans import H2OKMeansEstimator
h2o.init(max_mem_size_GB=4)

def testH2O_kmeans(X, k):

    temp_file = tempfile.NamedTemporaryFile().name
    np.savetxt(temp_file, np.c_[X], delimiter=",")

    cls = H2OKMeansEstimator(k=k, standardize=True)
    blobdata = h2o.import_file(temp_file)

    tik = time.time()
    cls.train(x=range(blobdata.ncol), training_frame=blobdata)
    fit_time = time.time() - tik

    os.remove(temp_file)

    return fit_time

piece_of_dataset = pd.read_csv(census_csv_file,
iterator=True).get_chunk(500000).drop('caseid', axis=1).as_matrix()
time_results = {4: [], 8:[], 12:[]}
dataset_sizes = [20000, 200000, 500000]

for dataset_size in dataset_sizes:
    print "Dataset size:", dataset_size
    X = piece_of_dataset[:dataset_size,:]

    for K in [4, 8, 12]:
        print "K:", K
        fit_time = testH2O_kmeans(X, K)
        time_results[K].append(fit_time)

plt.plot(dataset_sizes, time_results[4], 'r', label='K=4')
plt.plot(dataset_sizes, time_results[8], 'g', label='K=8')
plt.plot(dataset_sizes, time_results[12], 'b', label='K=12')

plt.xlabel("Training set size")
```

```
plt.ylabel("Training time")
plt.legend(loc=0)
plt.show()

testH2O_kmeans(100000, 100)

h2o.shutdown(prompt=False)
```

Out:



Thanks to the H2O architecture, its implementation of K-means is very fast and scalable and able to perform the clustering of the 500K point datasets in less than 30 seconds for all the selected Ks.

LDA

LDA stands for **Latent Dirichlet Allocation**, and it's one of the widely-used techniques to analyze collections of textual documents.

Note

LDA is an acronym also used by another technique, Linear Discriminant Analysis, which is a supervised method for classification. Pay attention to how LDA is used as there's no connection between these two algorithms.

A full mathematical explanation of LDA would require the knowledge of probabilistic modeling, which is beyond the scope of this practical book. Here, instead, we will give you the most important intuitions behind the model and how to practically apply this model on a massive dataset.

First at all, LDA is used in a branch of data science named text mining, where the focus is on building learners to understand the natural language, for instance, based on textual examples. Specifically, LDA belongs to the category of topic-modeling algorithms as it tries to model the topics included in a document. Ideally, LDA is able to understand whether a document is about finance, politics, or religion, for example. However, differently from a classifier, it is also able to quantify the presence of topics in a document. For example, let's think about a Harry Potter novel, by Rowling. A classifier would be able to assess its category (fantasy novel); LDA, instead, is able to understand how much comedy, drama, mystery, romance, and adventure is in there. Moreover, LDA doesn't require any label; it's an unsupervised method and internally builds the output categories or topic and its composition (that is, given by the set of words composing a topic).

During the processing, LDA builds both a topics-per-document model and a words-per-topic model, modeled as Dirichlet distributions. Although the complexity is high, the processing time needed to output stable results is not all that long, thanks to an iterative Monte Carlo-like core function.

The LDA model is easy to understand: each document is modeled as a distribution of topics, and each topic is modeled as a distribution of words. Distributions assume to have a Dirichlet prior (with different parameters as the number of words per topic are usually different than the number of topics per document). Thanks to Gibbs sampling, distributions shouldn't be directly sampled, but an accurate approximate of it is obtained iteratively. Similar results can be obtained using the variational Bayes technique, where the approximation is generated with an Expectation-Maximization approach.

The resulting LDA model is generative (as happens with Hidden Markov Models, Naïve Bayes, and Restricted Boltzmann Machines), therefore each variable can be simulated and observed.

Let's now see how it works on a real-world dataset—the 20 Newsgroup dataset. It's composed of a collection of e-mails exchanged in 20 newsgroups. Let's initially load it, removing the e-mail headers, footers, and quotes from replied e-mails:

```
In:from sklearn.datasets import fetch_20newsgroups
documents = fetch_20newsgroups(remove=('headers', 'footers', \
            'quotes'), random_state=101).data
```

Check the size of the dataset (that is, how many documents), and print one of them to see what one document is actually composed of:

```
In:len(documents)
```

```
Out:11314
```

```
In:document_num = 9960
print documents[document_num]
```

```
Out:Help!!!

I have an ADB graphicsd tablet which I want to connect to my
Quadra 950. Unfortunately, the 950 has only one ADB port and
it seems I would have to give up my mouse.

Please, can someone help me? I want to use the tablet as well as
the mouse (and the keyboard of course!!!).

Thanks in advance.
```

As in the example, one guy is looking for help for his video socket on his tablet.

Now, we import the Python packages needed to run LDA. The Gensim package is one of the best ones and, as you'll see at the end of the section, it is also very scalable:

```
In:import gensim
from gensim.utils import simple_preprocess
from gensim.parsing.preprocessing import STOPWORDS
from nltk.stem import WordNetLemmatizer, SnowballStemmer

np.random.seed(101)
```

As the first step, we should clean the text. A few steps are necessary, which is typical of any NLP text processing:

1. Tokenization is where the text is split into sentences and sentences are split into words. Finally, words are lowercased. At this point, punctuation (and accents) is removed.
2. Words composed of fewer than three characters are removed. (This step removes most of the acronyms, emoticons, and conjunctions.)
3. Words appearing in the list of English *stopwords* are removed. Words in this list are very common and have no predictive power (such as the, an, so, then, have, and so on).
4. Tokens are then lemmatized; words in third-person are changed to first-person, and verbs in past and future tenses are changed into present (for example, goes, went, and gone all become go).
5. Finally, stemming removes the inflection, reducing the word to its root (for example, shoes becomes shoe).

In the following piece of code, we will do exactly this: try to clean the text as much as possible and list the words composing each of them. At the end of the cell, we see how this operation changes the document seen previously:

```
In:lm = WordNetLemmatizer()
stemmer = SnowballStemmer("english")

def lem_stem(text):
    return stemmer.stem(lm.lemmatize(text, pos='v'))

def tokenize_lemmatize(text):
    return [lem_stem(token)
            for token in gensim.utils.simple_preprocess(text)
            if token not in gensim.parsing.preprocessing.STOPWORDS
and len(token) > 3]

print tokenize_lemmatize(documents[document_num])

Out:[u'help', u'graphicsd', u'tablet', u'want', u'connect',
u'quadra', u'unfortun', u'port', u'mous', u'help', u'want',
u'tablet', u'mous', u'keyboard', u'cours', u'thank', u'advanc']
```

Now, as the next step, let's operate the cleaning steps on all the documents. After this, we have to build a dictionary containing how many times a word appears in the training set. Thanks to the Gensim package, this operation is straightforward:

```
In:processed_docs = [tokenize(doc) for doc in documents]
word_count_dict = gensim.corpora.Dictionary(processed_docs)
```

Now, as we want to build a generic and fast solution, let's remove all the very rare and very common words. For example, we can filter out all the words appearing less than 20 times (in total) and in no more than 20% of the documents:

```
In:word_count_dict.filter_extremes(no_below=20, no_above=0.2)
```

As the next step, with such a reduced set of words, we now build the bag-of-words model for each document; that is, for each document, we create a dictionary reporting how many words and how many times the words appear:

```
In:bag_of_words_corpus = [word_count_dict.doc2bow(pdoc) \
for pdoc in processed_docs]
```

As an example, let's have a peek at the bag-of-words model of the preceding document:

```
In:bow_doc1 = bag_of_words_corpus[document_num]

for i in range(len(bow_doc1)):
    print "Word {} (\"{}\") appears {} time[s]" \
```

```
.format(bow_doc1[i][0], \
word_count_dict[bow_doc1[i][0]], bow_doc1[i][1])

Out:Word 178 ("want") appears 2 time[s]
Word 250 ("keyboard") appears 1 time[s]
Word 833 ("unfortun") appears 1 time[s]
Word 1037 ("port") appears 1 time[s]
Word 1142 ("help") appears 2 time[s]
Word 1543 ("quadra") appears 1 time[s]
Word 2006 ("advanc") appears 1 time[s]
Word 2124 ("cours") appears 1 time[s]
Word 2391 ("thank") appears 1 time[s]
Word 2898 ("mous") appears 2 time[s]
Word 3313 ("connect") appears 1 time[s]
```

Now, we have arrived at the core part of the algorithm: running LDA. As for our decision, let's ask for 12 topics (there are 20 different newsletters, but some are similar):

```
In:lda_model = gensim.models.LdaMulticore(bag_of_words_corpus,
num_topics=10, id2word=word_count_dict, passes=50)
```

Note

If you get an error with such a code, try to mono-process the version with the `gensim.models.LdaModel` class instead of `gensim.models.LdaMulticore`.

Let's now print the topic composition, that is, the words appearing in each topic and their relative weight:

```
In:for idx, topic in lda_model.print_topics(-1):
    print "Topic:{} Word composition:{}".format(idx, topic)
    print

Out:
Topic:0 Word composition:0.015*imag + 0.014*version + 0.013*avail +
0.013*includ + 0.013*softwar + 0.012*file + 0.011*graphic +
0.010*program + 0.010*data + 0.009*format

Topic:1 Word composition:0.040*window + 0.030*file + 0.018*program +
0.014*problem + 0.011*widget + 0.011*applic + 0.010*server +
0.010*entri + 0.009*display + 0.009*error

Topic:2 Word composition:0.011*peopl + 0.010*mean + 0.010*question +
0.009*believ + 0.009*exist + 0.008*encrypt + 0.008*point +
0.008*reason + 0.008*post + 0.007*thing

Topic:3 Word composition:0.010*caus + 0.009*good + 0.009*test +
0.009*bike + 0.008*problem + 0.008*effect + 0.008*differ +
```

```
0.008*engin + 0.007*time + 0.006*high

Topic:4 Word composition:0.018*state + 0.017*govern + 0.015*right +
0.010*weapon + 0.010*crime + 0.009*peopl + 0.009*protect +
0.008*legal + 0.008*control + 0.008*drug

Topic:5 Word composition:0.017*christian + 0.016*armenian +
0.013*jesus + 0.012*peopl + 0.008*say + 0.008*church + 0.007*bibl +
0.007*come + 0.006*live + 0.006*book

Topic:6 Word composition:0.018*go + 0.015*time + 0.013*say +
0.012*peopl + 0.012*come + 0.012*thing + 0.011*want + 0.010*good +
0.009*look + 0.009*tell

Topic:7 Word composition:0.012*presid + 0.009*state + 0.008*peopl +
0.008*work + 0.008*govern + 0.007*year + 0.007*israel + 0.007*say +
0.006*american + 0.006*isra

Topic:8 Word composition:0.022*thank + 0.020*card + 0.015*work +
0.013*need + 0.013*price + 0.012*driver + 0.010*sell + 0.010*help +
0.010*mail + 0.010*look

Topic:9 Word composition:0.019*space + 0.011*inform + 0.011*univers
+ 0.010*mail + 0.009*launch + 0.008*list + 0.008*post + 0.008*anonym
+ 0.008*research + 0.008*send

Topic:10 Word composition:0.044*game + 0.031*team + 0.027*play +
0.022*year + 0.020*player + 0.016*season + 0.015*hockey +
0.014*leagu + 0.011*score + 0.010*goal

Topic:11 Word composition:0.075*drive + 0.030*disk + 0.028*control +
0.028*scsi + 0.020*power + 0.020*hard + 0.018*wire + 0.015*cabl +
0.013*instal + 0.012*connect
```

Unfortunately, LDA doesn't provide a name for each topic; we should do it manuall yourselves, based on our interpretation of the results of the algorithm. After having carefully examined the composition, we can name the discovered topics as follows:

Topic	**Name**
0	Software
1	Applications

Topic	Name
2	Reasoning
3	Transports
4	Government
5	Religion
6	People actions
7	Middle-East
8	PC Devices
9	Space
10	Games
11	Drives

Let's now try to understand what topics are represented in the preceding document and their weights:

```
In:
for index, score in sorted( \
lda_model[bag_of_words_corpus[document_num]], \
key=lambda tup: -1*tup[1]):
    print "Score: {}\t Topic: {}".format(score,
lda_model.print_topic(index, 10))
Out:Score: 0.938887758964     Topic: 0.022*thank + 0.020*card +
0.015*work + 0.013*need + 0.013*price + 0.012*driver + 0.010*sell +
0.010*help + 0.010*mail + 0.010*look
```

The highest score is associated with the topic *PC Devices*. Based on our previous knowledge of the collections of documents, it seems that the topic extraction has performed quite well.

Now, let's evaluate the model as a whole. The perplexity (or its logarithm) provides us with a metric to understand how well LDA has performed on the training dataset:

```
In:print "Log perplexity of the model is",
lda_model.log_perplexity(bag_of_words_corpus)

Out:Log perplexity of the model is -7.2985188569
```

In this case, the perplexity is 2-7.298, and it's connected to the (log) likelihood that the LDA model is able to generate the documents in the test set, given the distribution of topics for those documents. The lower the perplexity, the better the model, because it basically means that the model can regenerate the text quite well.

Now, let's try to use the model on an unseen document. For simplicity, the document contains only the sentences, *Golf or tennis? Which is the best sport to play?*:

```
In:unseen_document = "Golf or tennis? Which is the best sport to
play?"

bow_vector = word_count_dict.doc2bow(\
tokenize_lemmatize(unseen_document))
for index, score in sorted(lda_model[bow_vector], \
key=lambda tup: -1*tup[1]):
    print "Score: {}\t Topic: {}".format(score, \
                lda_model.print_topic(index, 5))

Out:Score: 0.610691655136     Topic: 0.044*game + 0.031*team +
0.027*play + 0.022*year + 0.020*player

Score: 0.222640440339     Topic: 0.018*state + 0.017*govern +
0.015*right + 0.010*weapon + 0.010*crime
```

As expected, the topic with a higher score is the one about "Games", followed by others with a relatively smaller score.

How does LDA scale with the size of the corpus? Fortunately, very well; the algorithm is iterative and allows online learning, similar to the mini-batch one. The key for the online process is the `.update()` method offered by `LdaModel` (or `LdaMulticore`).

We will do this test on a subset of the original corpus composed of the first 1,000 documents, and we will update our LDA model with batches of 50, 100, 200, and 500 documents. For each mini-batch updating the model, we will record the time and plot them on a graph:

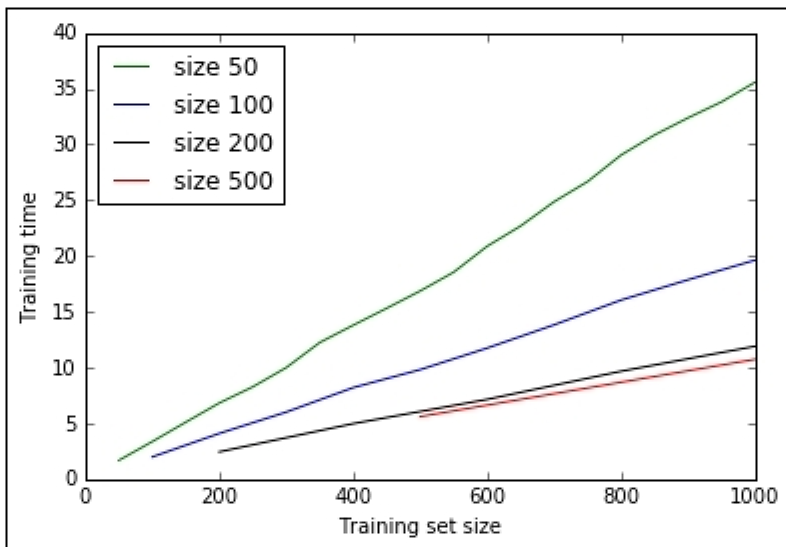
```
In:small_corpus = bag_of_words_corpus[:1000]
batch_times = {}

for batch_size in [50, 100, 200, 500]:
    print "batch_size =", batch_size
    tik0 = time.time()
    lda_model = gensim.models.LdaModel(num_topics=12, \
```

```
                id2word=word_count_dict)
    batch_times[batch_size] = []

    for i in range(0, len(small_corpus), batch_size):
        lda_model.update(small_corpus[i:i+batch_size], \
update_every=25, \
passes=1+500/batch_size)
        batch_times[batch_size].append(time.time() - tik0)

Out:batch_size = 50
batch_size = 100
batch_size = 200
batch_size = 500
```

Note that we've set the `update_every` and `passes` parameters in the model update. This is necessary to make the model converge at each iteration and not return a non-converging model. Note that 500 has been chosen heuristically; if you set it lower, you'll have many warnings from Gensim about the non-convergence of the model.

Let's now plot the results:

```
In:plt.plot(range(50, 1001, 50), batch_times[50], 'g', \
label='size 50')
plt.plot(range(100, 1001, 100), batch_times[100], 'b', \
label='size 100')
plt.plot(range(200, 1001, 200), batch_times[200], 'k', \
label='size 200')
plt.plot(range(500, 1001, 500), batch_times[500], 'r', \
label='size 500')

plt.xlabel("Training set size")
plt.ylabel("Training time")
plt.xlim([0, 1000])
plt.legend(loc=0)
plt.show()

Out:
```



The bigger the batch, the faster the training. (Remember that big batches need fewer passes while updating the model.) On the other hand, the bigger the batch, the greater the amount of memory you need in order to store and process the corpora. Thanks to the mini-batches update method, LDA is able to scale to process a corpora of million documents. In fact, the implementation provided by the Gensim package is able to scale and process the whole of Wikipedia in a couple of hours on a domestic computer. If you're brave enough to try it yourself, here are the complete instructions to accomplish the task, provided by the author of the package:

https://radimrehurek.com/gensim/wiki.html

Scaling LDA – memory, CPUs, and machines

Gensim is very flexible and built to crunch big textual corpora; in fact, this library is able to scale without any modification or additional download:

1. With the number of CPUs, allowing parallel processes on a single node (with the classes, as seen in the first example).
2. With the number of observations, allowing online learning based on mini-batches. This can be achieved with the `update` method available in `LdaModel` and `LdaMulticore` (as shown in the previous example).
3. Running it on a cluster, distributing the workload across the nodes in the cluster, thanks to the Python library Pyro4 and the `models.lda_dispatcher` (as a scheduler) and `models.lda_worker` (as a worker process) objects, both provided by Gensim.

Beyond the classical LDA algorithm, Gensim also provides its hierarchical version, named **Hierarchical Dirichlet Processing** (**HDP**). Using this algorithm, topics follow a multilevel structure, enabling the user to understand complex corpora better (that is, where some documents are generic and some specific on a topic). This module is fairly new and, as of the end of 2015, not as scalable as the classic LDA.

Summary

In this chapter, we've introduced three popular unsupervised learners able to scale to cope with big data. The first, PCA, is able to reduce the number of features by creating ones containing the majority of variance (that is, the principal ones). K-means is a clustering algorithm able to group similar points together and associate them with a centroid. LDA is a powerful method to do topic modeling on textual data, that is, model the topics per document and the words appearing in a topic jointly.

In the next chapter, we will introduce some advanced and very recent methods of machine learning, still not part of the mainstream, naturally great for small datasets, but also suitable to process large scale machine learning.

Chapter 8. Distributed Environments – Hadoop and Spark

In this chapter, we will introduce a new way to process data, scaling horizontally. So far, we've focused our attention primarily on processing big data on a standalone machine; here, we will introduce some methods that run on a cluster of machines.

Specifically, we will first illustrate the motivations and circumstances when we need a cluster to process big data. Then, we will introduce the Hadoop framework and all its components with a few examples (HDFS, MapReduce, and YARN), and finally, we will introduce the Spark framework and its Python interface—pySpark.

From a standalone machine to a bunch of nodes

The amount of data stored in the world is increasing exponentially. Nowadays, for a data scientist, having to process a few Terabytes of data a day is not an unusual request. To make things more complex, usually data comes from many different heterogeneous systems and the expectation of business is to produce a model within a short time.

Handling big data, therefore, is not just a matter of size, it's actually a three-dimensional phenomenon. In fact, according to the 3V model, systems operating on big data can be classified using three (orthogonal) criteria:

1. The first criterion is the velocity that the system archives to process the data. Although a few years ago, speed was indicating how quickly a system was able to process a batch; nowadays, velocity indicates whether a system can provide real-time outputs on streaming data.
2. The second criterion is volume, that is, how much information is available to be processed. It can be expressed in number of rows, features, or just a bare count of the bytes. To stream data, volume indicates the throughput of data arriving in the system.
3. The last criterion is variety, that is, the type of data sources. A few years ago, variety was limited by structured datasets; nowadays, data can be structured (tables, images, and so on), semi-structured (JSON, XML, and so on), and unstructured (webpages, social data, and so on). Usually, big data systems try to process as many relevant sources as possible, mixing all kinds of sources.

Beyond these criteria, many other Vs have appeared in the last years, trying to explain other features of big data. Some of these are as follows:

- Veracity (providing an indication of abnormality, bias, and noise contained in the data; ultimately, its accuracy)
- Volatility (indicating for how long the data can be used to extract meaningful information)
- Validity (the correctness of the data)
- Value (indicating the return over investment of the data)

In the recent years, all of the Vs have increased dramatically; now many companies have found that the data they retain has a huge value that can be monetized and they want to extract information out of it. The technical challenge has moved to have enough storage and processing power in order to be able to extract meaningful insights quickly, at scale, and using different input data streams.

Current computers, even the newest and most expensive ones, have a limited amount of disk, memory, and CPU. It looks very hard to process terabytes (or petabytes) of information per day, producing a quick model. Moreover, a standalone server containing both data and processing software needs to be replicated; otherwise, it could become the single point of failure of the system.

The world of big data has therefore moved to clusters: they're composed by a variable number of *not very expensive* nodes and sit on a high-speed Internet connection. Usually, some clusters are dedicated to storing data (a big hard disk, little CPU, and low amount of memory), and others are devoted to processing the data (a powerful CPU, medium-to-big amount of memory, and small hard disk). Moreover, if a cluster is properly set, it can ensure reliability (no single point of failure) and high availability.

Pay attention that, when we store data in a distributed environment (like a cluster), we should also consider the limitation of the CAP theorem; in the system, we can just ensure two out of the following three properties:

- Consistency: All the nodes are able to deliver the same data at the same time to a client
- Availability: A client requesting data is guaranteed to always receive a response, for both succeeded and failed requests
- Partition tolerance: If the network experiences failures and all the nodes cannot be in contact, the system is able to keep working

Specifically, the following are the consequences of the CAP theorem:

- If you give up on consistency, you'll create an environment where data is distributed across the nodes, and even though the network experiences some problems, the system is still able to provide a response to each request, although it is not guaranteed that the response to the same question is the same (it can be inconsistent). Typical examples of this configuration are DynamoDB, CouchDB, and Cassandra.
- If you give up on availability, you'll create a distributed system that can fail to respond to a query. Examples of this class are distributed-cache databases such as Redis, MongoDb, and MemcacheDb.
- Lastly, if you vice up on partition tolerance, you fall in the rigid schema of relational databases that don't allow the network to be split. This category includes MySQL, Oracle, and SQL Server.

Why do we need a distributed framework?

The easiest way to build a cluster is to use some nodes as storage nodes and others as processing ones. This configuration seems very easy as we don't need a complex framework to handle this situation. In fact, many small clusters are exactly built in this way: a couple of servers handle the data (plus their replica) and another bunch process the data.

Although this may appear as a great solution, it's not often used for many reasons:

- It just works for embarrassingly parallel algorithms. If an algorithm requires a common area of memory shared among the processing servers, this approach cannot be used.
- If one or many storage nodes die, the data is not guaranteed to be consistent. (Think about a situation where a node and its replica dies at the same time or where a node dies just after a write operation which has not yet been replicated.
- If a processing node dies, we are not able to keep track of the process that it was executing, making it hard to resume the processing on another node.
- If the network experiences failures, it's very hard to predict the situation after it's back to normality.

Let's now compute what the probability of a node failure is. Is it so rare that we can discard it? Is it something more concrete that we shall always take into consideration? The solution is easy: let's take into consideration a 100-node cluster, where each node has a probability of 1% of failure (cumulative of hardware and software crash) in the first year. What's the probability that all of the 100 will survive the first year? Under the hypothesis that every server is independent (that is, each node can crash independently of all the others), it's simply a multiplication:

$$
\begin{aligned}
P(cluster = ok) &= P(note_1 = ok, node_2 = ok, ..., node_{100} = ok) \\
&= \left(1 - P(fail)\right)^{100} \\
&= 37\%
\end{aligned}
$$

The result is very surprising at the beginning, but it explains why the big data community has put a lot of emphasis on the problem and developed many solutions for the cluster management in the past decade. From the formula's results, it seems that a crash event (or even more than one) is quite likely, a fact requiring that such an occurrence must be thought of in advance and handled properly to ensure the continuity of operations on the data. Furthermore, using cheap hardware or a bigger cluster, it looks almost certain that at least a node will fail.

Tip

The learning point here is that, once you go enterprise with big data, you must adopt enough countermeasures for node failures; it's the norm rather than the exception and should be handled properly to ensure the continuity of operations.

So far, the vast majority of cluster frameworks use the approach named *divide et impera* (split and conquer):

- There are modules *specialized* for the data nodes and some others *specialized* for data processing nodes (also named worker).
- Data is replicated across the data nodes, and one node is the master, ensuring that both the write and read operations succeed.

- The processing steps are split across the worker nodes. They don't share any state (unless stored in the data nodes) and their master ensures that all the tasks are performed positively and in the right order.

Note

Later on, we will introduce the Apache Hadoop framework in this chapter; although it is now a mature cluster management system, it still relies on solid foundations. Before that, let's set up the right working environment on our machines.

Setting up the VM

Setting up a cluster is a long and difficult operation; senior big data engineers earn their (high) salaries not just downloading and executing a binary application, but skillfully and carefully adapting the cluster manager to the desired working environment. It's a tough and complex operation; it may take a long time and if results are below the expectations, the whole business (including data scientists and software developers) won't be able to be productive. Data engineers must know every small detail of the nodes, data, operations that will be carried out, and network before starting to build the cluster. The output is usually a balanced, adaptive, fast, and reliable cluster, which can be used for years by all the technical people in the company.

Note

Is a cluster with a low number of very powerful nodes better than a cluster with many less powerful servers? The answer should be evaluated case-by-case, and it's highly dependent on the data, processing algorithms, number of people accessing it, speed at which we want the results, overall price, robustness of the scalability, network speed, and many other factors. Simply stated, it's not easy at all to decide for the best!

As setting up an environment is very difficult, we authors prefer to provide readers with a virtual machine image containing everything that you need in order to try some operations on a cluster. In the following sections, you'll learn how to set up a guest operating system on your machine, containing one node of a cluster with all the software you'd find on a real cluster.

Why only one node? As the framework that we've used is not lightweight, we decided to go for the atomic piece of a cluster ensuring that the environment you'll find in the node is exactly the same you'll find in a real-world situation. In order to run the virtual machine on your computer, you need two software: Virtualbox and Vagrant. Both of them are free of charge and open source.

VirtualBox

VirtualBox is an open source software used to virtualize one-to-many guest operative systems on Windows, macOS, and Linux host machines. From the user's point of view, a virtualized machine looks like another computer running in a window, with all its functionalities.

VirtualBox has become very popular because of its high performance, simplicity, and clean **graphical user interface** (**GUI**). Starting, stopping, importing, and terminating a virtual machine with VirtualBox is just a matter of a click.

Technically, VirtualBox is a hypervisor, which supports the creation and management of multiple **virtual machines** (**VM**) including many versions of Windows, Linux, and BSD-like distributions. The machine where VirtualBox runs is named *host*, while the virtualized machines are named *guests*. Note that there are no restrictions between the host and guests; for example, a Windows host can run Windows (the same version, a previous, or the most recent one) as well as any Linux and BSD distribution that is VirtualBox-compatible.

Virtualbox is often used to run software Operative-System specific; some software runs only on Windows or just a specific version of Windows, some is available only in Linux, and so on. Another application is to simulate new features on a cloned production environment; before trying the modifications in the live (production) environment, software developers usually test it on a clone, like one running on VirtualBox. Thanks to the guest isolation from the host, if something goes wrong in the guest (even formatting the hard disk), this doesn't impact the host. To have it back, just clone your machine before doing anything dangerous; you'll always be in time to recover it.

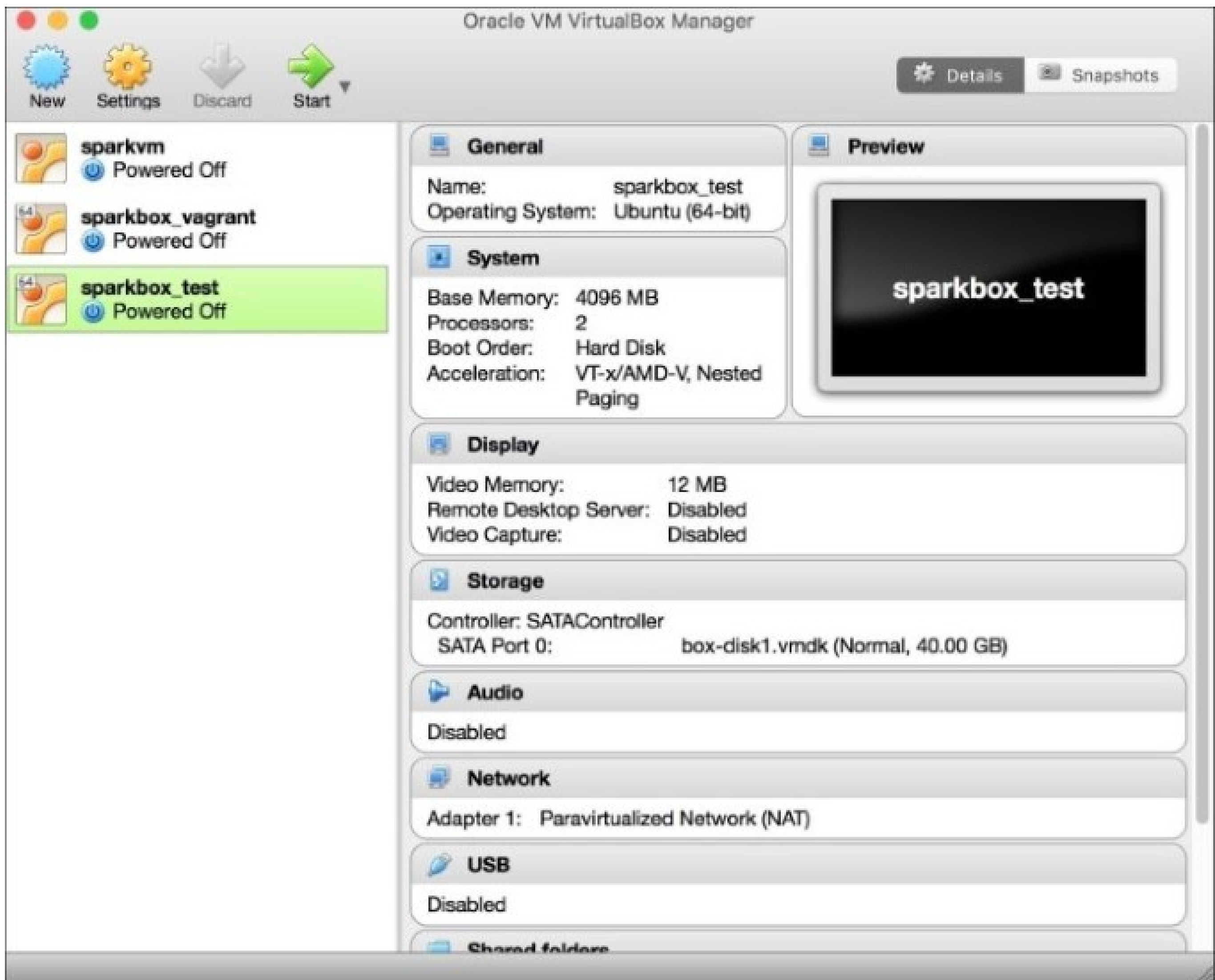
For those who want to start from scratch, VirtualBox supports virtual hard drives (including hard disks, CDs, DVDs, and floppy disks); this makes the installation of a new OS very simple. For example, if you want to install a plain vanilla version of Linux Ubuntu 14.04, you first download the `.iso` file. Instead of burning it on a CD/DVD, you can simply add it as a virtual drive to VirtualBox. Then, thanks to the simple step-by-step interface, you can select the hard drive size and the features of the guest machine (RAM, number of CPUs, video memory, and network connectivity). When operating with real bios, you can select the boot order: selecting the CD/DVD as a higher priority, you can start the process of the installation of Ubuntu as soon as you turn on the guest.

Now, let's download VirtualBox; remember to select the right version for your operating system.

Note

To install it on your computer, follow the instructions at https://www.virtualbox.org/wiki/Downloads.

At the time of writing this, the latest version is 5.1. Once installed, the graphical interface looks like the one in the following screenshot



We strongly advise you to take a look at how to set up a guest machine on your machine. Each guest machine will appear on the left-hand side of the window. (In the image, you can see that, on our computer, we have three stopped guests.) By clicking on each of them, on the right side will appear the detailed description of the virtualized hardware. In the example image, if the virtual machine named `sparkbox_test` (the one highlighted on the left) is turned on, it will be run on a virtual computer whose hardware is composed by a 4GB RAM, two processors, 40GB hard drive, and a video card with 12MB of RAM attached to the network with NAT.

Vagrant

Vagrant is a software that configures virtual environments at a high level. The core piece of Vagrant is the scripting capability, often used to create programmatically and automatically specific virtual environments. Vagrant uses VirtualBox (but also other virtualizers) to build and configure the virtual machines.

Note

To install it, follow the instructions at https://www.vagrantup.com/downloads.html.

Using the VM

With Vagrant and VirtualBox installed, you're now ready to run the node of a cluster environment. Create an empty directory and insert the following Vagrant commands into a new file named `Vagrantfile`:

```
Vagrant.configure("2") do |config|
    config.vm.box = "sparkpy/sparkbox_test_1"
    config.vm.hostname = "sparkbox"
    config.ssh.insert_key = false

    # Hadoop ResourceManager
    config.vm.network :forwarded_port, guest: 8088, host: 8088,
auto_correct: true

    # Hadoop NameNode
    config.vm.network :forwarded_port, guest: 50070, host: 50070,
auto_correct: true

    # Hadoop DataNode
    config.vm.network :forwarded_port, guest: 50075, host: 50075,
auto_correct: true

    # Ipython notebooks (yarn and standalone)
    config.vm.network :forwarded_port, guest: 8888, host: 8888,
auto_correct: true

    # Spark UI (standalone)
    config.vm.network :forwarded_port, guest: 4040, host: 4040,
auto_correct: true


    config.vm.provider "virtualbox" do |v|
      v.customize ["modifyvm", :id, "--natdnshostresolver1", "on"]
    v.customize ["modifyvm", :id, "--natdnsproxy1", "on"]
v.customize ["modifyvm", :id, "--nictype1", "virtio"]
    v.name = "sparkbox_test"
    v.memory = "4096"
    v.cpus = "2"
    end

end
```

From top to bottom, the first lines download the right virtual machine (that we authors created and uploaded on a repository). Then, we set some ports to be forwarded to the guest machine; in this way, you'll be able to access some webservices of the virtualized machine. Finally, we set the hardware of the node.

Note

The configuration is set for a virtual machine with exclusive use of 4GB RAM and two cores. If your system can't meet these requirements, modify `v.memory` and `v.cpus` values to those good for your machine. Note that some of the following code examples may fail if the configuration that you set is not adequate.

Now, open a terminal and navigate to the directory containing the `Vagrantfile`. Here, launch the virtual machine with the following command:

```
$ vagrant up
```

The first time, this command will take a while as it downloads (it's an almost 2GB download) and builds the correct structure of the virtual machine. The next time, this command takes a smaller amount of time as there is nothing more to download.

After having turned on the virtual machine on your local system, you can access it as follows:

```
$ vagrant ssh
```

This command simulates an SSH access, and you'll be inside the virtualized machine finally.

Note

On Windows machines, this command may fail with an error due to the missing SSH executable. In such a situation, download and install an SSH client for Windows, such as Putty (http://www.putty.org/), Cygwin openssh (http://www.cygwin.com/), or Openssh for Windows (http://sshwindows.sourceforge.net/). Unix systems should not be affected by this problem.

To turn if off, you first need to exit the machine. From inside the VM, simply use the `exit` command to exit the SSH connection and then shut down the VM:

```
$ vagrant halt
```

Note

The virtual machine consumes resources. Remember to turn it off when you're done with the work with the `vagranthalt` command from the directory where the VM is located.

The preceding command shuts down the virtual machine, exactly as you would do with a server. To remove it and delete all its content, use the `vagrant destroy` command. Use it carefully: after having destroyed the machine, you won't be able to recover the files in there.

Here are the instructions to use IPython (Jupyter) Notebook inside the virtual machine:

1. Launch `vagrant up` and `vagrant ssh` from the folder containing the `Vagrantfile`. You should now be inside the virtual machine.
2. Now, launch the script:

    ```
    vagrant@sparkbox:~$ ./start_hadoop.sh
    ```

3. At this point, launch the following shell script:

    ```
    vagrant@sparkbox:~$ ./start_jupyter_yarn.sh
    ```

    Open a browser on your local machine and point it to `http://localhost:8888`.

Here is the notebook that's backed by the cluster node. To turn the notebook and virtual machine off, perform the following steps:

1. To terminate the Jupyter console, press *Ctrl* + *C* (and then type *Y* for Yes).
2. Terminate the Hadoop framework as follows:

    ```
    vagrant@sparkbox:~$ ./stop_hadoop.sh
    ```

3. Exit the virtual machine with the following command:

    ```
    vagrant@sparkbox:~$ exit
    ```

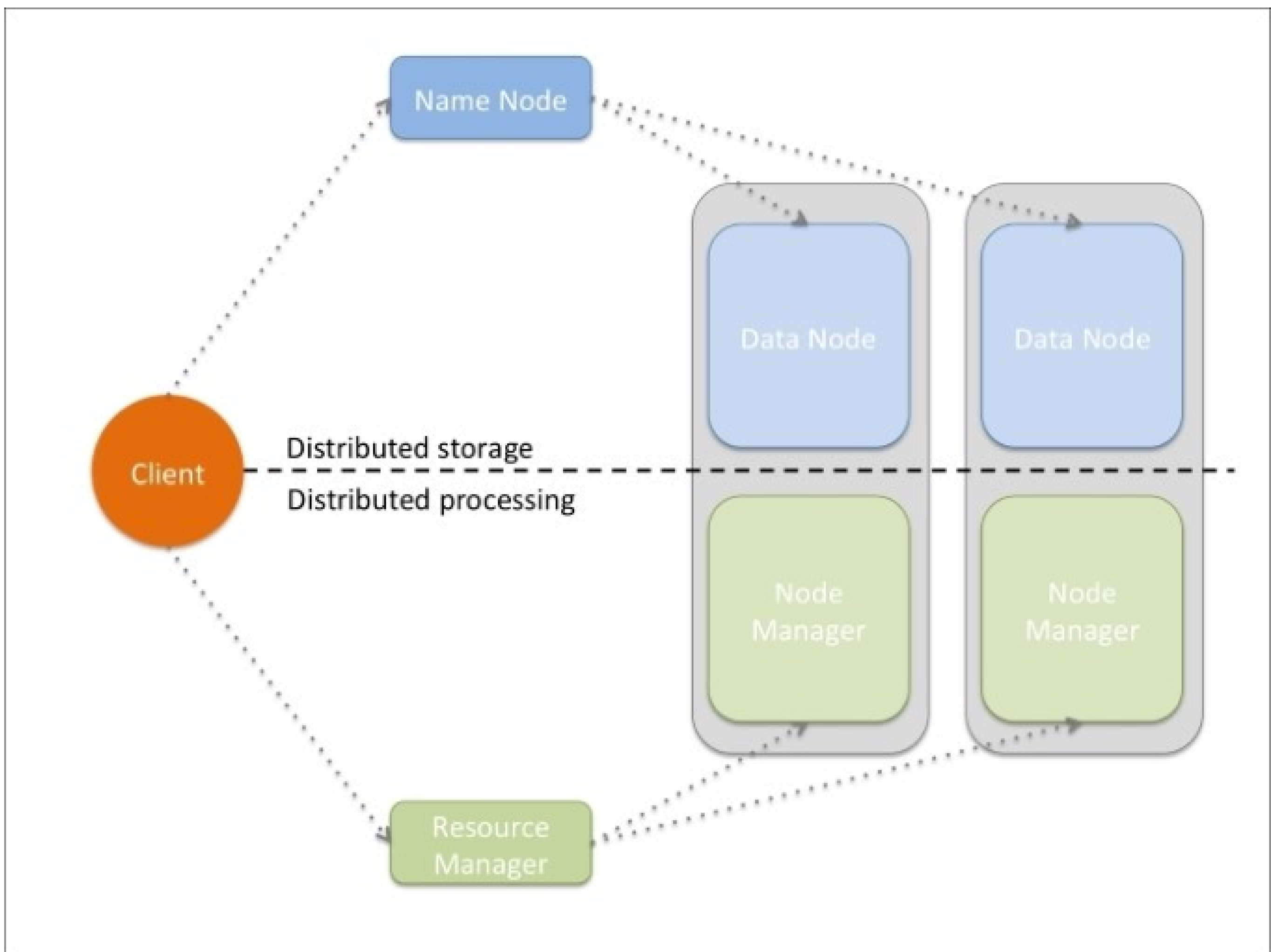
4. Shut down the VirtualBox machine with `vagrant halt`.

The Hadoop ecosystem

Apache Hadoop is a very popular software framework for distributed storage and distributed processing on a cluster. Its strengths are in the price (it's free), flexibility (it's open source, and although being written in Java, it can by used by other programming languages), scalability (it can handle clusters composed by thousands of nodes), and robustness (it was inspired by a published paper from Google and has been around since 2011), making it the de facto standard to handle and process big data. Moreover, lots of other projects from the Apache foundation extend its functionalities.

Architecture

Logically, Hadoop is composed of two pieces: distributed storage (HDFS) and distributed processing (YARN and MapReduce). Although the code is very complex, the overall architecture is fairly easy to understand. A client can access both storage and processing through two dedicated modules; they are then in charge of distributing the job across all theorking nodes:



All the Hadoop modules run as services (or instances), that is, a physical or virtual node can run many of them. Typically, for small clusters, all the nodes run both distributed computing and processing services; for big clusters, it may be better to separate the two functionalities specializing the nodes.

We will see the functionalities offered by the two layers in detail.

HDFS

The **Hadoop Distributed File System** (**HDFS**) is a fault-tolerant distributed filesystem, designed to run on commodity low-cost hardware and able to handle very large datasets (in the order of hundred petabytes to exabytes). Although HDFS requires a fast network connection to transfer data across nodes, the latency can't be as low as in classic filesystems (it may be in the order of seconds); therefore, HDFS has been designed for batch processing and high throughput. Each HDFS node contains a part of the filesystem's data; the same data is also replicated in other instances and this ensures a high throughput access and fault-tolerance.

HDFS's architecture is master-slave. If the master (Name Node) fails, there is a secondary/backup one ready to take control. All the other instances are slaves (Data Nodes); if one of them fails, there's no problem as HDFS has been designed with this in mind.

Data Nodes contain blocks of data: each file saved in HDFS is broken up in chunks (or blocks), typically 64MB each, and then distributed and replicated in a set of Data Nodes.

The Name Node stores just the metadata of the files in the distributed filesystem; it doesn't store any actual data, but just the right indications on how to access the files in the multiple Data Nodes that it manages.

A client asking to read a file shall first contact the Name Node, which will give back a table containing an ordered list of blocks and their locations (as in Data Nodes). At this point, the client should contact the Data Nodes separately, downloading all the blocks and reconstructing the file (by appending the blocks together).

To write a file, instead, a client should first contact the Name Node, which will first decide how to handle the request, updating its records and then replying to the client with an ordered list of Data Nodes of where to write each block of the file. The client will now contact and upload the blocks to the Data Nodes, as reported in the Name Node reply.

Namespace queries (for example, listing a directory content, creating a folder, and so on) are instead completely handled by the Name Node by accessing its metadata information.

Moreover, Name Node is also responsible for handling a Data Node failure properly (it's marked as dead if no Heartbeat packets are received) and its data re-replication to other nodes.

Although these operations are long and hard to be implemented with robustness, they're completely transparent to the user, thanks to many libraries and the HDFS shell. The way you operate on HDFS is pretty similar to what you're currently doing on your filesystem and this is a great benefit of Hadoop: hiding the complexity and letting the user use it with simplicity.

Let's now take a look at the HDFS shell and later, a Python library.

Note

Use the preceding instructions to turn the VM on and launch the IPython Notebook on your computer.

Now, open a new notebook; this operation will take more time than usual as each notebook is connected to the Hadoop cluster framework. When the notebook is ready to be used, you'll see a flag saying **Kernel starting, please wait …** on the top right disappear.

The first piece is about the HDFS shell; therefore, all the following commands can be run at a prompt or shell of the virtualized machine. To run them in an IPython Notebook, all of them are anticipated by a question mark `!`, which is a short way to execute bash code in a notebook.

The common denominator of the following command lines is the executable; we will always run the `hdfs` command. It's the main interface to access and manage the HDFS system and the main command for the HDFS shell.

We start with a report on the state of HDFS. To obtain the details of the **distributed filesystem** (**dfs**) and its Data Nodes, use the `dfsadmin` subcommand:

```
In:!hdfs dfsadmin -report

Out:Configured Capacity: 42241163264 (39.34 GB)
Present Capacity: 37569168058 (34.99 GB)
DFS Remaining: 37378433024 (34.81 GB)
DFS Used: 190735034 (181.90 MB)
DFS Used%: 0.51%
Under replicated blocks: 0
Blocks with corrupt replicas: 0
Missing blocks: 0

-------------------------------------------------
Live datanodes (1):

Name: 127.0.0.1:50010 (localhost)
Hostname: sparkbox
Decommission Status : Normal
Configured Capacity: 42241163264 (39.34 GB)
DFS Used: 190735034 (181.90 MB)
Non DFS Used: 4668290330 (4.35 GB)
DFS Remaining: 37380775936 (34.81 GB)
DFS Used%: 0.45%
DFS Remaining%: 88.49%
Configured Cache Capacity: 0 (0 B)
Cache Used: 0 (0 B)
Cache Remaining: 0 (0 B)
Cache Used%: 100.00%
```

```
Cache Remaining%: 0.00%
Xceivers: 1
Last contact: Tue Feb 09 19:41:17 UTC 2016
```

The dfs subcommand allows using some well-known Unix commands to access and interact with the distributed filesystem. For example, list the content of the the root directory as follows:

```
In:!hdfs dfs -ls /

Out:Found 2 items
drwxr-xr-x   - vagrant supergroup          0 2016-01-30 16:33 /spark
drwxr-xr-x   - vagrant supergroup          0 2016-01-30 18:12 /user
```

The output is similar to the `ls` command provided by Linux, listing the permissions, number of links, user and group owning the file, size, timestamp of the last modification, and name for each file or directory.

Similar to the `df` command, we can invoke the `-df` argument to display the amount of available disk space in HDFS. The `-h` option will make the output more readable (using gigabytes and megabytes instead of bytes):

```
In:!hdfs dfs -df -h /

Out:Filesystem               Size     Used  Available  Use%
hdfs://localhost:9000  39.3 G  181.9 M     34.8 G    0%
```

Similar to `du`, we can use the `-du` argument to display the size of each folder contained in the root. Again, `-h` will produce a more human readable output:

```
In:!hdfs dfs -du -h /

Out:178.9 M  /spark
1.4 M    /user
```

So far, we've extracted some information from HDFS. Let's now do some operations on the distributed filesystem, which will modify it. We can start with creating a folder with the `-mkdir` option followed by the name. Note that this operation may fail if the directory already exists (exactly as in Linux, with the `mkdir` command):

```
In:!hdfs dfs -mkdir /datasets
```

Let's now transfer some files from the hard disk of the node to the distributed filesystem. In the VM that we've created, there is already a text file in the `../datasets` directory; let's download a text file from the Internet. Let's move both of them to the HDFS directory that we've created with the previous command:

```
In:
!wget -q http://www.gutenberg.org/cache/epub/100/pg100.txt \
    -O ../datasets/shakespeare_all.txt

!hdfs dfs -put ../datasets/shakespeare_all.txt \
/datasets/shakespeare_all.txt

!hdfs dfs -put ../datasets/hadoop_git_readme.txt \
/datasets/hadoop_git_readme.txt
```

Was the importing successful? Yes, we didn't have any errors. However, to remove any doubt, let's list the HDFS directory/datasets to see the two files:

```
In:!hdfs dfs -ls /datasets

Out:Found 2 items
-rw-r--r--   1 vagrant supergroup       1365 2016-01-31 12:41
/datasets/hadoop_git_readme.txt
-rw-r--r--   1 vagrant supergroup    5589889 2016-01-31 12:41
/datasets/shakespeare_all.txt
```

To concatenate some files to the standard output, we can use the `-cat` argument. In the following piece of code, we're counting the new lines appearing in a text file. Note that the first command is piped into another command that is operating on the local machine:

```
In:!hdfs dfs -cat /datasets/hadoop_git_readme.txt | wc –l

Out:30
```

Actually, with the `-cat` argument, we can concatenate multiple files from both the local machine and HDFS. To see it, let's now count how many newlines are present when the file stored on HDFS is concatenated to the same one stored on the local machine. To avoid misinterpretations, we can use the full **Uniform Resource Identifier** (**URI**), referring to the files in HDFS with the `hdfs:` scheme and to local files with the `file:` scheme:

```
In:!hdfs dfs -cat \
    hdfs:///datasets/hadoop_git_readme.txt \
    file:///home/vagrant/datasets/hadoop_git_readme.txt | wc –l

Out:60
```

In order to copy in HDFS, we can use the `-cp` argument:

```
In : !hdfs dfs -cp /datasets/hadoop_git_readme.txt \
/datasets/copy_hadoop_git_readme.txt
```

To delete a file (or directories, with the right option), we can use the `-rm` argument. In this snippet of code, we're removing the file that we've just created with the preceding command. Note that HDFS has the thrash mechanism; consequently, a deleted file is not actually removed from the HDFS but just moved to a special directory:

```
In:!hdfs dfs -rm /datasets/copy_hadoop_git_readme.txt

Out:16/02/09 21:41:44 INFO fs.TrashPolicyDefault: Namenode trash
configuration: Deletion interval = 0 minutes, Emptier interval = 0
minutes.

Deleted /datasets/copy_hadoop_git_readme.txt
```

To empty the thrashed data, here's the command:

```
In:!hdfs dfs –expunge

Out:16/02/09 21:41:44 INFO fs.TrashPolicyDefault: Namenode trash
configuration: Deletion interval = 0 minutes, Emptier interval = 0
minutes.
```

To obtain (get) a file from HDFS to the local machine, we can use the `-get` argument:

```
In:!hdfs dfs -get /datasets/hadoop_git_readme.txt \
/tmp/hadoop_git_readme.txt
```

To take a look at a file stored in HDFS, we can use the `-tail` argument. Note that there's no head function in HDFS as it can be done using `cat` and the result then piped in a local head command. As for the tail, the HDFS shell just displays the last kilobyte of data:

```
In:!hdfs dfs -tail /datasets/hadoop_git_readme.txt

Out:ntry, of
encryption software.  BEFORE using any encryption software, please
check your country's laws, regulations and policies concerning the
import, possession, or use, and re-export of encryption software, to
see if this is permitted.  See <http://www.wassenaar.org/> for more
information.
[...]
```

The `hdfs` command is the main entry point for HDFS, but it's slow and invoking system commands from Python and reading back the output is very tedious. For this, there exists a library for Python, Snakebite, which wraps many distributed filesystem operations. Unfortunately, the library is not as complete as the HDFS shell and is bound to a Name Node. To install it on your local machine, simply use `pip install snakebite`.

To instantiate the client object, we should provide the IP (or its alias) and the port of the Name Node. In the VM we provided, it's running on port 9000:

```
In:from snakebite.client import Client
client = Client("localhost", 9000)
```

To print some information about the HDFS, the client object has the `serverdefaults` method:

```
In:client.serverdefaults()

Out:{'blockSize': 134217728L,
 'bytesPerChecksum': 512,
 'checksumType': 2,
 'encryptDataTransfer': False,
 'fileBufferSize': 4096,
 'replication': 1,
 'trashInterval': 0L,
 'writePacketSize': 65536}
```

To list the files and directories in the root, we can use the `ls` method. The result is a list of dictionaries, one for each file, containing information such as permissions, timestamp of the last modification, and so on. In this example, we're just interested in the paths (that is, the names):

```
In:for x in client.ls(['/']):
    print x['path']

Out:/datasets
/spark
/user
```

Exactly as the preceding code, the Snakebite client has the `du` (for disk usage) and `df` (for disk free) methods available. Note that many methods (like `du`) return generators, which means that they need to be consumed (like an iterator or list) to be executed:

```
In:client.df()

Out:{'capacity': 42241163264L,
 'corrupt_blocks': 0L,
 'filesystem': 'hdfs://localhost:9000',
 'missing_blocks': 0L,
 'remaining': 37373218816L,
 'under_replicated': 0L,
 'used': 196237268L}

In:list(client.du(["/"]))

Out:[{'length': 5591254L, 'path': '/datasets'},
```

```
 {'length': 187548272L, 'path': '/spark'},
 {'length': 1449302L, 'path': '/user'}]
```

As for the HDFS shell example, we will now try to count the newlines appearing in the same file with Snakebite. Note that the `.cat` method returns a generator:

```
In:
for el in client.cat(['/datasets/hadoop_git_readme.txt']):
    print el.next().count("\n")

Out:30
```

Let's now delete a file from HDFS. Again, pay attention that the `delete` method returns a generator and the execution never fails, even if we're trying to delete a non-existing directory. In fact, Snakebite doesn't raise exceptions, but just signals to the user in the output dictionary that the operation failed:

```
In:client.delete(['/datasets/shakespeare_all.txt']).next()

Out:{'path': '/datasets/shakespeare_all.txt', 'result': True}
```

Now, let's copy a file from HDFS to the local filesystem. Observe that the output is a generator, and you need to check the output dictionary to see if the operation was successful:

```
In:
(client
.copyToLocal(['/datasets/hadoop_git_readme.txt'],
             '/tmp/hadoop_git_readme_2.txt')
.next())

Out:{'error': '',
 'path': '/tmp/hadoop_git_readme_2.txt',
 'result': True,
 'source_path': '/datasets/hadoop_git_readme.txt'}
```

Finally, create a directory and delete all the files matching a string:

```
In:list(client.mkdir(['/datasets_2']))

Out:[{'path': '/datasets_2', 'result': True}]

In:client.delete(['/datasets*'], recurse=True).next()

Out:{'path': '/datasets', 'result': True}
```

Where is the code to put a file in HDFS? Where is the code to copy an HDFS file to another one? Well, these functionalities are not yet implemented in Snakebite. For them, we shall use the HDFS shell through system calls.

MapReduce

MapReduce is the programming model implemented in the earliest versions of Hadoop. It's a very simple model, designed to process large datasets on a distributed cluster in parallel batches. The core of MapReduce is composed of two programmable functions—a mapper that performs filtering and a reducer that performs aggregation—and a shuffler that moves the objects from the mappers to the right reducers.

Note

Google has published a paper in 2004 on Mapreduce, a few months after having been granted a patent on it.

Specifically, here are the steps of MapReduce for the Hadoop implementation:

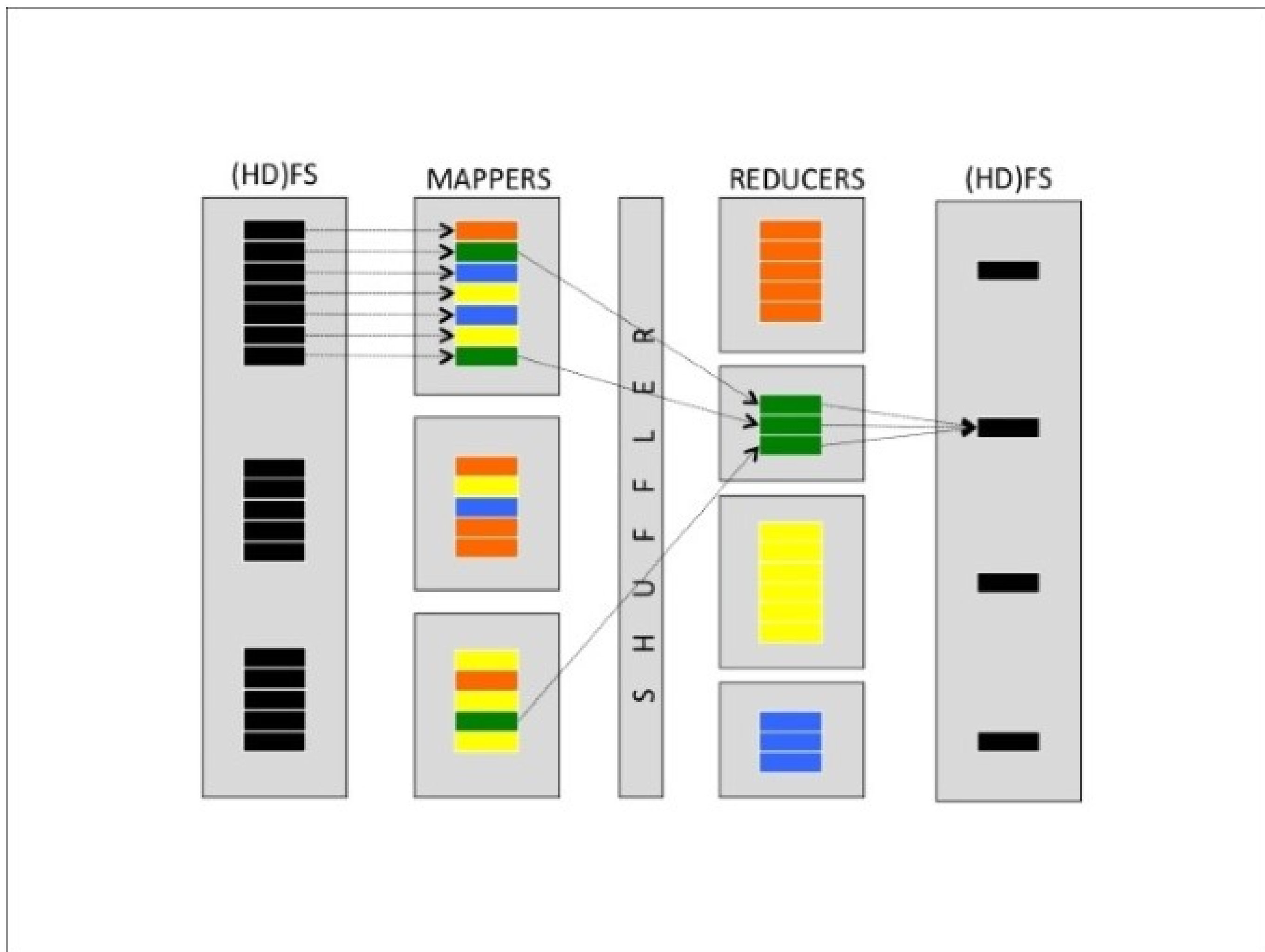
1. Data chunker. Data is read from the filesystem and split into chunks. A chunk is a piece of the input dataset, typically either a fixed-size block (for example, a HDFS block read from a Data Node) or another more appropriate split.

   For example, if we want to count the number of characters, words, and lines in a text file, a nice split can be a line of text.
2. Mapper: From each chunk, a series of key-value pairs is generated. Each mapper instance applies the same mapping function on different chunks of data.

   Continuing the preceding example, for each line, three key-value pairs are generated in this step—one containing the number of characters in the line (the key can simply be a *chars* string), one containing the number of words (in this case, the key must be different, let's say *words*), and one containing the number of lines, which is always one (in this case, the key can be *lines*).
3. Shuffler: From the key and number of available reducers, the shuffler distributes all the key-value pairs with the same key to the same reducers. Typically, this operation is the hash of the key, modulo the number of reducers. This should ensure a fair amount of keys for each reducer. This function is not user-programmable, but provided by the MapReduce framework.
4. Reducer: Each reducer receives all the key-value pairs for a specific set of keys and can produce zero or more aggregate results.

   In the example, all the values connected to the *words* key arrive at a reducer; its job is just summing up all the values. The same happens for the other keys, resulting in three final values: the number of characters, number of words, and number of lines. Note that these results may be on different reducers.
5. Output writer: The outputs of the reducers are written on the filesystem (or HDFS). In the default Hadoop configuration, each reducer writes a file (`part-r-00000` is the output of the first reducer, `part-r-00001` of the second, and so on). To have a full list of results on a file, you should concatenate all of them.

   Visually, this operation can be simply communicated and understood as follows:



There's also an optional step that can be run by each mapper instance after the mapping step—the combiner. It basically anticipates, if possible, the reducing step on the mapper and is often used to decrease the amount of information to shuffle, speeding up the process. In the preceding example, if a mapper processes more than one line of the input file, during the (optional) combiner step, it can pre-aggregate the results, outputting the smaller number of key-value pairs. For example, if the mapper processes 100 lines of text in each chunk, why output 300 key-value pairs (100 for the number of chars, 100 for words, and 100 for lines) when the information can be aggregated in three? That's actually the goal of the combiner.

In the MapReduce implementation provided by Hadoop, the shuffle operation is distributed, optimizing the communication cost, and it's possible to run more than one mapper and reducer per node, making full use of the hardware resources available on the nodes. Also, the Hadoop infrastructure provides redundancy and fault-tolerance as the same task can be assigned to multiple workers.

Let's now see how it works. Although the Hadoop framework is written in Java, thanks to the Hadoop Streaming utility, mappers and reducers can be any executable, including Python. Hadoop Streaming

uses the pipe and standard inputs and outputs to stream the content; therefore, mappers and reducers must implement a reader from stdin and a key-value writer on stdout.

Now, turn on the virtual machine and open a new IPython notebook. Even in this case, we will first introduce the command line way to run MapReduce jobs provided by Hadoop, then introduce a pure Python library. The first example will be exactly what we've described: a counter of the number of characters, words, and lines of a text file.

First, let's insert the datasets into HDFS; we're going to use the Hadoop Git readme (a short text file containing the readme file distributed with Apache Hadoop) and the full text of all the Shakespeare books, provided by Project Gutenberg (although it's just 5MB, it contains almost 125K lines). In the first cell, we'll be cleaning up the folder from the previous experiment, then, we download the file containing the Shakespeare bibliography in the dataset folder, and finally, we put both datasets on HDFS:

```
In:!hdfs dfs -mkdir -p /datasets
!wget -q http://www.gutenberg.org/cache/epub/100/pg100.txt \
    -O ../datasets/shakespeare_all.txt
!hdfs dfs -put -f ../datasets/shakespeare_all.txt /datasets/
shakespeare_all.txt
!hdfs dfs -put -f ../datasets/hadoop_git_readme.txt /datasets/
hadoop_git_readme.txt
!hdfs dfs -ls /datasets
```

Now, let's create the Python executable files containing the mapper and reducer. We will use a very dirty hack here: we're going to write Python files (and make them executable) using a write operation from the Notebook.

Both the mapper and reducer read from the stdin and write to the stdout (with simple print commands). Specifically, the mapper reads lines from the stdin and prints the key-value pairs of the number of characters (except the newline), the number of words (by splitting the line on the whitespace), and the number of lines, always one. The reducer, instead, sums up the values for each key and prints the grand total:

```
In:
with open('mapper_hadoop.py', 'w') as fh:
    fh.write("""#!/usr/bin/env python

import sys

for line in sys.stdin:
    print "chars", len(line.rstrip('\\n'))
    print "words", len(line.split())
    print "lines", 1
    """)

with open('reducer_hadoop.py', 'w') as fh:
    fh.write("""#!/usr/bin/env python
```

```
import sys

counts = {"chars": 0, "words":0, "lines":0}

for line in sys.stdin:
    kv = line.rstrip().split()
    counts[kv[0]] += int(kv[1])

for k,v in counts.items():
    print k, v
    """)
```

```
In:!chmod a+x *_hadoop.py
```

To see it at work, let's try it locally without using Hadoop. In fact, as mappers and reducers read and write to the standard input and output, we can just pipe all the things together. Note that the shuffler can be replaced by the `sort -k1,1` command, which sorts the input strings using the first field (that is, the key):

```
In:!cat ../datasets/hadoop_git_readme.txt | ./mapper_hadoop.py |
sort -k1,1 | ./reducer_hadoop.py
```

```
Out:chars 1335
lines 31
words 179
```

Let's now use the Hadoop MapReduce way to get the same result. First of all, we should create an empty directory in HDFS able to store the results. In this case, we create a directory named `/tmp` and we remove anything inside named in the same way as the job output (Hadoop will fail if the output file already exists). Then, we use the right command to run the MapReduce job. This command includes the following:

- The fact that we want to use the Hadoop Streaming capability (indicating the Hadoop streaming jar file)
- The mappers and reducers that we want to use (the `-mapper` and `-reducer` options)
- The fact that we want to distribute these files to each mapper as they're local files (with the `-files` option)
- The input file (the `-input` option) and the output directory (the `-output` option)

```
In:!hdfs dfs -mkdir -p /tmp
!hdfs dfs -rm -f -r /tmp/mr.out

!hadoop jar /usr/local/hadoop/share/hadoop/tools/lib/
hadoop-streaming-2.6.4.jar \
-files mapper_hadoop.py,reducer_hadoop.py \
-mapper mapper_hadoop.py -reducer reducer_hadoop.py \
```

```
-input /datasets/hadoop_git_readme.txt -output /tmp/mr.out

Out:[...]
16/02/04 17:12:22 INFO mapreduce.Job: Running job:
job_1454605686295_0003
16/02/04 17:12:29 INFO mapreduce.Job: Job job_1454605686295_0003
running in uber mode : false
16/02/04 17:12:29 INFO mapreduce.Job:  map 0% reduce 0%
16/02/04 17:12:35 INFO mapreduce.Job:  map 50% reduce 0%
16/02/04 17:12:41 INFO mapreduce.Job:  map 100% reduce 0%
16/02/04 17:12:47 INFO mapreduce.Job:  map 100% reduce 100%
16/02/04 17:12:47 INFO mapreduce.Job: Job job_1454605686295_0003
completed successfully
[...]
    Shuffle Errors
        BAD_ID=0
        CONNECTION=0
        IO_ERROR=0
        WRONG_LENGTH=0
        WRONG_MAP=0
        WRONG_REDUCE=0
[...]
16/02/04 17:12:47 INFO streaming.StreamJob: Output directory: /tmp/
mr.out
```

The output is very verbose; we just extracted three important sections in it. The first indicates the progress of the MapReduce job, and it's very useful to track and estimate the time needed to complete the operation. The second section highlights the errors, which may have occurred during the job, and the last section reports the output directory and timestamp of the termination. The whole process on the small file (of 30 lines) took almost half a minute! The reasons are very simple: first, Hadoop MapReduce has been designed for robust big data processing and contains a lot of overhead, and second, the ideal environment is a cluster of powerful machines, not a virtualized VM with 4GB of RAM. On the other hand, this code can be run on much bigger datasets and a cluster of a very powerful machine, without changing anything.

Let's not see the results immediately. First, let's take a peek at the output directory in HDFS:

```
In:!hdfs dfs -ls /tmp/mr.out

Out:Found 2 items
-rw-r--r--   1 vagrant supergroup          0 2016-02-04 17:12 /tmp/
mr.out/_SUCCESS
-rw-r--r--   1 vagrant supergroup         33 2016-02-04 17:12 /tmp/
mr.out/part-00000
```

There are two files: the first is empty and named `_SUCCESS` and indicates that the MapReduce job has finished the writing stage in the directory, and the second is named part-00000 and contains the actual

results (as we're operating on a node with just one reducer). Reading this file will provide us with the final results:

```
In:!hdfs dfs -cat /tmp/mr.out/part-00000

Out:chars 1335
lines 31
words 179
```

As expected, they're the same as the piped command line shown previously.

Although conceptually simple, Hadoop Streaming is not the best way to run Hadoop jobs with Python code. For this, there are many libraries available on Pypy; the one we're presenting here is one of the most flexible and maintained open source one—**MrJob**. It allows you to run the jobs seamlessly on your local machine, your Hadoop cluster, or the same cloud cluster environments, such as Amazon Elastic MapReduce; it merges all the code in a standalone file even if multiple MapReduce steps are needed (think about iterative algorithms) and interprets Hadoop errors in the code. Also, it's very simple to install; to have the MrJob library on your local machine, simply use `pip install mrjob`.

Although MrJob is a great piece of software, it doesn't work very well with IPython Notebook as it requires a main function. Here, we need to write the MapReduce Python code in a separate file and then run a command line.

We start with the example that we've seen many times so far: counting characters, words, and lines in a file. First, let's write the Python file using the MrJob functionalities; mappers and reducers are *wrapped* in a subclass of `MRJob`. Inputs are not read from stdin, but passed as a function argument, and outputs are not printed, but yielded (or returned).

Thanks to MrJob, the whole MapReduce program becomes just a few lines of code:

```
In:
with open("MrJob_job1.py", "w") as fh:
    fh.write("""
from mrjob.job import MRJob


class MRWordFrequencyCount(MRJob):

    def mapper(self, _, line):
        yield "chars", len(line)
        yield "words", len(line.split())
        yield "lines", 1

    def reducer(self, key, values):
        yield key, sum(values)
```

```
if __name__ == '__main__':
    MRWordFrequencyCount.run()
    """)
```

Let's now execute it locally (with the local version of the dataset).The MrJob library, beyond executing the mapper and reducer steps (locally, in this case), also prints the result and cleans up the temporary directory:

```
In:!python MrJob_job1.py ../datasets/hadoop_git_readme.txt

Out: [...]
Streaming final output from /tmp/
MrJob_job1.vagrant.20160204.171254.595542/output
"chars"    1335
"lines"    31
"words"    179
removing tmp directory /tmp/MrJob_job1.vagrant.20160204.171254.595542
```

To run the same process on Hadoop, just run the same Python file, this time inserting the `-r hadoop` option in the command line, and automatically MrJob will execute it using Hadoop MapReduce and HDFS. In this case, remember to point the `hdfs` path of the input file:

```
In:
!python MrJob_job1.py -r hadoop hdfs:///datasets/
hadoop_git_readme.txt

Out:[...]
HADOOP: Running job: job_1454605686295_0004
HADOOP: Job job_1454605686295_0004 running in uber mode : false
HADOOP:  map 0% reduce 0%
HADOOP:  map 50% reduce 0%
HADOOP:  map 100% reduce 0%
HADOOP:  map 100% reduce 100%
HADOOP: Job job_1454605686295_0004 completed successfully
[...]
HADOOP:     Shuffle Errors
HADOOP:         BAD_ID=0
HADOOP:         CONNECTION=0
HADOOP:         IO_ERROR=0
HADOOP:         WRONG_LENGTH=0
HADOOP:         WRONG_MAP=0
HADOOP:         WRONG_REDUCE=0
[...]
Streaming final output from hdfs:///user/vagrant/tmp/mrjob/
MrJob_job1.vagrant.20160204.171255.073506/output
"chars"    1335
"lines"    31
```

```
"words"    179
removing tmp directory /tmp/MrJob_job1.vagrant.20160204.171255.073506
deleting hdfs:///user/vagrant/tmp/mrjob/
MrJob_job1.vagrant.20160204.171255.073506 from HDFS
```

You will see the same output of the Hadoop Streaming command line as seen previously, plus the results. In this case, the HDFS temporary directory, used to store the results, is removed after the termination of the job.

Now, to see the flexibility of MrJob, let's try running a process that requires more than one MapReduce Step. While done from the command line, this is a very difficult task; in fact, you have to run the first iteration of MapReduce, check the errors, read the results, and then launch the second iteration of MapReduce, check the errors again, and finally read the results. This sounds very time-consuming and prone to errors. Thanks to MrJob, this operation is very easy: within the code, it's possible to create a cascade of MapReduce operations, where each output is the input of the next stage.

As an example, let's now find the most common word used by Shakespeare (using, as input, the 125K lines file). This operation cannot be done in a single MapReduce step; it requires at least two of them. We will implement a very simple algorithm based on two iterations of MapReduce:

- Data chunker: Just as for the MrJob default, the input file is split on each line.
- Stage 1 – map: A key-map tuple is yielded for each word; the key is the lowercased word and the value is always 1.
- Stage 1 – reduce: For each key (lowercased word), we sum all the values. The output will tell us how many times the word appears in the text.
- Stage 2 – map: During this step, we flip the key-value tuples and put them as values of a new key pair. To force one reducer to have all the tuples, we assign the same key, *None*, to each output tuple.
- Stage 2 – reduce: We simply discard the only key available and extract the maximum of the values, resulting in extracting the maximum of all the tuples (count, word).

```
In:
with open("MrJob_job2.py", "w") as fh:
    fh.write("""
from mrjob.job import MRJob
from mrjob.step import MRStep
import re

WORD_RE = re.compile(r"[\w']+")


class MRMostUsedWord(MRJob):

    def steps(self):
        return [
            MRStep(mapper=self.mapper_get_words,
                   reducer=self.reducer_count_words),
            MRStep(mapper=self.mapper_word_count_one_key,
```

```
                    reducer=self.reducer_find_max_word)
        ]

    def mapper_get_words(self, _, line):
        # yield each word in the line
        for word in WORD_RE.findall(line):
            yield (word.lower(), 1)

    def reducer_count_words(self, word, counts):
        # send all (num_occurrences, word) pairs to the same reducer.
        yield (word, sum(counts))

    def mapper_word_count_one_key(self, word, counts):
        # send all the tuples to same reducer
        yield None, (counts, word)

    def reducer_find_max_word(self, _, count_word_pairs):
        # each item of word_count_pairs is a tuple (count, word),
        yield max(count_word_pairs)


if __name__ == '__main__':
    MRMostUsedWord.run()
""")
```

We can then decide to run it locally or on the Hadoop cluster, obtaining the same result: the most common word used by William Shakespeare is the word *the*, used more than 27K times. In this piece of code, we just want the result outputted; therefore, we launch the job with the `--quiet` option:

```
In:!python MrJob_job2.py --quiet ../datasets/shakespeare_all.txt

Out:27801    "the"

In:!python MrJob_job2.py -r hadoop --quiet hdfs:///datasets/
shakespeare_all.txt

Out:27801    "the"
```

YARN

With Hadoop 2 (the current branch as of 2016),a layer has been introduced on top of HDFS that allows multiple applications to run, for example, MapReduce is one of them (targeting batch processing). The name of this layer is **Yet Another Resource Negotiator** (**YARN**) and its goal is to manage the resource management in the cluster.

YARN follows the paradigm of master/slave and is composed of two services: Resource Manager and Node Manager.

The Resource Manager is the master and is responsible for two things: scheduling (allocating resources) and application management (handling job submission and tracking their status). Each Node Manager, the slaves of the architecture, is the per-worker framework running the tasks and reporting to the Resource Manager.

The YARN layer introduced with Hadoop 2 ensures the following:

- Multitenancy, that is, having multiple engines to use Hadoop
- Better cluster utilization as the allocation of the tasks is dynamic and schedulable
- Better scalability; YARN does not provide a processing algorithm, it's just a resource manager of the cluster
- Compatibility with MapReduce (the higher layer in Hadoop 1)

Spark

Apache Spark is an evolution of Hadoop and has become very popular in the last few years. Contrarily to Hadoop and its Java and batch-focused design, Spark is able to produce iterative algorithms in a fast and easy way. Furthermore, it has a very rich suite of APIs for multiple programming languages and natively supports many different types of data processing (machine learning, streaming, graph analysis, SQL, and so on).

Apache Spark is a cluster framework designed for quick and general-purpose processing of big data. One of the improvements in speed is given by the fact that data, after every job, is kept in-memory and not stored on the filesystem (unless you want to) as would have happened with Hadoop, MapReduce, and HDFS. This thing makes iterative jobs (such as the clustering K-means algorithm) faster and faster as the latency and bandwidth provided by the memory are more performing than the physical disk. Clusters running Spark, therefore, need a high amount of RAM memory for each node.

Although Spark has been developed in Scala (which runs on the JVM, like Java), it has APIs for multiple programming languages, including Java, Scala, Python, and R. In this book, we will focus on Python.

Spark can operate in two different ways:

- Standalone mode: It runs on your local machine. In this case, the maximum parallelization is the number of cores of the local machine and the amount of memory available is exactly the same as the local one.
- Cluster mode: It runs on a cluster of multiple nodes, using a cluster manager such as YARN. In this case, the maximum parallelization is the number of cores across all the nodes composing the cluster and the amount of memory is the sum of the amount of memory of each node.

pySpark

In order to use the Spark functionalities (or pySpark, containing the Python APIs of Spark), we need to instantiate a special object named SparkContext. It tells Spark how to access the cluster and contains some application-specific parameters. In the IPython Notebook provided in the virtual machine, this variable is already available and named `sc` (it's the default option when an IPython Notebook is started); let's now see what it contains.

First, open a new IPython Notebook; when it's ready to be used, type the following in the first cell:

```
In:sc._conf.getAll()

Out:[(u'spark.rdd.compress', u'True'),
 (u'spark.master', u'yarn-client'),
 (u'spark.serializer.objectStreamReset', u'100'),
 (u'spark.yarn.isPython', u'true'),
 (u'spark.submit.deployMode', u'client'),
 (u'spark.executor.cores', u'2'),
 (u'spark.app.name', u'PySparkShell')]
```

It contains multiple information: the most important is the `spark.master`, in this case set as a client in YARN, `spark.executor.cores` set to two as the number of CPUs of the virtual machine, and `spark.app.name`, the name of the application. The name of the app is particularly useful when the (YARN) cluster is shared; going to `ht.0.0.1:8088`, it is possible to check the state of the application:

The data model used by Spark is named **Resilient Distributed Dataset** (**RDD**), which is a distributed collection of elements that can be processed in parallel. An RDD can be created from an existing collection (a Python list, for example) or from an external dataset, stored as a file on the local machine, HDFS, or other sources.

Let's now create an RDD containing integers from 0 to 9. To do so, we can use the `parallelize` method provided by the `SparkContext` object:

```
In:numbers = range(10)
numbers_rdd = sc.parallelize(numbers)

numbers_rdd

Out:ParallelCollectionRDD[1] at parallelize at PythonRDD.scala:423
```

As you can see, you can't simply print the RDD content as it's split into multiple partitions (and distributed in the cluster). The default number of partitions is twice the number of CPUs (so, it's four in the provided VM), but it can be set manually using the second argument of the parallelize method.

To print out the data contained in the RDD, you should call the `collect` method. Note that this operation, while run on a cluster, collects all the data on the node; therefore, the node should have enough memory to contain it all:

```
In:numbers_rdd.collect()

Out:[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

To obtain just a partial peek, use the `take` method indicating how many elements you'd want to see. Note that as it's a distributed dataset, it's not guaranteed that elements are in the same order as when we inserted it:

```
In:numbers_rdd.take()

Out:[0, 1, 2, 3]
```

To read a text file, we can use the `textFile` method provided by the Spark Context. It allows reading both HDFS files and local files, and it splits the text on the newline characters; therefore, the first element of the RDD is the first line of the text file (using the `first` method). Note that if you're using a local path, all the nodes composing the cluster should access the same file through the same path:

```
In:sc.textFile("hdfs:///datasets/hadoop_git_readme.txt").first()

Out:u'For the latest information about Hadoop, please visit our
website at:'

In:sc.textFile("file:///home/vagrant/datasets/
hadoop_git_readme.txt").first()

Out:u'For the latest information about Hadoop, please visit our
website at:'
```

To save the content of an RDD on disk, you can use the `saveAsTextFile` method provided by the RDD. Here, you can use multiple destinations; in this example, let's save it in HDFS and then list the content of the output:

```
In:numbers_rdd.saveAsTextFile("hdfs:///tmp/numbers_1_10.txt")

In:!hdfs dfs -ls /tmp/numbers_1_10.txt

Out:Found 5 items
-rw-r--r--   1 vagrant supergroup          0 2016-02-12 14:18 /tmp/
numbers_1_10.txt/_SUCCESS
-rw-r--r--   1 vagrant supergroup          4 2016-02-12 14:18 /tmp/
numbers_1_10.txt/part-00000
-rw-r--r--   1 vagrant supergroup          4 2016-02-12 14:18 /tmp/
numbers_1_10.txt/part-00001
-rw-r--r--   1 vagrant supergroup          4 2016-02-12 14:18 /tmp/
numbers_1_10.txt/part-00002
-rw-r--r--   1 vagrant supergroup          8 2016-02-12 14:18 /tmp/
numbers_1_10.txt/part-00003
```

Spark writes one file for each partition, exactly as MapReduce, writing one file for each reducer. This way speeds up the saving time as each partition is saved independently, but on a 1-node cluster, it makes things harder to read.

Can we take all the partitions to 1 before writing the file or, generically, can we lower the number of partitions in an RDD? The answer is yes, through the `coalesce` method provided by the RDD, passing the number of partitions we'd want to have as an argument. Passing `1` forces the RDD to be in a standalone partition and, when saved, produces just one output file. Note that this happens even when saving on the local filesystem: a file is created for each partition. Mind that doing so on a cluster environment composed by multiple nodes won't ensure that all the nodes see the same output files:

```
In:
numbers_rdd.coalesce(1) \
.saveAsTextFile("hdfs:///tmp/numbers_1_10_one_file.txt")

In : !hdfs dfs -ls /tmp/numbers_1_10_one_file.txt

Out:Found 2 items
-rw-r--r--   1 vagrant supergroup          0 2016-02-12 14:20 /tmp/
numbers_1_10_one_file.txt/_SUCCESS
-rw-r--r--   1 vagrant supergroup         20 2016-02-12 14:20 /tmp/
numbers_1_10_one_file.txt/part-00000

In:!hdfs dfs -cat /tmp/numbers_1_10_one_file.txt/part-00000

Out:0
1
2
3
4
5
6
7
8
9

In:numbers_rdd.saveAsTextFile("file:///tmp/numbers_1_10.txt")

In:!ls /tmp/numbers_1_10.txt

Out:part-00000  part-00001    part-00002  part-00003    _SUCCESS
```

An RDD supports just two types of operations:

- Transformations transform the dataset into a different one. Inputs and outputs of transformations are both RDDs; therefore, it's possible to chain together multiple transformations, approaching a functional style programming. Moreover, transformations are lazy, that is, they don't compute their results straightaway.
- Actions return values from RDDs, such as the sum of the elements and the count, or just collect all the elements. Actions are the trigger to execute the chain of (lazy) transformations as an output is required.

Typical Spark programs are a chain of transformations with an action at the end. By default, all the transformations on the RDD are executed each time you run an action (that is, the intermediate state after each transformer is not saved). However, you can override this behavior using the `persist` method (on the RDD) whenever you want to cache the value of the transformed elements. The `persist` method allows both memory and disk persistency.

In the next example, we will square all the values contained in an RDD and then sum them up; this algorithm can be executed through a mapper (square elements) followed by a reducer (summing up the array). According to Spark, the `map` method is a transformer as it just transforms the data element by element; `reduce` is an action as it creates a value out of all the elements together.

Let's approach this problem step by step to see the multiple ways in which we can operate. First, start with the mapping: we first define a function that returns the square of the input argument, then we pass this function to the `map` method in the RDD, and finally we collect the elements in the RDD:

```
In:
def sq(x):
    return x**2

numbers_rdd.map(sq).collect()

Out:[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Although the output is correct, the `sq` function is taking a lot of space; we can rewrite the transformation more concisely, thanks to Python's lambda expression, in this way:

```
In:numbers_rdd.map(lambda x: x**2).collect()

Out:[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Remember: why did we need to call collect to print the values in the transformed RDD? This is because the `map` method will not spring to action, but will be just lazily evaluated. The `reduce` method, on the other hand, is an action; therefore, adding the reduce step to the previous RDD should output a value. As for map, reduce takes as an argument a function that should have two arguments (left value and right value) and should return a value. Even in this case, it can be a verbose function defined with `def` or a `lambda` function:

```
In:numbers_rdd.map(lambda x: x**2).reduce(lambda a,b: a+b)

Out:285
```

To make it even simpler, we can use the sum action instead of the reducer:

```
In:numbers_rdd.map(lambda x: x**2).sum()

Out:285
```

So far, we've shown a very simple example of pySpark. Think about what's going on under the hood: the dataset is first loaded and partitioned across the cluster, then the mapping operation is run on the distributed environment, and then all the partitions are collapsed together to generate the result (sum or reduce), which is finally printed on the IPython Notebook. A huge task, yet made super simple by pySpark.

Let's now advance one step and introduce the key-value pairs; although RDDs can contain any kind of object (we've seen integers and lines of text so far), a few operations can be made when the elements are tuples composed by two elements: key and value.

To show an example, let's now first group the numbers in the RDD in odds and evens and then compute the sum of the two groups separately. As for the MapReduce model, it would be nice to map each number with a key (`odd` or `even`) and then, for each key, reduce using a sum operation.

We can start with the map operation: let's first create a function that tags the numbers, outputting `even` if the argument number is even, `odd` otherwise. Then, create a key-value mapping that creates a key-value pair for each number, where the key is the tag and the value is the number itself:

```
In:
def tag(x):
    return "even" if x%2==0 else "odd"


numbers_rdd.map(lambda x: (tag(x), x) ).collect()

Out:[('even', 0),
 ('odd', 1),
 ('even', 2),
 ('odd', 3),
 ('even', 4),
 ('odd', 5),
 ('even', 6),
 ('odd', 7),
 ('even', 8),
 ('odd', 9)]
```

To reduce each key separately, we can now use the `reduceByKey` method (which is not a Spark action). As an argument, we should pass the function that we should apply to all the values of each key; in this case, we will sum up all of them. Finally, we should call the collect method to print the results:

```
In:
numbers_rdd.map(lambda x: (tag(x), x) ) \
.reduceByKey(lambda a,b: a+b).collect()

Out:[('even', 20), ('odd', 25)]
```

Now, let's list some of the most important methods available in Spark; it's not an exhaustive guide, but just includes the most used ones.

We start with transformations; they can be applied to an RDD and they produce an RDD:

- `map(function)`: This returns an RDD formed by passing each element through the function.
- `flatMap(function)`: This returns an RDD formed by flattening the output of the function for each element of the input RDD. It's used when each value at the input can be mapped to 0 or more output elements.

  For example, to count the number of times that each word appears in a text, we should map each word to a key-value pair (the word would be the key, `1` the value), producing more than one key-value element for each input line of text in this way:
- `filter(function)`: This returns a dataset composed by all the values where the function returns true.
- `sample(withReplacement, fraction, seed)`: This bootstraps the RDD, allowing you to create a sampled RDD (with or without replacement) whose length is a fraction of the input one.
- `distinct()`: This returns an RDD containing distinct elements of the input RDD.
- `coalesce(numPartitions)`: This decreases the number of partitions in the RDD.
- `repartition(numPartitions)`: This changes the number of partitions in the RDD. This methods always shuffles all the data over the network.
- `groupByKey()`: This creates an RDD where, for each key, the value is a sequence of values that have that key in the input dataset.
- `reduceByKey(function)`: This aggregates the input RDD by key and then applies the reduce function to the values of each group.
- `sortByKey(ascending)`: This sorts the elements in the RDD by key in ascending or descending order.
- `union(otherRDD)`: This merges two RDDs together.
- `intersection(otherRDD)`: This returns an RDD composed by just the values appearing both in the input and argument RDD.
- `join(otherRDD)`: This returns a dataset where the key-value inputs are joined (on the key) to the argument RDD.

  Similar to the join function in SQL, there are available these methods as well: `cartesian`, `leftOuterJoin`, `rightOuterJoin`, and `fullOuterJoin`.

Now, let's overview what are the most popular actions available in pySpark. Note that actions trigger the processing of the RDD through all the transformers in the chain:

- `reduce(function)`: This aggregates the elements of the RDD producing an output value
- `count()`: This returns the count of the elements in the RDD
- `countByKey()`: This returns a Python dictionary, where each key is associated with the number of elements in the RDD with that key
- `collect()`: This returns all the elements in the transformed RDD locally
- `first()`: This returns the first value of the RDD
- `take(N)`: This returns the first *N* values in the RDD

- `takeSample(withReplacement, N, seed)`: This returns a bootstrap of *N* elements in the RDD with or without replacement, eventually using the random seed provided as argument
- `takeOrdered(N, ordering)`: This returns the top *N* element in the RDD after having sorted it by value (ascending or descending)
- `saveAsTextFile(path)`: This saves the RDD as a set of text files in the specified directory

There are also a few methods that are neither transformers nor actions:

- `cache()`: This caches the elements of the RDD; therefore, future computations based on the same RDD can reuse this as a starting point
- `persist(storage)`: This is the same as cache, but you can specify where to store the elements of RDD (memory, disk, or both)
- `unpersist()`: This undoes the persist or cache operation

Let's now try to replicate the examples that we've seen in the section about MapReduce with Hadoop. With Spark, the algorithm should be as follows:

1. The input file is read and parallelized on an RDD. This operation can be done with the `textFile` method provided by the Spark Context.
2. For each line of the input file, three key-value pairs are returned: one containing the number of chars, one the number of words, and the last the number of lines. In Spark, this is a flatMap operation as three outputs are generated for each input line.
3. For each key, we sum up all the values. This can be done with the `reduceByKey` method.
4. Finally, results are collected. In this case, we can use the `collectAsMap` method that collects the key-value pairs in the RDD and returns a Python dictionary. Note that this is an action; therefore, the RDD chain is executed and a result is returned.

```
In:
def emit_feats(line):
    return [("chars", len(line)), \
            ("words", len(line.split())), \
            ("lines", 1)]

print (sc.textFile("/datasets/hadoop_git_readme.txt")
 .flatMap(emit_feats)
 .reduceByKey(lambda a,b: a+b)
 .collectAsMap())

Out:{'chars': 1335, 'lines': 31, 'words': 179}
```

We can immediately note the enormous speed of this method compared to the MapReduce implementation. This is because all of the dataset is stored in-memory and not in HDFS. Secondly, this is a pure Python implementation and we don't need to call external command lines or libraries—pySpark is self-contained.

Let's now work on the example on the larger file, containing the Shakespeare texts, to extract the most popular word. In the Hadoop MapReduce implementation, it takes two map-reduce steps and therefore four write/read on HDFS. In pySpark, we can do all this in an RDD:

1. The input file is read and parallelized on an RDD with the `textFile` method.
2. For each line, all the words are extracted. For this operation, we can use the flatMap method and a regular expression.
3. Each word in the text (that is, each element of the RDD) is now mapped to a key-value pair: the key is the lowercased word and the value is always `1`. This is a map operation.
4. With a `reduceByKey` call, we count how many times each word (key) appears in the text (RDD). The output is key-value pairs, where the key is a word and value is the number of times the word appears in the text.
5. We flip keys and values, creating a new RDD. This is a map operation.
6. We sort the RDD in descending order and extract (take) the first element. This is an action and can be done in one operation with the `takeOrdered` method.

```
In:import re
WORD_RE = re.compile(r"[\w']+")

print (sc.textFile("/datasets/shakespeare_all.txt")
 .flatMap(lambda line: WORD_RE.findall(line))
 .map(lambda word: (word.lower(), 1))
 .reduceByKey(lambda a,b: a+b)
 .map(lambda (k,v): (v,k))
 .takeOrdered(1, key = lambda x: -x[0]))

Out:[(27801, u'the')]
```

The results are the same that we had using Hadoop and MapReduce, but in this case, the computation takes far less time. We can actually further improve the solution, collapsing the second and third steps together (flatMap-ing a key-value pair for each word, where the key is the lowercased word and value is the number of occurrences) and the fifth and sixth steps together (taking the first element and ordering the elements in the RDD by their value, that is, the second element of the pair):

```
In:
print (sc.textFile("/datasets/shakespeare_all.txt")
 .flatMap(lambda line: [(word.lower(), 1) for word in
WORD_RE.findall(line)])
 .reduceByKey(lambda a,b: a+b)
 .takeOrdered(1, key = lambda x: -x[1]))

Out:[(u'the', 27801)]
```

To check the state of the processing, you can use the Spark UI: it's a graphical interface that shows the jobs run by Spark step-by-step. To access the UI, you should first figure out what's the name of the pySpark IPython application, searching in the bash shell where you've launched the notebook by its name (typically, it is in the form `application_<number>_<number>`), and then point your

browser to the page: `http://localhost:8088/proxy/application_<number>_<number>`

The result is similar to the one in the following image. It contains all the jobs run in spark (as IPython Notebook cells), and you can also visualize the execution plan as a **directed acyclic graph** (**DAG**):

localhost:8088/proxy/application_1455474673628_0002/stages/

Home | Chapter_9_Spark | PySparkShell - Stages for All Jobs

Spark 1.6.0 Jobs Stages Storage Environment Executors PySparkShell application UI

Stages for All Jobs

Completed Stages: 21

Completed Stages (21)

Stage Id	Description		Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
20	takeOrdered at <ipython-input-23-26fc3c5af2ef>:4	+details	2016/02/14 18:38:06	0.1 s	2/2			682.0 KB	
19	reduceByKey at <ipython-input-23-26fc3c5af2ef>:3	+details	2016/02/14 18:38:05	1 s	2/2	63.9 KB			682.0 KB
18	takeOrdered at <ipython-input-22-67b60bde7b70>:9	+details	2016/02/14 18:38:05	0.2 s	2/2			682.0 KB	
17	reduceByKey at <ipython-input-22-67b60bde7b70>:7	+details	2016/02/14 18:38:03	2 s	2/2	63.9 KB			682.0 KB
16	collectAsMap at <ipython-input-21-6e9dbbe6babf>:6	+details	2016/02/14 18:38:02	0.2 s	2/2			382.0 B	
15	reduceByKey at <ipython-input-21-6e9dbbe6babf>:6	+details	2016/02/14 18:38:02	0.5 s	2/2	683.0 B			382.0 B
14	collect at <ipython-input-20-78cf20e84376>:1	+details	2016/02/14 18:38:01	0.3 s	4/4			848.0 B	
13	reduceByKey at <ipython-input-20-78cf20e84376>:1	+details	2016/02/14 18:38:01	0.3 s	4/4				900.0 B
12	collect at <ipython-input-19-7054b6862d4a>:5	+details	2016/02/14 18:38:01	0.2 s	4/4				
11	sum at <ipython-input-18-de392cf955f9>:1	+details	2016/02/14 18:38:00	0.2 s	4/4				
10	reduce at <ipython-input-17-9c3809c99714>:1	+details	2016/02/14 18:38:00	0.1 s	4/4				
9	collect at <ipython-input-16-9e6ba29fb009>:1	+details	2016/02/14 18:38:00	0.2 s	4/4				

Summary

In this chapter, we've introduced some primitives to be able to run distributed jobs on a cluster composed by multiple nodes. We've seen the Hadoop framework and all its components, features, and limitations, and then we illustrated the Spark framework.

In the next chapter, we will dig deep in to Spark, showing how it's possible to do data science in a distributed environment.

Chapter 9. Practical Machine Learning with Spark

In the previous chapter, we saw the main functionalities of data processing with Spark. In this chapter, we will focus on data science with Spark on a real data problem. During the chapter, you will learn the following topics:

- How to share variables across a cluster's nodes
- How to create DataFrames from structured (CSV) and semi-structured (JSON) files, save them on disk, and load them
- How to use SQL-like syntax to select, filter, join, group, and aggregate datasets, thus making the preprocessing extremely easy
- How to handle missing data in the dataset
- Which algorithms are available out of the box in Spark for feature engineering and how to use them in a real case scenario
- Which learners are available and how to measure their performance in a distributed environment
- How to run cross-validation for hyperparameter optimization in a cluster

Setting up the VM for this chapter

As machine learning needs a lot of computational power, in order to save some resources (especially memory) we will use the Spark environment not backed by YARN in this chapter. This mode of operation is named standalone and creates a Spark node without cluster functionalities; all the processing will be on the driver machine and won't be shared. Don't worry; the code that we will see in this chapter will work in a cluster environment as well.

In order to operate this way, perform the following steps:

1. Turn on the virtual machine using the `vagrant up` command.
2. Access the virtual machine when it's ready, with `vagrant ssh`.
3. Launch Spark standalone mode with the IPython Notebook from inside the virtual machine with `./start_jupyter.sh`.
4. Open a browser pointing to `http://localhost:8888`.

To turn it off, use the *Ctrl* + *C* keys to exit the IPython Notebook and `vagrant halt` to turn off the virtual machine.

Note

Note that, even in this configuration, you can access the Spark UI (when at least an IPython Notebook is running) at the following URL:

`http://localhost:4040`

Sharing variables across cluster nodes

When we're working on a distributed environment, sometimes it is required to share information across nodes so that all the nodes can operate using consistent variables. Spark handles this case by providing two kinds of variables: read-only and write-only variables. By not ensuring that a shared variable is both readable and writable anymore, it also drops the consistency requirement, letting the hard work of managing this situation fall on the developer's shoulders. Usually, a solution is quickly reached as Spark is really flexible and adaptive.

Broadcast read-only variables

Broadcast variables are variables shared by the driver node, that is, the node running the IPython Notebook in our configuration, with all the nodes in the cluster. It's a read-only variable as the variable is broadcast by one node and never read back if another node changes it.

Let's now see how it works on a simple example: we want to one-hot encode a dataset containing just gender information as a string. Precisely, the dummy dataset contains just a feature that can be male *M*, female *F*, or unknown *U* (if the information is missing). Specifically, we want all the nodes to use a defined one-hot encoding, as listed in the following dictionary:

```
In:one_hot_encoding = {"M": (1, 0, 0),
                       "F": (0, 1, 0),
                       "U": (0, 0, 1)
                      }
```

Let's now try doing it step by step.

The easiest solution (it's not working though) is to parallelize the dummy dataset (or read it from the disk) and then use the map method on the RDD with a lambda function to map a gender to its encoded tuple:

```
In:(sc.parallelize(["M", "F", "U", "F", "M", "U"])
   .map(lambda x: one_hot_encoding[x])
   .collect())

Out:
[(1, 0, 0), (0, 1, 0), (0, 0, 1), (0, 1, 0), (1, 0, 0), (0, 0, 1)]
```

This solution works locally, but it won't operate on a real distributed environment as all the nodes don't have the `one_hot_encoding` variable available in their workspace. A quick workaround is to include the Python dictionary in the mapped function (that's distributed) as we manage to do here:

```
In:
def map_ohe(x):
    ohe = {"M": (1, 0, 0),
           "F": (0, 1, 0),
           "U": (0, 0, 1)
```

```
        }
    return ohe[x]

sc.parallelize(["M", "F", "U", "F", "M", "U"]).map(map_ohe).collect()

Out:
[(1, 0, 0), (0, 1, 0), (0, 0, 1), (0, 1, 0), (1, 0, 0), (0, 0, 1)]
here are you I love you hello hi is that email with all leave formal
minutes very worrying A hey
```

Such a solution works both locally and on the server, but it's not very nice: we mixed data and process, making the mapping function not reusable. It would be better if the mapping function refers to a broadcasted variable so that it can be used with whatsoever mapping we need to one-hot encode the dataset.

For this, we first broadcast the Python dictionary (calling the `broadcast` method provided by the Spark context, `sc`) inside the mapped function; using its `.value` property, we can now have access to it. After doing this, we have a generic map function that can work on any one-hot map dictionary:

```
In:bcast_map = sc.broadcast(one_hot_encoding)

def bcast_map_ohe(x, shared_ohe):
    return shared_ohe[x]

(sc.parallelize(["M", "F", "U", "F", "M", "U"])
 .map(lambda x: bcast_map_ohe(x, bcast_map.value))
 .collect())

Out:
[(1, 0, 0), (0, 1, 0), (0, 0, 1), (0, 1, 0), (1, 0, 0), (0, 0, 1)]
```

Think about the broadcasted variable as a file written in HDFS. Then, when a generic node wants to access it, it just needs the HDFS path (which is passed as an argument of the map method) and you're sure that all of them will read the same thing, using the same path. Of course, Spark doesn't use HDFS, but an in-memory variation of it.

Note

Broadcasted variables are saved in-memory in all the nodes composing a cluster; therefore, they never share a large amount of data that can fill them and make the following processing impossible.

To remove a broadcasted variable, use the `unpersist` method on the broadcasted variable. This operation will free up the memory of that variable on all the nodes:

```
In:bcast_map.unpersist()
```

Accumulators write-only variables

The other variables that can be shared in a Spark cluster are accumulators. Accumulators are write-only variables that can be added together and are used typically to implement sums or counters. Just the driver node, the one that is running the IPython Notebook, can read its value; all the other nodes can't.

Let's see how it works using an example: we want to process a text file and understand how many lines are empty while processing it. Of course, we can do this by scanning the dataset twice (using two Spark jobs): the first one counting the empty lines, and the second time doing the real processing, but this solution is not very effective.

In the first, ineffective solution—extracting the number of empty lines using two standalone Spark jobs—we can read the text file, filter the empty lines, and count them, as shown here:

```
In:print "The number of empty lines is:"

(sc.textFile('file:///home/vagrant/datasets/hadoop_git_readme.txt')
   .filter(lambda line: len(line) == 0)
   .count())

Out:The number of empty lines is:
6
```

The second solution is instead more effective (and more complex). We instantiate an accumulator variable (with the initial value of `0`) and we add `1` for each empty line that we find while processing each line of the input file (with a map). At the same time, we can do some processing on each line; in the following piece of code, for example, we simply return `1` for each line, counting all the lines in the file in this way.

At the end of the processing, we will have two pieces of information: the first is the number of lines, from the result of the `count()` action on the transformed RDD, and the second is the number of empty lines contained in the `value` property of the accumulator. Remember, both of these are available after having scanned the dataset once:

```
In:accum = sc.accumulator(0)

def split_line(line):
    if len(line) == 0:
        accum.add(1)
    return 1

tot_lines = (
    sc.textFile('file:///home/vagrant/datasets/
hadoop_git_readme.txt')
      .map(split_line)
      .count())
```

```
empty_lines = accum.value

print "In the file there are %d lines" % tot_lines
print "And %d lines are empty" % empty_lines

Out:In the file there are 31 lines
And 6 lines are empty
```

Natively, Spark supports accumulators of numeric types, and the default operation is a sum. With a bit more coding, we can turn it into something more complex.

Broadcast and accumulators together – an example

Although broadcast and accumulators are simple and very limited variables (one is read-only, the other one is write-only), they can be actively used to create very complex operations. For example, let's try to apply different machine learning algorithms on the Iris dataset in a distributed environment. We will build a Spark job in the following way:

1. The dataset is read and broadcasted to all the nodes (as it's small enough to fit in-memory).
2. Each node will use a different classifier on the dataset and return the classifier name and its accuracy score on the full dataset. Note that, to keep things easy in this simple example, we won't do any preprocessing, train/test splitting, or hyperparameter optimization.
3. If the classifiers raise any exception, the string representation of the error along with the classifier name should be stored in an accumulator.
4. The final output should contain a list of the classifiers that performed the classification task without errors and their accuracy score.

As the first step, we load the Iris dataset and broadcast it to all the nodes in the cluster:

```
In:from sklearn.datasets import load_iris

bcast_dataset = sc.broadcast(load_iris())
```

Now, let's create a custom accumulator. It will contain a list of tuples to store the classifier name and the exception it experienced as a string. The custom accumulator is derived by the `AccumulatorParam` class and should contain at least two methods: `zero` (which is called when it's initialized) and `addInPlace` (which is called when the add method is called on the accumulator).

The easiest way to do this is shown in the following code, followed by its initialization as an empty list. Mind that the additive operation is a bit tricky: we need to combine two elements, a tuple, and a list, but we don't know which element is the list and which is the tuple; therefore, we first ensure that both elements are lists and then we can proceed to concatenate them in an easy way (with the + operator):

```
In:from pyspark import AccumulatorParam

class ErrorAccumulator(AccumulatorParam):
    def zero(self, initialList):
        return initialList
```

```
    def addInPlace(self, v1, v2):
        if not isinstance(v1, list):
            v1 = [v1]
        if not isinstance(v2, list):
            v2 = [v2]
        return v1 + v2

errAccum = sc.accumulator([], ErrorAccumulator())
```

Now, let's define the mapping function: each node should train, test, and evaluate a classifier on the broadcasted Iris dataset. As an argument, the function will receive the classifier object and should return a tuple containing the classifier name and its accuracy score contained in a list.

If any exception is raised by doing so, the classifier name and exception as a string are added to the accumulator, and it's returned as an empty list:

```
In:
def apply_classifier(clf, dataset):

    clf_name = clf.__class__.__name__
    X = dataset.value.data
    y = dataset.value.target

    try:
        from sklearn.metrics import accuracy_score

        clf.fit(X, y)
        y_pred = clf.predict(X)
        acc = accuracy_score(y, y_pred)

        return [(clf_name, acc)]

    except Exception as e:
        errAccum.add((clf_name, str(e)))
        return []
```

Finally, we have arrived at the core of the job. We're now instantiating a few objects from Scikit-learn (some of them are not classifiers, in order to test the accumulator). We will transform them into an RDD and apply the map function that we created in the previous cell. As the returned value is a list, we can use `flatMap` to collect just the outputs of the mappers that didn't get caught in any exception:

```
In:from sklearn.linear_model import SGDClassifier
from sklearn.dummy import DummyClassifier
from sklearn.decomposition import PCA
from sklearn.manifold import MDS
```

```
classifiers = [DummyClassifier('most_frequent'),
               SGDClassifier(),
               PCA(),
               MDS()]

(sc.parallelize(classifiers)
     .flatMap(lambda x: apply_classifier(x, bcast_dataset))
     .collect())

Out:[('DummyClassifier', 0.33333333333333331),
 ('SGDClassifier', 0.66666666666666663)]
```

As expected, only the *real* classifiers are contained in the output. Let's now see which classifiers generated an error. Unsurprisingly, here we spot the two missing ones from the preceding output:

```
In:print "The errors are:"
errAccum.value

Out:The errors are:
 [('PCA', "'PCA' object has no attribute 'predict'"),
 ('MDS', "Proximity must be 'precomputed' or 'euclidean'. Got
euclidean instead")]
```

As a final step, let's clean up the broadcasted dataset:

```
In:bcast_dataset.unpersist()
```

Remember that in this example, we've used a small dataset that could be broadcasted. In real-world big data problems, you'll need to load the dataset from HDFS, broadcasting the HDFS path.

Data preprocessing in Spark

So far, we've seen how to load text data from the local filesystem and HDFS. Text files can contain either unstructured data (like a text document) or structured data (like a CSV file). As for semi-structured data, just like files containing JSON objects, Spark has special routines able to transform a file into a DataFrame, similar to the DataFrame in R and Python pandas. DataFrames are very similar to RDBMS tables, where a schema is set.

JSON files and Spark DataFrames

In order to import JSON-compliant files, we should first create a SQL context, creating a `SQLContext` object from the local Spark Context:

```
In:from pyspark.sql import SQLContext
sqlContext = SQLContext(sc)
```

Now, let's see the content of a small JSON file (it's provided in the Vagrant virtual machine). It's a JSON representation of a table with six rows and three columns, where some attributes are missing (such as the `gender` attribute for the user with `user_id=0`):

```
In:!cat /home/vagrant/datasets/users.json

Out:{"user_id":0, "balance": 10.0}
{"user_id":1, "gender":"M", "balance": 1.0}
{"user_id":2, "gender":"F", "balance": -0.5}
{"user_id":3, "gender":"F", "balance": 0.0}
{"user_id":4, "balance": 5.0}
{"user_id":5, "gender":"M", "balance": 3.0}
```

Using the `read.json` method provided by `sqlContext`, we already have the table well formatted and with all the right column names in a variable. The output variable is typed as Spark DataFrame. To show the variable in a nice, formatted table, use its `show` method:

```
In:
df = sqlContext.read \
.json("file:///home/vagrant/datasets/users.json")
df.show()

Out:
+-------+------+-------+
|balance|gender|user_id|
+-------+------+-------+
|   10.0|  null|      0|
|    1.0|     M|      1|
|   -0.5|     F|      2|
|    0.0|     F|      3|
```

```
|    5.0|  null|      4|
|    3.0|     M|      5|
+-------+------+-------+
```

Additionally, we can investigate the schema of the DataFrame using the `printSchema` method. We realize that, while reading the JSON file, each column type has been inferred by the data (in the example, the `user_id` column contains long integers, the gender column is composed by strings, and the balance is a double floating point):

```
In:df.printSchema()

Out:root
 |-- balance: double (nullable = true)
 |-- gender: string (nullable = true)
 |-- user_id: long (nullable = true)
```

Exactly like a table in an RDBMS, we can slide and dice the data in the DataFrame, making selections of columns and filtering the data by attributes. In this example, we want to print the balance, gender, and `user_id` of the users whose gender is not missing and have a balance strictly greater than zero. For this, we can use the `filter` and `select` methods:

```
In:(df.filter(df['gender'] != 'null')
.filter(df['balance'] > 0)
   .select(['balance', 'gender', 'user_id'])
   .show())

Out:
+-------+------+-------+
|balance|gender|user_id|
+-------+------+-------+
|    1.0|     M|      1|
|    3.0|     M|      5|
+-------+------+-------+
```

We can also rewrite each piece of the preceding job in a SQL-like language. In fact, `filter` and `select` methods can accept SQL-formatted strings:

```
In:(df.filter('gender is not null')
   .filter('balance > 0').select("*").show())

Out:
+-------+------+-------+
|balance|gender|user_id|
+-------+------+-------+
|    1.0|     M|      1|
|    3.0|     M|      5|
+-------+------+-------+
```

We can also use just one call to the `filter` method:

```
In:df.filter('gender is not null and balance > 0').show()

Out:
+-------+------+-------+
|balance|gender|user_id|
+-------+------+-------+
|    1.0|     M|      1|
|    3.0|     M|      5|
+-------+------+-------+
```

Dealing with missing data

A common problem of data preprocessing is to handle missing data. Spark DataFrames, similar to pandas DataFrames, offer a wide range of operations that you can do on them. For example, the easiest option to have a dataset composed by complete rows only is to discard rows containing missing information. For this, in a Spark DataFrame, we first have to access the `na` attribute of the DataFrame and then call the `drop` method. The resulting table will contain only the complete rows:

```
In:df.na.drop().show()

Out:
+-------+------+-------+
|balance|gender|user_id|
+-------+------+-------+
|    1.0|     M|      1|
|   -0.5|     F|      2|
|    0.0|     F|      3|
|    3.0|     M|      5|
+-------+------+-------+
```

If such an operation is removing too many rows, we can always decide what columns should be accounted for the removal of the row (as the augmented subset of the `drop` method):

```
In:df.na.drop(subset=["gender"]).show()

Out:
+-------+------+-------+
|balance|gender|user_id|
+-------+------+-------+
|    1.0|     M|      1|
|   -0.5|     F|      2|
|    0.0|     F|      3|
|    3.0|     M|      5|
+-------+------+-------+
```

Also, if you want to set default values for each column instead of removing the line data, you can use the `fill` method, passing a dictionary composed by the column name (as the dictionary key) and the default value to substitute missing data in that column (as the value of the key in the dictionary).

As an example, if you want to ensure that the variable balance, where missing, is set to `0`, and the variable gender, where missing, is set to `U`, you can simply do the following:

```
In:df.na.fill({'gender': "U", 'balance': 0.0}).show()

Out:
+-------+------+-------+
|balance|gender|user_id|
+-------+------+-------+
|   10.0|     U|      0|
|    1.0|     M|      1|
|   -0.5|     F|      2|
|    0.0|     F|      3|
|    5.0|     U|      4|
|    3.0|     M|      5|
+-------+------+-------+
```

Grouping and creating tables in-memory

To have a function applied on a group of rows (exactly as in the case of SQL `GROUP BY`), you can use two similar methods. In the following example, we want to compute the average balance per gender:

```
In:(df.na.fill({'gender': "U", 'balance': 0.0})
   .groupBy("gender").avg('balance').show())

Out:
+------+------------+
|gender|avg(balance)|
+------+------------+
|     F|       -0.25|
|     M|         2.0|
|     U|         7.5|
+------+------------+
```

So far, we've worked with DataFrames but, as you've seen, the distance between DataFrame methods and SQL commands is minimal. Actually, using Spark, it is possible to register the DataFrame as a SQL table to fully enjoy the power of SQL. The table is saved in-memory and distributed in a way similar to an RDD.

To register the table, we need to provide a name, which will be used in future SQL commands. In this case, we decide to name it `users`:

```
In:df.registerTempTable("users")
```

By calling the `sql` method provided by the Spark `sql` context, we can run any SQL-compliant table:

```
In:sqlContext.sql("""
    SELECT gender, AVG(balance)
    FROM users
    WHERE gender IS NOT NULL
    GROUP BY gender""").show()

Out:
+------+-----+
|gender|  _c1|
+------+-----+
|     F|-0.25|
|     M|  2.0|
+------+-----+
```

Not surprisingly, the table outputted by the command (as well as the `users` table itself) is of the Spark DataFrame type:

```
In:type(sqlContext.table("users"))

Out:pyspark.sql.dataframe.DataFrame
```

DataFrames, tables, and RDDs are intimately connected, and RDD methods can be used on a DataFrame. Remember that each row of the DataFrame is an element of the RDD. Let's see this in detail and first collect the whole table:

```
In:sqlContext.table("users").collect()
Out:[Row(balance=10.0, gender=None, user_id=0),
 Row(balance=1.0, gender=u'M', user_id=1),
 Row(balance=-0.5, gender=u'F', user_id=2),
 Row(balance=0.0, gender=u'F', user_id=3),
 Row(balance=5.0, gender=None, user_id=4),
 Row(balance=3.0, gender=u'M', user_id=5)]

In:
a_row = sqlContext.sql("SELECT * FROM users").first()
a_row

Out:Row(balance=10.0, gender=None, user_id=0)
```

The output is a list of `Row` objects (they look like Python's `namedtuple`). Let's dig deeper into it: `Row` contains multiple attributes, and it's possible to access them as a property or dictionary key; that is, to have the balance out from the first row, we can choose between the two following ways:

```
In:print a_row['balance']
print a_row.balance
```

```
Out:10.0
10.0
```

Also, Row can be collected as a Python dictionary using the asDict method of Row. The result contains the property names as a key and property values as dictionary values:

```
In:a_row.asDict()

Out:{'balance': 10.0, 'gender': None, 'user_id': 0}
```

Writing the preprocessed DataFrame or RDD to disk

To write a DataFrame or RDD to disk, we can use the write method. We have a selection of formats; in this case, we will save it as a JSON file on the local machine:

```
In:(df.na.drop().write
   .save("file:///tmp/complete_users.json", format='json'))
```

Checking the output on the local filesystem, we immediately see that something is different from what we expected: this operation creates multiple files (part-r-…).

Each of them contains some rows serialized as JSON objects, and merging them together will create the comprehensive output. As Spark is made to process large and distributed files, the write operation is tuned for that and each node writes part of the full RDD:

```
In:!ls -als /tmp/complete_users.json

Out:total 28
4 drwxrwxr-x 2 vagrant vagrant 4096 Feb 25 22:54 .
4 drwxrwxrwt 9 root    root    4096 Feb 25 22:54 ..
4 -rw-r--r-- 1 vagrant vagrant   83 Feb 25 22:54 part-r-00000-...
4 -rw-rw-r-- 1 vagrant vagrant   12 Feb 25 22:54 .part-r-00000-...
4 -rw-r--r-- 1 vagrant vagrant   82 Feb 25 22:54 part-r-00001-...
4 -rw-rw-r-- 1 vagrant vagrant   12 Feb 25 22:54 .part-r-00001-...
0 -rw-r--r-- 1 vagrant vagrant    0 Feb 25 22:54 _SUCCESS
4 -rw-rw-r-- 1 vagrant vagrant    8 Feb 25 22:54 ._SUCCESS.crc
```

In order to read it back, we don't have to create a standalone file—even multiple pieces are fine in the read operation. A JSON file can also be read in the FROM clause of a SQL query. Let's now try to print the JSON that we've just written on disk without creating an intermediate DataFrame:

```
In:sqlContext.sql(
    "SELECT * FROM json.`file:///tmp/complete_users.json`").show()

Out:
+-------+------+-------+
```

```
|balance|gender|user_id|
+-------+------+-------+
|    1.0|     M|      1|
|   -0.5|     F|      2|
|    0.0|     F|      3|
|    3.0|     M|      5|
+-------+------+-------+
```

Beyond JSON, there is another format that's very popular when dealing with structured big datasets: Parquet format. Parquet is a columnar storage format that's available in the Hadoop ecosystem; it compresses and encodes the data and can work with nested structures: all such qualities make it very efficient.

Saving and loading is very similar to JSON and, even in this case, this operation produces multiple files written to disk:

```
In:df.na.drop().write.save(
    "file:///tmp/complete_users.parquet", format='parquet')

In:!ls -als /tmp/complete_users.parquet/

Out:total 44
4 drwxrwxr-x  2 vagrant vagrant 4096 Feb 25 22:54 .
4 drwxrwxrwt 10 root    root    4096 Feb 25 22:54 ..
4 -rw-r--r--  1 vagrant vagrant  376 Feb 25 22:54 _common_metadata
4 -rw-rw-r--  1 vagrant vagrant   12 Feb 25 22:54 ._common_metadata..
4 -rw-r--r--  1 vagrant vagrant 1082 Feb 25 22:54 _metadata
4 -rw-rw-r--  1 vagrant vagrant   20 Feb 25 22:54 ._metadata.crc
4 -rw-r--r--  1 vagrant vagrant  750 Feb 25 22:54 part-r-00000-...
4 -rw-rw-r--  1 vagrant vagrant   16 Feb 25 22:54 .part-r-00000-...
4 -rw-r--r--  1 vagrant vagrant  746 Feb 25 22:54 part-r-00001-...
4 -rw-rw-r--  1 vagrant vagrant   16 Feb 25 22:54 .part-r-00001-...
0 -rw-r--r--  1 vagrant vagrant    0 Feb 25 22:54 _SUCCESS
4 -rw-rw-r--  1 vagrant vagrant    8 Feb 25 22:54 ._SUCCESS.crc
```

Working with Spark DataFrames

So far, we've described how to load DataFrames from JSON and Parquet files, but not how to create them from an existing RDD. In order to do so, you just need to create one `Row` object for each record in the RDD and call the `createDataFrame` method of the SQL context. Finally, you can register it as a temp table to use the power of the SQL syntax fully:

```
In:from pyspark.sql import Row

rdd_gender = \
    sc.parallelize([Row(short_gender="M", long_gender="Male"),
                    Row(short_gender="F", long_gender="Female")])
```

```
(sqlContext.createDataFrame(rdd_gender)
           .registerTempTable("gender_maps"))

In:sqlContext.table("gender_maps").show()

Out:
+-----------+------------+
|long_gender|short_gender|
+-----------+------------+
|       Male|           M|
|     Female|           F|
+-----------+------------+
```

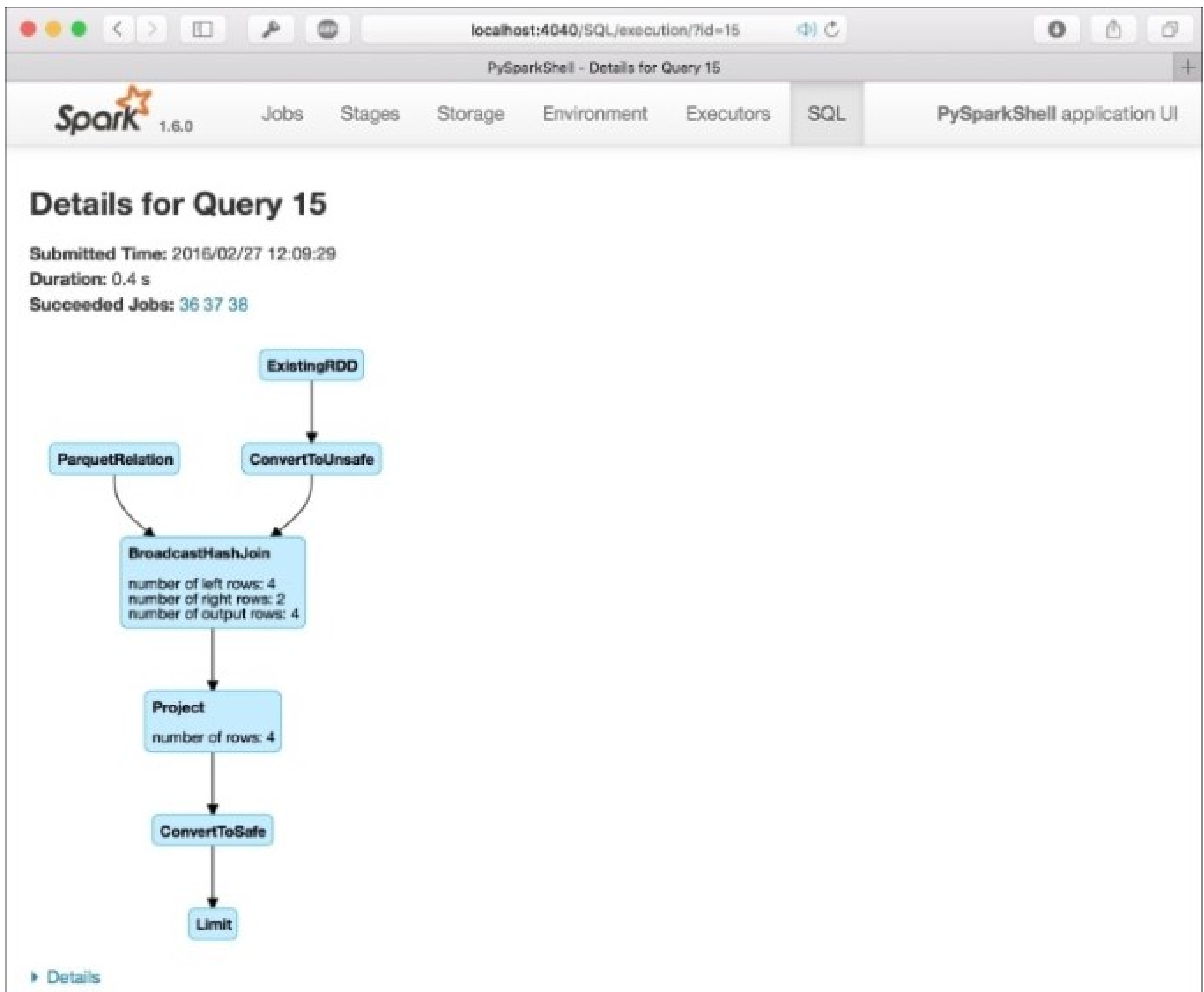
Note

This is also the preferred way to operate with CSV files. First, the file is read with `sc.textFile`; then with the `split` method, the `Row` constructor, and the `createDataFrame` method, the final DataFrame is created.

When you have multiple DataFrames in-memory, or that can be loaded from disk, you can join and use all the operations available in a classic RDBMS. In this example, we can join the DataFrame we've created from the RDD with the users dataset contained in the Parquet file that we've stored. The result is astonishing:

```
In:sqlContext.sql("""
    SELECT balance, long_gender, user_id
    FROM parquet.`file:///tmp/complete_users.parquet`
    JOIN gender_maps ON gender=short_gender""").show()

Out:
+-------+-----------+-------+
|balance|long_gender|user_id|
+-------+-----------+-------+
|    3.0|       Male|      5|
|    1.0|       Male|      1|
|    0.0|     Female|      3|
|   -0.5|     Female|      2|
+-------+-----------+-------+
```

In the web UI, each SQL query is mapped as a virtual **directed acyclic graph** (**DAG**) under the **SQL** tab. This is very nice to keep track of the progress of your job and understand the complexity of the query. While doing the preceding JOIN query, you can clearly see that two branches are entering the same `BroadcastHashJoin` block: the first one is from an RDD and the second one is from a Parquet file. Then, the following block is simply a projetion on the selected columns:



As the tables are in-memory, the last thing to do is to clean up releasing the memory used to keep them. By calling the `tableNames` method, provided by the `sqlContext`, we have the list of all the tables that we currently have in-memory. Then, to free them up, we can use `dropTempTable` with the name of the table as argument. Beyond this point, any further reference to these tables will return an error:

```
In:sqlContext.tableNames()

Out:[u'gender_maps', u'users']

In:
for table in sqlContext.tableNames():
    sqlContext.dropTempTable(table)
```

Since Spark 1.3, DataFrame is the preferred way to operate on a dataset when doing data science operations.

Machine learning with Spark

Here, we arrive at the main task of your job: creating a model to predict one or multiple attributes missing in the dataset. For this, we use some machine learning modeling, and Spark can provide us with a big hand in this context.

MLlib is the Spark machine learning library; although it is built in Scala and Java, its functions are also available in Python. It contains classification, regression, and recommendation learners, some routines for dimensionality reduction and feature selection, and has lots of functionalities for text processing. All of them are able to cope with huge datasets and use the power of all the nodes in the cluster to achieve the goal.

As of now (2016), it's composed of two main packages: `mllib`, which operates on RDDs, and `ml`, which operates on DataFrames. As the latter performs well and the most popular way to represent data in data science, developers have chosen to contribute and improve the `ml` branch, letting the former remain, but without further developments. MLlib seems a complete library at first sight but, after having started using Spark, you will notice that there's neither a statistic nor numerical library in the default package. Here, SciPy and NumPy come to your help, and once again, they're essential for data science!

In this section, we will try to explore the functionalities of the *new* `pyspark.ml` package; as of now, it's still in the early stages compared to the state-of-the-art Scikit-learn library, but it definitely has a lot of potential for the future.

Note

Spark is a high-level, distributed, and complex software that should be used just on big data and with a cluster of multiple nodes; in fact, if the dataset can fit in-memory, it's more convenient to use other libraries such as Scikit-learn or similar, which focus just on the data science side of the problem. Running Spark on a single node on a small dataset can be five times slower than the Scikit-learn-equivalent algorithm.

Spark on the KDD99 dataset

Let's conduct this exploration using a real-world dataset: the KDD99 dataset. The goal of the competition was to create a network intrusion detection system able to recognize which network flow is malicious and which is not. Moreover, many different attacks are in the dataset; the goal is to accurately predict them using the features of the flow of packets contained in the dataset.

As a side node on the dataset, it has been extremely useful to develop great solutions for intrusion detection systems in the first few years after its release. Nowadays, as an outcome of this, all the attacks included in the dataset are very easy to detect and so it's not used in IDS development anymore.

The features are, for example, the protocol (tcp, icmp, and udp), service (http, smtp, and so on), size of the packets, flags active in the protocol, number of attempts to become root, and so on.

Note

More information about the KDD99 challenge and datasets is available at http://kdd.ics.uci.edu/databases/kddcup99/kddcup99.html.

Although this is a classic multiclass classification problem, we will dig into it to show you how to perform this task in Spark. To keep things clean, we will use a new IPython Notebook.

Reading the dataset

First at all, let's download and decompress the dataset. We will be very conservative and use just 10% of the original training dataset (75MB, uncompressed) as all our analysis is run on a small virtual machine. If you want to give it a try, you can uncomment the lines in the following snippet of code and download the full training dataset (750MB uncompressed). We download the training dataset, testing (47MB), and feature names, using bash commands:

```
In:!rm -rf ../datasets/kdd*

# !wget -q -O ../datasets/kddtrain.gz \
# http://kdd.ics.uci.edu/databases/kddcup99/kddcup.data.gz

!wget -q -O ../datasets/kddtrain.gz \
http://kdd.ics.uci.edu/databases/kddcup99/kddcup.data_10_percent.gz

!wget -q -O ../datasets/kddtest.gz \
http://kdd.ics.uci.edu/databases/kddcup99/corrected.gz

!wget -q -O ../datasets/kddnames \
http://kdd.ics.uci.edu/databases/kddcup99/kddcup.names

!gunzip ../datasets/kdd*gz
```

Now, print the first few lines to have an understanding of the format. It is clear that it's a classic CSV without a header, containing a dot at the end of each line. Also, we can see that some fields are numeric but a few of them are textual, and the target variable is contained in the last field:

```
In:!head -3 ../datasets/kddtrain

Out:
0,tcp,http,SF,181,5450,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,8,8,0.00,0.00,
0.00,0.00,1.00,0.00,0.00,9,9,1.00,0.00,0.11,0.00,0.00,0.00,0.00,0.00,
normal.
0,tcp,http,SF,239,486,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,8,8,0.00,0.00,0
.00,0.00,1.00,0.00,0.00,19,19,1.00,0.00,0.05,0.00,0.00,0.00,0.00,0.00
,normal.
0,tcp,http,SF,235,1337,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,8,8,0.00,0.00,
```

```
0.00,0.00,1.00,0.00,0.00,29,29,1.00,0.00,0.03,0.00,0.00,0.00,0.00,0.0
0,normal.
```

To create a DataFrame with named fields, we should first read the header included in the `kddnames` file. The target field will be named simply `target`.

After having read and parsed the file, we print the number of features of our problem (remember that the target variable is not a feature) and their first 10 names:

```
In:
with open('../datasets/kddnames', 'r') as fh:
    header = [line.split(':')[0]
              for line in fh.read().splitlines()][1:]

header.append('target')

print "Num features:", len(header)-1
print "First 10:", header[:10]

Out:Num features: 41
First 10: ['duration', 'protocol_type', 'service', 'flag',
'src_bytes', 'dst_bytes', 'land', 'wrong_fragment', 'urgent', 'hot']
```

Let's now create two separate RDDs—one for the training data and the other for the testing data:

```
In:
train_rdd = sc.textFile('file:///home/vagrant/datasets/kddtrain')
test_rdd = sc.textFile('file:///home/vagrant/datasets/kddtest')
```

Now, we need to parse each line of each file to create a DataFrame. First, we split each line of the CSV file into separate fields, and then we cast each numerical value to a floating point and each text value to a string. Finally, we remove the dot at the end of each line.

As the last step, using the `createDataFrame` method provided by `sqlContext`, we can create two Spark DataFrames with named columns for both training and testing datasets:

```
In:
def line_parser(line):

    def piece_parser(piece):
            if "." in piece or piece.isdigit():
                return float(piece)
            else:
                return piece

    return [piece_parser(piece) for piece in line[:-1].split(',')]
```

```
train_df = sqlContext.createDataFrame(
    train_rdd.map(line_parser), header)

test_df = sqlContext.createDataFrame(
    test_rdd.map(line_parser), header)
```

So far we've written just RDD transformers; let's introduce an action to see how many observations we have in the datasets and, at the same time, check the correctness of the previous code.

```
In:print "Train observations:", train_df.count()
print "Test observations:", test_df.count()

Out:Train observations: 494021
Test observations: 311029
```

Although we're using a tenth of the full KDD99 dataset, we still work on half a million observations. Multiplied by the number of features, 41, we clearly see that we'll be training our classifier on an observation matrix containing more than 20 million values. This is not such a big dataset for Spark (and neither is the full KDD99); developers around the world are already using it on petabytes and billion records. Don't be scared if the numbers seem big: Spark is designed to cope with them!

Now, let's see how it looks on the schema of the DataFrame. Specifically, we want to identify which fields are numeric and which contain strings (note that the result has been truncated for brevity):

```
In:train_df.printSchema()

Out:root
 |-- duration: double (nullable = true)
 |-- protocol_type: string (nullable = true)
 |-- service: string (nullable = true)
 |-- flag: string (nullable = true)
 |-- src_bytes: double (nullable = true)
 |-- dst_bytes: double (nullable = true)
 |-- land: double (nullable = true)
 |-- wrong_fragment: double (nullable = true)
 |-- urgent: double (nullable = true)
 |-- hot: double (nullable = true)
...
...
...
 |-- target: string (nullable = true)
```

Feature engineering

From a visual analysis, only four fields are strings: `protocol_type`, `service`, `flag`, and `target` (which is the multiclass target label, as expected).

As we will use a tree-based classifier, we want to encode the text of each level to a number for each variable. With Scikit-learn, this operation can be done with a `sklearn.preprocessing.LabelEncoder` object. It's equivalent in Spark is `StringIndexer` of the `pyspark.ml.feature` package.

We need to encode four variables with Spark; then we have to chain four `StringIndexer` objects together in a cascade: each of them will operate on a specific column of the DataFrame, outputting a DataFrame with an additional column (similar to a map operation). The mapping is automatic, ordered by frequency: Spark ranks the count of each level in the selected column, mapping the most popular level to 0, the next to 1, and so on. Note that, with this operation, you will traverse the dataset once to count the occurrences of each level; if you already know the mapping, it would be more effective to broadcast it and use a `map` operation, as shown at the beginning of this chapter.

Similarly, we could have used a one-hot encoder to generate a numerical observation matrix. In case of a one-hot encoder, we would have had multiple output columns in the DataFrame, one for each level of each categorical feature. For this, Spark offers the `pyspark.ml.feature.OneHotEncoder` class.

Note

More generically, all the classes contained in the `pyspark.ml.feature` package are used to extract, transform, and select features from a DataFrame. All of them *read* some columns and *create* some other columns in the DataFrame.

As of Spark 1.6, the feature operations available in Python are contained in the following exhaustive list (all of them can be found in the `pyspark.ml.feature` package). Names should be intuitive, except for a couple of them that will be explained inline or later in the text:

- For text inputs (ideally):
    - HashingTF and IDF
    - Tokenizer and its regex-based implementation, RegexTokenizer
    - Word2vec
    - StopWordsRemover
    - Ngram
- For categorical features:
    - StringIndexer and it's inverse encoder, IndexToString
    - OneHotEncoder
    - VectorIndexer (out-of-the-box categorical to numerical indexer)
- For other inputs:
    - Binarizer
    - PCA
    - PolynomialExpansion
    - Normalizer, StandardScaler, and MinMaxScaler
    - Bucketizer (buckets the values of a feature)
    - ElementwiseProduct (multiplies columns together)
- Generic:
    - SQLTransformer (implements transformations defined by a SQL statement, referring to DataFrame as a table named `__THIS__`)
    - RFormula (selects columns using an R-style syntax)

- VectorAssembler (creates a feature vector from multiple columns)

Going back to the example, we now want to encode the levels in each categorical variable as discrete numbers. As we've explained, for this, we will use a `StringIndexer` object for each variable. Moreover, we can use an ML Pipeline and set them as stages of it.

Then, to fit all the indexers, you just need to call the `fit` method of the pipeline. Internally, it will fit all the staged objects sequentially. When it's completed the fit operation, a new object is created and we can refer to it as the fitted pipeline. Calling the `transform` method of this new object will sequentially call all the staged elements (which are already fitted), each after the previous one is completed. In this snippet of code, you'll see the pipeline in action. Note that transformers compose the pipeline. Therefore, as no actions are present, nothing is actually executed. In the output DataFrame, you'll note four additional columns named the same as the original categorical ones, but with the `_cat` suffix:

```
In:from pyspark.ml import Pipeline
from pyspark.ml.feature import StringIndexer


cols_categorical = ["protocol_type", "service", "flag","target"]
preproc_stages = []

for col in cols_categorical:
    out_col = col + "_cat"
    preproc_stages.append(
        StringIndexer(
            inputCol=col, outputCol=out_col, handleInvalid="skip"))

pipeline = Pipeline(stages=preproc_stages)
indexer = pipeline.fit(train_df)

train_num_df = indexer.transform(train_df)
test_num_df = indexer.transform(test_df)
```

Let's investigate the pipeline a bit more. Here, we will see the stages in the pipeline: unfit pipeline and fitted pipeline. Note that there's a big difference between Spark and Scikit-learn: in Scikit-learn, `fit` and `transform` are called on the same object, and in Spark, the `fit` method produces a new object (typically, its name is added with a `Model` suffix, just as for `Pipeline` and `PipelineModel`), where you'll be able to call the `transform` method. This difference is derived from closures—a fitted object is easy to distribute across processes and the cluster:

```
In:print pipeline.getStages()
print
print pipeline
print indexer

Out:
[StringIndexer_432c8aca691aaee949b8,
```

```
StringIndexer_4f10bbcde2452dd1b771,
StringIndexer_4aad99dc0a3ff831bea6,
StringIndexer_4b369fea07873fc9c2a3]

Pipeline_48df9eed31c543ba5eba
PipelineModel_46b09251d9e4b117dc8d
```

Let's see how the first observation, that is, the first line in the CSV file, changes after passing through the pipeline. Note that we use an action here, therefore all the stages in the pipeline and in the pipeline model are executed:

```
In:print "First observation, after the 4 StringIndexers:\n"
print train_num_df.first()

Out:First observation, after the 4 StringIndexers:

Row(duration=0.0, protocol_type=u'tcp', service=u'http', flag=u'SF',
src_bytes=181.0, dst_bytes=5450.0, land=0.0, wrong_fragment=0.0,
urgent=0.0, hot=0.0, num_failed_logins=0.0, logged_in=1.0,
num_compromised=0.0, root_shell=0.0, su_attempted=0.0, num_root=0.0,
num_file_creations=0.0, num_shells=0.0, num_access_files=0.0,
num_outbound_cmds=0.0, is_host_login=0.0, is_guest_login=0.0,
count=8.0, srv_count=8.0, serror_rate=0.0, srv_serror_rate=0.0,
rerror_rate=0.0, srv_rerror_rate=0.0, same_srv_rate=1.0,
diff_srv_rate=0.0, srv_diff_host_rate=0.0, dst_host_count=9.0,
dst_host_srv_count=9.0, dst_host_same_srv_rate=1.0,
dst_host_diff_srv_rate=0.0, dst_host_same_src_port_rate=0.11,
dst_host_srv_diff_host_rate=0.0, dst_host_serror_rate=0.0,
dst_host_srv_serror_rate=0.0, dst_host_rerror_rate=0.0,
dst_host_srv_rerror_rate=0.0, target=u'normal',
protocol_type_cat=1.0, service_cat=2.0, flag_cat=0.0, target_cat=2.0)
```

The resulting DataFrame looks very complete and easy to understand: all the variables have names and values. We immediately note that the categorical features are still there, for instance, we have both `protocol_type` (categorical) and `protocol_type_cat` (the numerical version of the variable mapped from categorical).

Extracting some columns from the DataFrame is as easy as using `SELECT` in a SQL query. Let's now build a list of names for all the numerical features: starting from the names found in the header, we remove the categorical ones and replace them with the numerically-derived. Finally, as we want just the features, we remove the target variable and its numerical-derived equivalent:

```
In:features_header = set(header) \
                - set(cols_categorical) \
                | set([c + "_cat" for c in cols_categorical]) \
                - set(["target", "target_cat"])
features_header = list(features_header)
```

```
print features_header
print "Total numerical features:", len(features_header)

Out:['num_access_files', 'src_bytes', 'srv_count',
'num_outbound_cmds', 'rerror_rate', 'urgent', 'protocol_type_cat',
'dst_host_same_srv_rate', 'duration', 'dst_host_diff_srv_rate',
'srv_serror_rate', 'is_host_login', 'wrong_fragment', 'serror_rate',
'num_compromised', 'is_guest_login', 'dst_host_rerror_rate',
'dst_host_srv_serror_rate', 'hot', 'dst_host_srv_count',
'logged_in', 'srv_rerror_rate', 'dst_host_srv_diff_host_rate',
'srv_diff_host_rate', 'dst_host_same_src_port_rate', 'root_shell',
'service_cat', 'su_attempted', 'dst_host_count',
'num_file_creations', 'flag_cat', 'count', 'land', 'same_srv_rate',
'dst_bytes', 'num_shells', 'dst_host_srv_rerror_rate', 'num_root',
'diff_srv_rate', 'num_failed_logins', 'dst_host_serror_rate']
Total numerical features: 41
```

Here, the `VectorAssembler` class comes to our help to build the feature matrix. We just need to pass the columns to be selected as argument and the new column to be created in the DataFrame. We decide that the output column will be named simply `features`. We apply this transformation to both training and testing sets, and then we select just the two columns that we're interested in—`features` and `target_cat`:

```
In:from pyspark.ml.feature import VectorAssembler

assembler = VectorAssembler(
    inputCols=features_header,
    outputCol="features")

Xy_train = (assembler
                .transform(train_num_df)
                .select("features", "target_cat"))
Xy_test = (assembler
                .transform(test_num_df)
                .select("features", "target_cat"))
```

Also, the default behavior of `VectorAssembler` is to produce either `DenseVectors` or `SparseVectors`. In this case, as the vector of `features` contains many zeros, it returns a sparse vector. To see what's inside the output, we can print the first line. Note that this is an action. Consequently, the job is executed before getting a result printed:

```
In:Xy_train.first()

Out:Row(features=SparseVector(41, {1: 181.0, 2: 8.0, 6: 1.0, 7: 1.0,
20: 9.0, 21: 1.0, 25: 0.11, 27: 2.0, 29: 9.0, 31: 8.0, 33: 1.0, 39:
5450.0}), target_cat=2.0)
```

Training a learner

Finally, we're arrived at the hot piece of the task: training a classifier. Classifiers are contained in the `pyspark.ml.classification` package and, for this example, we're using a random forest.

As of Spark 1.6, the extensive list of classifiers using a Python interface are as follows:

- Classification (the `pyspark.ml.classification` package):
  - `LogisticRegression`
  - `DecisionTreeClassifier`
  - `GBTClassifier` (a Gradient Boosted implementation for classification based on decision trees)
  - `RandomForestClassifier`
  - `NaiveBayes`
  - `MultilayerPerceptronClassifier`

Note that not all of them are capable of operating on multiclass problems and may have different parameters; always check the documentation related to the version in use. Beyond classifiers, the other learners implemented in Spark 1.6 with a Python interface are as follows:

- Clustering (the `pyspark.ml.clustering` package):
  - KMeans
- Regression (the `pyspark.ml.regression` package):
  - AFTSurvivalRegression (Accelerated Failure Time Survival regression)
  - DecisionTreeRegressor
  - GBTRegressor (a Gradient Boosted implementation for regression based on regression trees)
  - IsotonicRegression
  - LinearRegression
  - RandomForestRegressor
- Recommender (the `pyspark.ml.recommendation` package):
  - ALS (collaborative filtering recommender, based on Alternating Least Squares)

Let's go back to the goal of the KDD99 challenge. Now it's time to instantiate a random forest classifier and set its parameters. The parameters to set are `featuresCol` (the column containing the feature matrix), `labelCol` (the column of the DataFrame containing the target label), seed (the random seed to make the experiment replicable), and `maxBins` (the maximum number of bins to use for the splitting point in each node of the tree). The default value for the number of trees in the forest is 20, and each tree is maximum five levels deep. Moreover, by default, this classifier creates three output columns in the DataFrame: `rawPrediction` (to store the prediction score for each possible label), `probability` (to store the likelihood of each label), and `prediction` (the most probable label):

```
In:from pyspark.ml.classification import RandomForestClassifier

clf = RandomForestClassifier(
    labelCol="target_cat", featuresCol="features",
```

```
    maxBins=100, seed=101)
fit_clf = clf.fit(Xy_train)
```

Even in this case, the trained classifier is a different object. Exactly as before, the trained classifier is named the same as the classifier with the `Model` suffix:

```
In:print clf
print fit_clf

Out:RandomForestClassifier_4797b2324bc30e97fe01
RandomForestClassificationModel (uid=rfc_44b551671c42) with 20 trees
```

On the trained classifier object, that is, `RandomForestClassificationModel`, it's possible to call the `transform` method. Now we predict the label on both the training and test datasets and print the first line of the test dataset; as set in the classifier, the predictions will be found in the column named `prediction`:

```
In:Xy_pred_train = fit_clf.transform(Xy_train)
Xy_pred_test = fit_clf.transform(Xy_test)

In:print "First observation after classification stage:"
print Xy_pred_test.first()

Out:First observation after classification stage:
Row(features=SparseVector(41, {1: 105.0, 2: 1.0, 6: 2.0, 7: 1.0, 20:
254.0, 27: 1.0, 29: 255.0, 31: 1.0, 33: 1.0, 35: 0.01, 39: 146.0}),
target_cat=2.0, rawPrediction=DenseVector([0.0109, 0.0224, 19.7655,
0.0123, 0.0099, 0.0157, 0.0035, 0.0841, 0.05, 0.0026, 0.007, 0.0052,
0.002, 0.0005, 0.0021, 0.0007, 0.0013, 0.001, 0.0007, 0.0006,
0.0011, 0.0004, 0.0005]), probability=DenseVector([0.0005, 0.0011,
0.9883, 0.0006, 0.0005, 0.0008, 0.0002, 0.0042, 0.0025, 0.0001,
0.0004, 0.0003, 0.0001, 0.0, 0.0001, 0.0, 0.0001, 0.0, 0.0, 0.0,
0.0001, 0.0, 0.0]), prediction=2.0)
```

Evaluating a learner's performance

The next step in any data science task is to check the performance of the learner on the training and testing sets. For this task, we will use the F1 score as it's a good metric that merges precision and recall performances.

Evaluation metrics are enclosed in the `pyspark.ml.evaluation` package; among the few choices, we're using the one to evaluate multiclass classifiers: `MulticlassClassificationEvaluator`. As parameters, we're providing the metric (precision, recall, accuracy, f1 score, and so on) and the name of the columns containing the true label and predicted label:

```
In:
from pyspark.ml.evaluation import MulticlassClassificationEvaluator
```

```
evaluator = MulticlassClassificationEvaluator(
    labelCol="target_cat", predictionCol="prediction",
    metricName="f1")

print "F1-score train set:", evaluator.evaluate(Xy_pred_train)
print "F1-score test set:", evaluator.evaluate(Xy_pred_test)

Out:F1-score train set: 0.992356962712
F1-score test set: 0.967512379842
```

Obtained values are pretty high, and there's a big difference between the performance on the training set and the testing set.

Beyond the evaluator for multiclass classifiers, an evaluator object for regressor (where the metric can be MSE, RMSE, R2, or MAE) and binary classifiers are available in the same package.

The power of the ML pipeline

So far, we've built and displayed the output piece by piece. It's also possible to put all the operations in cascade and set them as stages of a pipeline. In fact, we can chain together what we've seen so far (the four label encoders, vector builder, and classifier) in a standalone pipeline, fit it on the training dataset, and finally use it on the test dataset to obtain the predictions.

This way to operate is more effective, but you'll lose the exploratory power of the step-by-step analysis. Readers who are data scientists are advised to use end-to-end pipelines only when they are completely sure of what's going on inside and only to build production models.

To show that the pipeline is equivalent to what we've seen so far, we compute the F1 score on the test set and print it. Unsurprisingly, it's exactly the same value:

```
In:full_stages = preproc_stages + [assembler, clf]
full_pipeline = Pipeline(stages=full_stages)
full_model = full_pipeline.fit(train_df)
predictions = full_model.transform(test_df)
print "F1-score test set:", evaluator.evaluate(predictions)

Out:F1-score test set: 0.967512379842
```

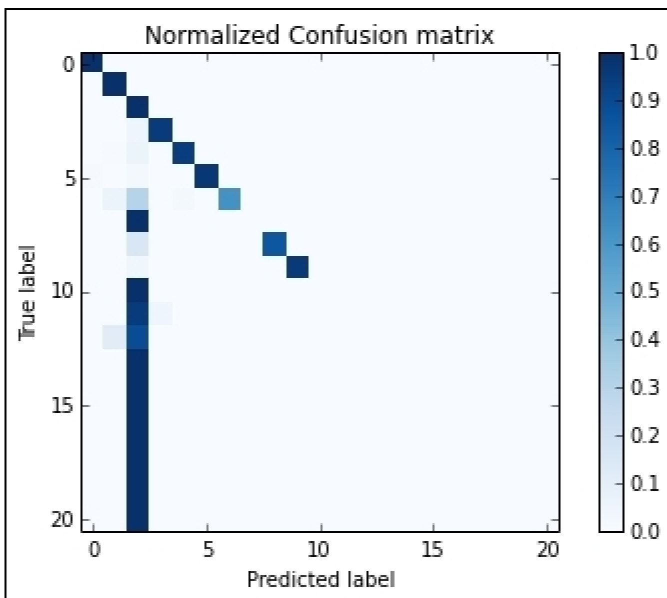
On the driver node, the one running the IPython Notebook, we can also use the `matplotlib` library to visualize the results of our analysis. For example, to show a normalized confusion matrix of the classification results (normalized by the support of each class), we can create the following function:

```
In:import matplotlib.pyplot as plt
import numpy as np
%matplotlib inline
```

```
def plot_confusion_matrix(cm):
    cm_normalized = \
        cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]
    plt.imshow(
        cm_normalized, interpolation='nearest', cmap=plt.cm.Blues)
    plt.title('Normalized Confusion matrix')
    plt.colorbar()
    plt.tight_layout()
    plt.ylabel('True label')
    plt.xlabel('Predicted label')
```

Spark is able to build a confusion matrix, but that method is in the `pyspark.mllib` package. In order to be able to use the methods in this package, we have to transform the DataFrame into an RDD using the `.rdd` method:

```
In:from pyspark.mllib.evaluation import MulticlassMetrics

metrics = MulticlassMetrics(
    predictions.select("prediction", "target_cat").rdd)
conf_matrix = metrics.confusionMatrix()tArray()
plot_confusion_matrix(conf_matrix)

Out:
```



Manual tuning

Although the F1 score was close to 0.97, the normalized confusion matrix shows that the classes are strongly unbalanced and the classifier has just learned how to classify the most popular ones properly. To improve the results, we can resample each class, trying to balance the training dataset better.

First, let's count how many cases there are in the training dataset for each class:

```
In:
train_composition =
train_df.groupBy("target").count().rdd.collectAsMap()
train_composition

Out:
{u'back': 2203,
 u'buffer_overflow': 30,
 u'ftp_write': 8,
 u'guess_passwd': 53,
 u'neptune': 107201,
 u'nmap': 231,
 u'normal': 97278,
 u'perl': 3,
 ...
...
 u'warezmaster': 20}
```

This is clear evidence of a strong imbalance. We can try to improve the performance by oversampling *rare* classes and subsampling too *popular* classes.

In this example, we will create a training dataset, where each class is represented at least 1,000 times, but up to 25,000. For this, let's first create the subsampling/oversampling rate and broadcast it throughout the cluster, and then flatMap each line of the training dataset to resample it properly:

```
In:
def set_sample_rate_between_vals(cnt, the_min, the_max):
    if the_min <= cnt <= the_max:
        # no sampling
        return 1

    elif cnt < the_min:
        # Oversampling: return many times the same observation
        return the_min/float(cnt)

    else:
        # Subsampling: sometime don't return it
        return the_max/float(cnt)
```

```
sample_rates = {k:set_sample_rate_between_vals(v, 1000, 25000)
                for k,v in train_composition.iteritems()}
sample_rates

Out:{u'back': 1,
 u'buffer_overflow': 33.333333333333336,
 u'ftp_write': 125.0,
 u'guess_passwd': 18.867924528301888,
 u'neptune': 0.23320677978750198,
 u'nmap': 4.329004329004329,
 u'normal': 0.2569954152017928,
 u'perl': 333.3333333333333,
 ...
 ...
 u'warezmaster': 50.0}

In:bc_sample_rates = sc.broadcast(sample_rates)

def map_and_sample(el, rates):
    rate = rates.value[el['target']]
    if rate > 1:
        return [el]*int(rate)
    else:
        import random
        return [el] if random.random() < rate else []

sampled_train_df = (train_df
                   .flatMap(
                     lambda x: map_and_sample(x, bc_sample_rates))
                   .toDF()
                   .cache())
```

The resampled dataset in the `sampled_train_df` DataFrame variable is also cached; we will use it many times during the hyperparameter optimization step. It should easily fit in-memory as the number of lines is lower than the original one:

```
In:sampled_train_df.count()

Out:97335
```

To get an idea of what's inside, we can print the first line. Pretty quick to print the value, isn't it? Of course, it's cached!

```
In:sampled_train_df.first()

Out:Row(duration=0.0, protocol_type=u'tcp', service=u'http',
flag=u'SF', src_bytes=217.0, dst_bytes=2032.0, land=0.0,
```

```
wrong_fragment=0.0, urgent=0.0, hot=0.0, num_failed_logins=0.0,
logged_in=1.0, num_compromised=0.0, root_shell=0.0,
su_attempted=0.0, num_root=0.0, num_file_creations=0.0,
num_shells=0.0, num_access_files=0.0, num_outbound_cmds=0.0,
is_host_login=0.0, is_guest_login=0.0, count=6.0, srv_count=6.0,
serror_rate=0.0, srv_serror_rate=0.0, rerror_rate=0.0,
srv_rerror_rate=0.0, same_srv_rate=1.0, diff_srv_rate=0.0,
srv_diff_host_rate=0.0, dst_host_count=49.0,
dst_host_srv_count=49.0, dst_host_same_srv_rate=1.0,
dst_host_diff_srv_rate=0.0, dst_host_same_src_port_rate=0.02,
dst_host_srv_diff_host_rate=0.0, dst_host_serror_rate=0.0,
dst_host_srv_serror_rate=0.0, dst_host_rerror_rate=0.0,
dst_host_srv_rerror_rate=0.0, target=u'normal')
```

Let's now use the pipeline that we created to make some predictions and print the F1 score of this new solution:

```
In:full_model = full_pipeline.fit(sampled_train_df)
predictions = full_model.transform(test_df)
print "F1-score test set:", evaluator.evaluate(predictions)

Out:F1-score test set: 0.967413322985
```

Test it on a classifier of 50 trees. To do so, we can build another pipeline (named `refined_pipeline`) and substitute the final stage with the new classifier. Performances seem the same even if the training set has been slashed in size:

```
In:clf = RandomForestClassifier(
    numTrees=50, maxBins=100, seed=101,
    labelCol="target_cat", featuresCol="features")

stages = full_pipeline.getStages()[:-1]
stages.append(clf)

refined_pipeline = Pipeline(stages=stages)

refined_model = refined_pipeline.fit(sampled_train_df)
predictions = refined_model.transform(test_df)
print "F1-score test set:", evaluator.evaluate(predictions)

Out:F1-score test set: 0.969943901769
```

Cross-validation

We can go forward with manual optimization and find the right model after having exhaustively tried many different configurations. Doing that, it would lead to both an immense waste of time (and

reusability of the code) and will overfit the test dataset. Cross-validation is instead the correct key to run the hyperparameter optimization. Let's now see how Spark performs this crucial task.

First of all, as the training will be used many times, we can cache it. Let's therefore cache it after all the transformations:

```
In:pipeline_to_clf = Pipeline(
    stages=preproc_stages + [assembler]).fit(sampled_train_df)
train = pipeline_to_clf.transform(sampled_train_df).cache()
test = pipeline_to_clf.transform(test_df)
```

The useful classes for hyperparameter optimization with-cross validation are contained in the `pyspark.ml.tuning` package. Two elements are essential: a grid map of parameters (that can be built with `ParamGridBuilder`) and the actual cross-validation procedure (run by the `CrossValidator` class).

In the example, we want to set some parameters of our classifier that won't change throughout the cross-validation. Exactly as with Scikit-learn, they're set when the classification object is created (in this case, column names, seed, and maximum number of bins).

Then, thanks to the grid builder, we decide which arguments should be changed for each iteration of the cross-validation algorithm. In the example, we want to check the classification performance changing the maximum depth of each tree in the forest from 3 to 12 (incrementing by 3) and the number of trees in the forest to 20 or 50.

Finally, we launch the cross-validation (with the `fit` method) after having set the grid map, classifier that we want to test, and number of folds. The parameter evaluator is essential: it will tell us which is the best model to keep after the cross-validation. Note that this operation may take 15-20 minutes to run (under the hood, *4*2*3=24* models are trained and tested):

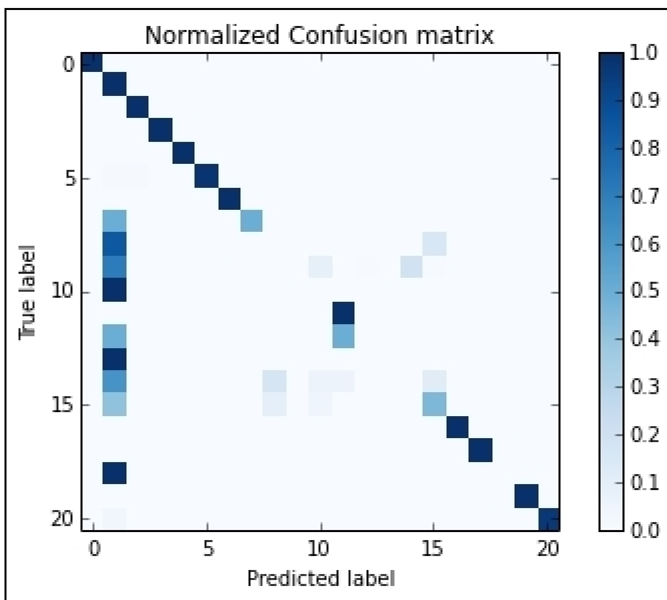
```
In:
from pyspark.ml.tuning import ParamGridBuilder, CrossValidator

rf = RandomForestClassifier(
    cacheNodeIds=True, seed=101, labelCol="target_cat",
    featuresCol="features", maxBins=100)

grid = (ParamGridBuilder()
        .addGrid(rf.maxDepth, [3, 6, 9, 12])
        .addGrid(rf.numTrees, [20, 50])
        .build())

cv = CrossValidator(
    estimator=rf, estimatorParamMaps=grid,
    evaluator=evaluator, numFolds=3)
cvModel = cv.fit(train)
```

Finally, we can predict the label using the cross-validated model as we're using a pipeline or classifier by itself. In this case, the performances of the classifier chosen with cross-validation are slightly better than in the previous case and allow us to beat the 0.97 barrier:

```
In:predictions = cvModel.transform(test)
print "F1-score test set:", evaluator.evaluate(predictions)

Out:F1-score test set: 0.97058134007
```

Furthermore, by plotting the normalized confusion matrix, you immediately realize that this solution is able to discover a wider variety of attacks, even the less popular ones:

```
In:metrics = MulticlassMetrics(predictions.select(
        "prediction", "target_cat").rdd)
conf_matrix = metrics.confusionMa().toArray()
plot_confusion_matrix(conf_matrix)

Out:
```



Final cleanup

Here, we are at the end of the classification task. Remember to remove all the variables that you've used and the temporary table that you've created from the cache:

```
In:bc_sample_rates.unpersist()
sampled_train_df.unpersist()
train.unpersist()
```

After the Spark memory is cleared, we can turn off the Notebook.

Summary

This is the final chapter of the book. We have seen how to do data science at scale on a cluster of machines. Spark is able to train and test machine learning algorithms using all the nodes in a cluster with a simple interface, very similar to Scikit-learn. It's proved that this solution is able to cope with petabytes of information, creating a valid alternative to observation subsampling and online learning.

To become an expert in Spark and streaming processing, we strongly advise you to read the book, *Mastering Apache Spark*, *Mike Frampton*, *Packt Publishing*.

If you're brave enough to switch to Scala, the main programming language for Spark, this book is the best for such a transition: *Scala for Data Science, Pascal Bugnion, Packt Publishing*.

Appendix A. Introduction to GPUs and Theano

Up until now we performed neural networks and deep learning tasks utilizing regular CPU's. Lately however, computational advantages of GPU's become widespread. This chapter dives in to the basics of GPU together with the Theano framework for deep learning.

GPU computing

When we use regular CPU computing packages for machine learning, such as Scikit-learn, the amount of parallelization is surprisingly limited because, by default, an algorithm utilizes only one core even when there are multiple cores available. In the chapter about **Classification and Regression Trees** (**CART**), we will see some advanced examples of speeding up Scikit-learn algorithms.

Unlike CPU, GPU units are designed to work in parallel from the ground up. Imagine projecting an image on a screen through a graphical card; it will come as no surprise that the GPU unit has to be able to process and project a lot of information (motion, color, and spatiality) at the same time. CPUs on the other hand are designed for sequential processing suitable for tasks where more control is needed, such as branching and checking. In contrast to the CPU, GPUs are composed of lots of cores that can handle thousands of tasks simultaneously. The GPU can outperform a CPU 100-fold at a lower cost. Another advantage is that modern GPUs are relatively cheap compared to state-of-the-art CPUs.

So all this sounds great but remember that the GPU is only good at carrying out a certain type of task. A CPU consists of a few cores optimized for sequential serial processing while a GPU consists of thousands of smaller, more efficient cores designed to handle tasks simultaneously.

CPUs and GPUs have different architectures that make them better-suited to different tasks. There are still a lot of tasks such as checking, debugging, and switching that GPUs can't do effectively because of its architecture.

A simple way to understand the difference between a CPU and GPU is to compare how they process tasks. An analogy that is often made is that of the analytical and sequential left brain (CPU) and the holistic right brain (GPU). This is just an analogy and should not be taken too seriously.

GPU	CPU
Large number of cores (but slower than CPU cores)	Small number of cores, but much faster than GPU-cores
High memory bandwidth to control the cores	Lower memory bandwidth
Special purpose	General purpose
highly parallel processing	sequential processing

See more at the following links:

- http://www.nvidia.com/object/what-is-gpu-computing.html#sthash.c4R7eJ3s.dpuf
- http://www.nvidia.com/object/what-is-gpu-computing.html#sthash.c4R7eJ3s.dpuf

In order to utilize the GPU for machine learning, a specific platform is required. Unfortunately, as of yet, there are no stable GPU computation platforms other than CUDA; this means that you must have an NVIDIA graphical card installed on your computer. GPU computing will NOT work without an NVIDIA card. Yes, I know that this is bad news for most Mac users out there. I really wish it were different but it is a limitation that we have to live with. There are other projects such as OpenCL that provide GPU computation for other GPU brands through initiatives such as **BLAS** (https://github.com/clMathLibraries/clBLAS), but they are under heavy development and are not fully optimized for deep learning applications in Python. Another limitation of OpenCL is that only AMD is actively involved so that it will be beneficial to AMD GPUs. There is no hope for a hardware-independent GPU application for machine learning in the following years (decade even!). However, check out the news and developments of the OpenCL project (https://www.khronos.org/opencl/). Considering the widespread media attention that this limitation of GPU accessibility might be quite underwhelming. Only NVIDIA seems to put their research efforts in developing GPU platforms, and it is highly unlikely to see any new serious developments in that field in the years to come.

You will need the following things for the usage of CUDA.

You need to test if the graphical card on your computer is suitable for CUDA. It should at least be an NVIDIA card. You can test if your GPU is viable for CUDA with this line of code in the terminal:

```
$ su
```

Now type your password at the root:

```
$ lspci | grep -i nvidia
```

If you do have an NVIDIA-based GNU, you can download the NVIDIA CUDA Toolkit (http://developer.nvidia.com/cuda-downloads).

At the time of writing, NVIDIA is on the verge of releasing CUDA version 8, which will have different installation procedures, so we advice you to follow the directions on the CUDA website. For further installation procedures, consult the NVIDIA website:

http://docs.nvidia.com/cuda/cuda-getting-started-guide-for-linux/#axzz3xBimv9ou

Theano – parallel computing on the GPU

Theano is a Python library originally developed by James Bergstra at the University of Montreal. It aims at providing more expressive ways to write mathematical functions with symbolic representations (F. Bastien, P. Lamblin, R. Pascanu, J. Bergstra, I. Goodfellow, A. Bergeron, N. Bouchard, D. Warde-Farley and Y. Bengio. *Theano: new features and speed improvements*. NIPS 2012 Deep Learning Workshop). Interestingly, Theano is named after the Greek mathematician, who may have been Pythagoras' wife. It's strongest points are fast c-compiled computations, symbolic expressions, and GPU computation, and Theano is under active development. Improvements are made regularly with new features. Theano's implementations are much wider than scalable machine learning so I will narrow down and use Theano for deep learning. Visit the Theano website for more information—http://deeplearning.net/software/theano/.

When we want to perform more complex computations on multidimensional matrices, basic NumPy will resort to costly loops and iterations driving up the CPU load as we have seen earlier. Theano aims to optimize these computations by compiling them into highly optimized C-code and, if possible, utilizing the GPU. For neural networks and deep learning, Theano has the useful capability to automatically differentiate mathematical functions, which is very convenient for calculation of the partial derivatives when using algorithms such as backpropagation.

Currently, Theano is used in all sorts of deep learning projects and has become the most used platform in this field. Lately, new packages have been built on top of Theano in order to make utilizing deep learning functionalities easier for us. Considering the steep learning curve of Theano, we will use packages built on Theano, such as theanets, pylearn2, and Lasagne.

Installing Theano

First, make sure that you install the development version from the Theano page. Note that if you do "$ pip install theano", you might end up with problems. Installing the development version from GitHub directly is a safer bet:

```
$ git clone git://github.com/Theano/Theano.git
$  pip install Theano
```

If you want to upgrade Theano, you can use the following command:

```
$ sudo pip install --upgrade theano
```

If you have questions and want to connect with the Theano community, you can refer to https://groups.google.com/forum/#!forum/theano-users.

That's it, we are ready to go!

To make sure that we set the directory path toward the Theano folder, we need to do the following:

```
#!/usr/bin/python
import cPickle as pickle
from six.moves import cPickle as pickle
import os

#set your path to the theano folder here
path = '/Users/Quandbee1/Desktop/pthw/Theano/'
```

Let's install all the packages that we need:

```
from theano import tensor
import theano.tensor as T
import theano.tensor.nnet as nnet
import numpy as np
import numpy
```

In order for Theano to work on the GPU (if you have an NVIDIA card + CUDA installed), we need to configure the Theano framework first.

Normally, NumPy and Theano use the double-precision floating-point format (`float64`). However, if we want to utilize the GPU for Theano, a 32-bit floating point is used. This means that we have to change the settings between 32- and 64-bits floating points depending on our needs. If you want to see which configuration is used by your system by default, type the following:

```
print(theano.config.floatX)
output: float64
```

You can to change your configuration to 32 bits for GPU computing as follows:

```
theano.config.floatX = 'float32'
```

Sometimes it is more practical to change the settings via the terminal.

For a 32-bit floating point, type as follows:

```
$ export THEANO_FLAGS=floatX=float32
```

For a 64-bit floating point, type as follows:

```
$ export THEANO_FLAGS=floatX=float64
```

If you want a certain setting attached to a specific Python script, you can do this:

```
$ THEANO_FLAGS=floatX=float32 python you_name_here.py
```

If you want to see which computational method your Theano system is using, type the following:

```
print(theano.config.device)
```

If you want to change all the settings, both bits floating point and computational method (GPU or CPU) of a specific piece of script, type as follows:

```
$ THEANO_FLAGS=device=gpu,floatX=float32 python your_script.py
```

This can be very handy for the testing and coding. You might not want to use the GPU all the time; sometimes it is better to use the CPU for the prototyping and sketching and run it on the GPU once your script is ready.

First, let's test if GPU works for your setup. You can skip this if you don't have an NVIDIA GPU card on your computer:

```
from theano import function, config, shared, sandbox
import theano.tensor as T
import numpy
import time

vlen = 10 * 30 * 768  # 10 x #cores x # threads per core
iters = 1000

rng = numpy.random.RandomState(22)
x = shared(numpy.asarray(rng.rand(vlen), config.floatX))
f = function([], T.exp(x))
print(f.maker.fgraph.toposort())
t0 = time.time()
for i in xrange(iters):
```

```
    r = f()
t1 = time.time()
print("Looping %d times took %f seconds" % (iters, t1 - t0))
print("Result is %s" % (r,))
if numpy.any([isinstance(x.op, T.Elemwise) for x in
f.maker.fgraph.toposort()]):
    print('Used the cpu')
else:
    print('Used the gpu')
```

Now that we know how to configure Theano, let's run through some simple examples to see how it works. Basically, every piece of Theano code is composed of the same structure:

1. The initialization part where the variables are declared in the class.
2. The compiling where the functions are formed.
3. The execution where the functions are applied to data types.

Let's use these principles in some basic examples of vector computations and mathematical expressions:

```
#Initialize a simple scalar
x = T.dscalar()


fx = T.exp(T.tan(x**2)) #initialize the function we want to use.

type(fx)            #just to show you that fx is a theano variable
type


#Compile create a tanh function
f = theano.function(inputs=[x], outputs=[fx])

#Execute the function on a number in this case

f(10)
```

As we mentioned before, we can use Theano for mathematical expressions. Look at this example where we use a powerful Theano feature called *autodifferentiation*, a feature that becomes highly useful for backpropagation:

```
fp = T.grad(fx, wrt=x)
fs= theano.function([x], fp)


fs(3)

output:] 4.59
```

Now that we understand the way in which we can use variables and functions, let's perform a simple logistic function:

```
#now we can apply this function to  matrices as well
x = T.dmatrix('x')
s = 1 / (1 + T.exp(-x))
logistic = theano.function([x], s)
logistic([[2, 3], [.7, -2],[1.5,2.3]])

output:
array([[ 0.88079708,  0.95257413],
       [ 0.66818777,  0.11920292],
       [ 0.81757448,  0.90887704]])
```

We can clearly see that Theano provides faster methods of applying functions to data objects than would be possible with NumPy.

Appendix A. Bibliography

This course is a blend of text and quizzes, all packaged up keeping your journey in mind. It includes content from the following Packt products:

- *Python Machine Learning Cookbook, Prateek Joshi*
- *Advanced Machine Learning with Python, John Hearty*
- *Large Scale Machine Learning with Python, Bastiaan Sjardin, Alberto Boschetti, Luca Massaron*

Index

A

- accumulators write-only variables
  - sharing, across cluster nodes / Accumulators write-only variables
- accuracy
  - evaluating, with cross validation / Getting ready…
- AdaBoost / Estimating housing prices
  - about / Applying boosting methods, CART and boosting
- Adam
  - URL / Machine learning on TensorFlow with SkFlow
  - about / Machine learning on TensorFlow with SkFlow
- Adaptive Boosting technique
  - reference / Building a face detector using Haar cascades
- adaptive gradient (ADAGRAD)
  - about / The neural network architecture
- additive expansion
  - about / Gradient Boosting Machines
- Adjusted Rand Index (ARI)
  - about / Kick-starting clustering analysis
- Affinity Propagation / Finding patterns in stock market data
- agglomerative clustering
  - about / Grouping data using agglomerative clustering
  - reference / Grouping data using agglomerative clustering
  - used, for grouping data / How to do it…
- AlexNet example
  - URL / CNN's with an incremental approach
- Anaconda
  - about / Scientific distributions
  - URL / Scientific distributions
  - URL, for packages / Scientific distributions
- architecture, neural network
  - input layer / The input layer
  - hidden layer / The hidden layer
  - output layer / The output layer
- area under the curve (AUC) / Testing the performance of our model, Describing the target
- Area Under the Curve (AUC)
  - about / Testing our prepared data
- audio data
  - reading / Reading and plotting audio data, How to do it…
  - plotting / Reading and plotting audio data, How to do it…
- audio signal
  - transforming, into frequency domain / Transforming audio signals into the frequency domain, How to do it…
- audio signals

    - generating, custom parameters used / Generating audio signals with custom parameters, How to do it…
- autoencoders
    - about / Autoencoders, Introducing the autoencoder, Autoencoders
    - topology / Topology
    - training / Training
    - denoising / Denoising autoencoders
    - unsupervised learning / Autoencoders and unsupervised learning
    - deep learning, with stacked denoising autoencoders / Autoencoders
- Average Stochastic Descent (ASGD) / Achieving SVM at scale with SGD
- averaging ensembles
    - about / Understanding averaging ensembles
    - bagging algorithms, using / Using bagging algorithms
    - random forests, using / Using random forests

B

- backoff taggers
    - about / Backoff tagging
- backoff tagging
    - about / Backoff tagging
- backpropagation
    - about / The neural network architecture
    - common problems / The neural network architecture
    - with mini batch / The neural network architecture
- bag-of-words model
    - building / Building a bag-of-words model, How to do it…, How it works…
- bagging
    - about / Bagging and random forests, Using bagging algorithms
- bagging algorithms
    - using / Using bagging algorithms
- Batch Normalization
    - about / Applying a CNN
- batch normalization function
    - URL / GPU Computing
- BeautifulSoup
    - text data, cleaning / Text cleaning with BeautifulSoup
- Best Matching Unit (BMU)
    - about / SOM – a primer
- bicycle demand distribution
    - estimating / Estimating bicycle demand distribution, How to do it…, There's more…
- bike sharing dataset
    - about / The bike-sharing dataset
    - URL / The bike-sharing dataset
- Bing Traffic API
    - about / Acquiring data via RESTful APIs, The Bing Traffic API
- BLAS
    - URL / GPU computing

- blend-of-blends / Using stacking ensembles
- blind source separation
  - about / Performing blind source separation
  - performing / Performing blind source separation, How to do it…
  - reference / Performing blind source separation
- Blocks / Knowing when to use these libraries
- Boltzmann machines
  - about / Autoencoders and unsupervised learning
- boosting
  - about / CART and boosting
- boosting methods
  - applying / Applying boosting methods
  - Extreme Gradient Boosting (XGBoost), using / Using XGBoost
- bootstrap aggregation (bagging)
  - about / Bootstrap aggregation
- Borda count / Strategies to managing model robustness
- Boston datasets
  - URL / Understanding the Scikit-learn SVM implementation
- Brill taggers
  - about / Backoff tagging
- broadcast and accumulators variables
  - sharing, across cluster nodes / Broadcast and accumulators together – an example
- broadcast read-only variables
  - sharing, across cluster nodes / Broadcast read-only variables
- bubble plots
  - plotting / Plotting bubble plots, How to do it…
  - animating / Animating bubble plots, How to do it…

C

- canny edge detector
  - about / How to do it…
  - URL / How to do it…
- carp
  - about / Sequential tagging
- cars
  - evaluating, based on characteristics / Evaluating cars based on their characteristics, Getting ready, How to do it…
- CART
  - about / CART and boosting
  - with H2O / Out-of-core CART with H2O
- cells / Introducing Jupyter/IPython
- Champion/Challenger / Strategies to managing model robustness
- characters
  - visualizing, in optical character recognition database / Visualizing the characters in an optical character recognition database, How to do it…
- chunking
  - used, for dividing text / Dividing text using chunking, How to do it…

    - about / Dividing text using chunking
- CIFAR-10 dataset
    - about / Understanding pooling layers
- Classification and Regression Trees (CART)
    - about / GPU computing
- classifier
    - constructing / Introduction
- class imbalance
    - tackling / Tackling class imbalance, How to do it…
- click-through rate (CTR)
    - about / Making large scale examples
- climate
    - about / Other useful packages to install on your system
- clustering
    - about / Introduction, Clustering – a primer
    - K-means / Clustering – K-means
- clustering algorithms
    - performance, evaluating / Evaluating the performance of clustering algorithms, How to do it…
- cluster nodes
    - variables, sharing across / Sharing variables across cluster nodes
    - broadcast read-only variables, sharing across / Broadcast read-only variables
    - accumulators write-only variables, sharing across / Accumulators write-only variables
    - broadcast and accumulators variables, sharing across / Broadcast and accumulators together – an example
- color scheme options
    - reference / How to do it…
- completeness
    - about / Selection of the best K
- completeness score
    - about / Kick-starting clustering analysis
- composable layer / Understanding the convnet topology
- Computer Vision
    - about / Introduction
- conda
    - about / Scientific distributions
- conditional random fields (CRFs)
    - about / Building Conditional Random Fields for sequential text data
    - building, for sequential text data / Building Conditional Random Fields for sequential text data, Getting ready, How to do it…
- confidence measurements
    - extracting / Extracting confidence measurements, How to do it…
- confusion matrix
    - visualizing / Visualizing the confusion matrix, How to do it…
    - about / Visualizing the confusion matrix
- Contrastive Pessimistic Likelihood Estimation (CPLE)
    - about / Introduction, Contrastive Pessimistic Likelihood Estimation

- ConvNets
  - about / Convolutional Neural Networks in TensorFlow through Keras
- convnet topology
  - about / Understanding the convnet topology
  - pooling layers / Understanding pooling layers
  - training / Training a convnet
  - forward pass / Training a convnet
  - backward pass / Training a convnet
  - implementing / Putting it all together
- Convolutional Neural Network (CNN)
  - about / Introduction to TensorFlow
- Convolutional Neural Networks (CNN)
  - about / Convolutional Neural Networks in TensorFlow through Keras
- convolutional neural networks (CNN)
  - about / Introducing the CNN
  - convnet topology / Understanding the convnet topology
  - convolution layers / Understanding convolution layers
  - applying / Applying a CNN, CNN's with an incremental approach
  - in TensorFlow, through Keras / Convolutional Neural Networks in TensorFlow through Keras
  - convolution layer / The convolution layer
  - pooling layer / The pooling layer
  - fully connected layer / The fully connected layer
  - computing, with GPU / GPU Computing
- convolution layer
  - about / The convolution layer
- convolution layers
  - about / Understanding convolution layers
- corner detection
  - about / Detecting corners, How to do it…
- correlation
  - about / Correlation
- covariance
  - about / PCA – a primer
- covertype dataset
  - about / The covertype dataset
  - URL / The covertype dataset
- cross validation
  - about / Evaluating the accuracy using cross-validation
  - used, for evaluating accuracy / Evaluating the accuracy using cross-validation, Getting ready…
- CUDA
  - about / Scale up with Python, Theano
  - reference link / Theano
- CUDA Toolkit
  - URL / Theano
- customer segmentation model

  - building / Building a customer segmentation model, How to do it…
- custom parameters
  - used, for generating audio signals / Generating audio signals with custom parameters, How to do it…
- Cygwin openssh
  - URL / Using the VM

D

- 3D scatter plots
  - plotting / Plotting 3D scatter plots
- data
  - preprocessing, different techniques used / Preprocessing data using different techniques, How to do it…
  - preprocessing ways / How to do it…
  - clustering, k-means algorithm used / Clustering data using the k-means algorithm, How to do it…
  - gourping, agglomerative clustering used / Grouping data using agglomerative clustering, How to do it…
  - preprocessing, tokenization used / Preprocessing data using tokenization, How to do it…
  - transforming, into time series format / Transforming data into the time series format, How to do it…
  - acquiring, via Twitter / Twitter
  - streaming, from resources / Streaming data from sources
- data, streaming from resources
  - about / Streaming data from sources
  - datasets, experimenting with / Datasets to try the real thing yourself
  - bike-sharing dataset, streaming / The first example – streaming the bike-sharing dataset
  - pandas I/O tools, using / Using pandas I/O tools
  - databases, working with / Working with databases
  - ordering of instances, warning / Paying attention to the ordering of instances
- data preprocessing, in Spark
  - about / Data preprocessing in Spark
  - JSON files, importing / JSON files and Spark DataFrames
  - Spark DataFrames / JSON files and Spark DataFrames
  - dealing, with missing data / Dealing with missing data
  - tables in-memory, creating / Grouping and creating tables in-memory
  - tables in-memory, grouping / Grouping and creating tables in-memory
  - preprocessed DataFrame, writing to disk / Writing the preprocessed DataFrame or RDD to disk
  - RDD, writing to disk / Writing the preprocessed DataFrame or RDD to disk
- data preprocessing ways
  - mean removal / Mean removal
  - scaling / Scaling
  - normalization / Normalization
  - binarization / Binarization
  - One Hot Encoding / One Hot Encoding

- dataset
  - splitting, for training and testing / Splitting the dataset for training and testing, How to do it…
  - reference / Building an event predictor, Getting ready, Visualizing the characters in an optical character recognition database
  - similar users, finding / Finding similar users in the dataset, How to do it…
- dataset attributes
  - buying / Getting ready
  - maint / Getting ready
  - doors / Getting ready
  - persons / Getting ready
  - lug_boot / Getting ready
  - safety / Getting ready
- datasets
  - reference link / Datasets to try the real thing yourself
  - Buzz in social media dataset, reference link / Datasets to try the real thing yourself
  - Census-Income (KDD) dataset, reference link / Datasets to try the real thing yourself
  - KDD Cup 1999 dataset, reference link / Datasets to try the real thing yourself
  - Bike-sharing dataset, reference link / Datasets to try the real thing yourself
  - BlogFeedback dataset, reference link / Datasets to try the real thing yourself
  - Covertype dataset, reference link / Datasets to try the real thing yourself
  - using / Datasets to experiment with on your own
  - bike sharing dataset / The bike-sharing dataset
  - covertype dataset / The covertype dataset
- data streams
  - used, for feature management / Feature management with data streams
- data visualization
  - about / Introduction
- date-formatted time series data
  - plotting / Plotting date-formatted time series data, How to do it…
- DBSCAN algorithm
  - about / Automatically estimating the number of clusters using DBSCAN algorithm
  - used, for estimating number of clusters / Automatically estimating the number of clusters using DBSCAN algorithm, How to do it…
  - reference / Automatically estimating the number of clusters using DBSCAN algorithm
- decision boundaries
  - about / Neural networks and decision boundaries
- decision tree regressor / Estimating housing prices
- deep belief network (DBN)
  - about / Deep belief networks
  - training / Training a DBN
  - applying / Applying the DBN
  - validating / Validating the DBN
- Deep Belief Networks (DBN)
  - about / Autoencoders and unsupervised learning
- DeepFace
  - about / Introducing the CNN

- deep learning
  - scaling, with H2O / Deep learning at scale with H2O
  - unsupervised pretraining / Deep learning and unsupervised pretraining
- deep neural network
  - building / Building a deep neural network, How to do it…
- denoising autoencoders
  - about / Autoencoders and unsupervised learning, Autoencoders
- denoising autoencoders (dA)
  - about / Denoising autoencoders
  - applying / Applying a dA
- DepthConcat element
  - about / Putting it all together
- development tools
  - about / Alternative development tools
  - Lasagne / Alternative development tools
  - TensorFlow / Alternative development tools
  - libraries usage, deciding / Knowing when to use these libraries
- Diabolo network
  - about / Autoencoders
- direct acyclic graph (DAG) / Working with Spark DataFrames
- Directed Acyclic Graph (DAG)
  - about / pySpark
- distributed filesystem (dfs)
  - about / HDFS
- distributed framework
  - need for / Why do we need a distributed framework?
- dynamic applications
  - models, using / Using models in dynamic applications
- dynamic signals
  - animating / Animating dynamic signals, How to do it…

E

- edge detection
  - about / Detecting edges
  - performing / Detecting edges, How to do it…
- eigenvalue
  - about / PCA – a primer
- eigenvector
  - about / PCA – a primer
- Elastic Compute Cloud (EC2)
  - about / Out-of-core learning
- elbow method
  - about / Tuning your clustering configurations, Applying boosting methods
- Elbow method
  - about / Selection of the best K
- empty squares / Getting ready
- ensembles

- about / Introducing ensembles
    - averaging ensembles / Understanding averaging ensembles
    - boosting methods, applying / Applying boosting methods
    - stacking ensembles, using / Using stacking ensembles
    - applying / Applying ensembles in practice
- epsilon
    - about / Automatically estimating the number of clusters using DBSCAN algorithm
- error correcting tournament (ECT)
    - about / The covertype dataset crunched by VW
- Euclidean distance score
    - computing / Computing the Euclidean distance score, How to do it…
- event predictor
    - building / Building an event predictor, Getting ready, How to do it…
- exemplars
    - reference / Finding patterns in stock market data
- expansion
    - about / The hidden layer
- expectation (E)
    - about / Clustering – K-means
- expectation-maximization (EM) algorithm
    - about / Clustering – K-means
- explicit high-dimensional mappings
    - about / Trying explicit high-dimensional mappings
- Exploratory Data Analysis (EDA)
    - about / Unsupervised methods
- Extreme Gradient Boosting (XGBoost)
    - using / Using XGBoost
- extreme gradient boosting (XGBoost)
    - about / CART and boosting, XGBoost
    - reference link / XGBoost
    - regression / XGBoost regression
    - variable importance, plotting / XGBoost and variable importance
    - large datasets, streaming / XGBoost streaming large datasets
    - model persistence / XGBoost model persistence
- extreme gradient boosting (XGBoost), parameters
    - eta [] / XGBoost
    - min_child_weight [] / XGBoost
    - max_depth [] / XGBoost
    - subsample [] / XGBoost
    - colsample_bytree [] / XGBoost
    - lambda [] / XGBoost
    - seed [] / XGBoost
- extremely random forests (ERFs)
    - about / Training an image classifier using Extremely Random Forests
    - used, for training image classifier / Training an image classifier using Extremely Random Forests, How to do it…
    - reference / Training an image classifier using Extremely Random Forests

- extremely randomized forest
    - about / Random forest and extremely randomized forest
    - URL / Random forest and extremely randomized forest
- extremely randomized trees
    - for randomized search / Extremely randomized trees and large datasets
- extremely randomized trees (ExtraTrees)
    - about / Using random forests
- eye and nose detectors
    - building / Building eye and nose detectors, How to do it…

F

- face dataset
    - reference / Building a face recognizer using Local Binary Patterns Histogram
- face detector
    - building, Haar cascades used / Building a face detector using Haar cascades, How to do it…
- face recognition
    - about / Introduction
- face recognizer
    - building, Local Binary Patterns Histograms used / Building a face recognizer using Local Binary Patterns Histogram, How to do it…
- Fast Fourier Transform
    - about / Training a convnet
- fast parameter
    - optimizing, with randomized search / Fast parameter optimization with randomized search
- feature decomposition
    - principal component analysis (PCA) / Feature decomposition – PCA
- feature engineering
    - about / Introduction, Feature engineering in practice
    - data, acquiring via RESTful APIs / Acquiring data via RESTful APIs
    - variables, deriving / Deriving and selecting variables using feature engineering techniques
    - variables, selecting / Deriving and selecting variables using feature engineering techniques
    - weather API, creating / The weather API
- feature engineering, for ML applications
    - about / Engineering features for ML applications
    - rescaling techniques, using / Using rescaling techniques to improve the learnability of features
    - effective derived variables, creating / Creating effective derived variables
    - non-numeric features, reinterpreting / Reinterpreting non-numeric features
- feature management, with data streams
    - about / Feature management with data streams
    - target, describing / Describing the target
    - hashing trick / The hashing trick
    - basis transformations / Other basic transformations

- testing / Testing and validation in a stream
- validation / Testing and validation in a stream
- SGD, using / Trying SGD in action

- features
  - creating, visual codebook and vector quantization used / Creating features using visual codebook and vector quantization, How to do it…
- feature selection
  - techniques, using / Using feature selection techniques
  - performing / Performing feature selection
  - correlation / Correlation
  - LASSO / LASSO
  - Recursive Feature Elimination (RFE) / Recursive Feature Elimination
  - genetic models / Genetic models
  - by regularization / Feature selection by regularization
- feature set
  - creating / Creating a feature set
  - feature engineering, for ML applications / Engineering features for ML applications
  - feature selection techniques, using / Using feature selection techniques
- feedforward neural network
  - about / The neural network architecture
- Fisher's discriminant ratio
  - about / Finessing your self-training implementation
- forward propagation
  - about / The neural network architecture
- Fourier transforms
  - URL / Transforming audio signals into the frequency domain
- frequency domain
  - audio signal, transforming / Transforming audio signals into the frequency domain, How to do it…
- frequency domain features
  - extracting / Extracting frequency domain features, How to do it…
- fully connected layer
  - about / The fully connected layer
- Fully Connected layer
  - about / Putting it all together
- function compositions
  - building, for data processing / Building function compositions for data processing, How to do it…

G

- gender
  - identifying / Identifying the gender, How to do it…
- genetic models
  - about / Genetic models
- Gensim
  - about / Gensim, Scaling LDA – memory, CPUs, and machines
  - URL / Gensim

- Gensim package
    - URL / LDA
- Gibbs sampling
    - about / Training
- Gini Impurity (gini) / Using stacking ensembles
- Git for Windows
    - URL / XGBoost
    - about / XGBoost
- Go
    - about / Introducing the CNN
- GoogLeNet / Introducing the CNN
    - about / Putting it all together
- Gource
    - URL / Nonlinear and faster with Vowpal Wabbit
- GPU
    - neural network, with theanets / Deep learning with theanets
    - computing / GPU computing, GPU computing
    - using, for convolutional neural networks (CNN) / GPU Computing
    - reference link, for computing / GPU computing
    - parallel computing, with Theano / Theano – parallel computing on the GPU
- gradient boosting machine (GBM)
    - about / CART and boosting, Gradient Boosting Machines
- gradient boosting machine (GBM)
    - max_depth / max_depth
    - learning_rate / learning_rate
    - subsample / Subsample
    - warm_start / Faster GBM with warm_start
    - speeding up, with warm_start / Speeding up GBM with warm_start
    - GBM models, training / Training and storing GBM models
    - GBM models, storing / Training and storing GBM models
- gradient descent
    - reference / How to do it…
- gradient descent algorithms
    - URL / Using rescaling techniques to improve the learnability of features
- graphical user interface (GUI)
    - about / VirtualBox
- gridsearch
    - on H2O / Gridsearch on H2O, Stochastic gradient boosting and gridsearch on H2O
- guests
    - about / VirtualBox

H

- h-dimensional representation / Introducing the autoencoder
- H2O
    - about / Scale up with Python, H2O
    - URL / H2O
    - deep learning, scaling / Deep learning at scale with H2O

- large scale deep learning / Large scale deep learning with H2O
- gridsearch / Gridsearch on H2O, Random forest and gridsearch on H2O, Stochastic gradient boosting and gridsearch on H2O
- CART / Out-of-core CART with H2O
- random forest / Random forest and gridsearch on H2O
- stochastic gradient boosting / Stochastic gradient boosting and gridsearch on H2O
- principal component analysis (PCA) / PCA with H2O
- K-means / K-means with H2O

- Haar cascades
  - about / Building a face detector using Haar cascades
  - used, for face detection / Building a face detector using Haar cascades, How to do it…
- Hadoop
  - ecosystem / The Hadoop ecosystem
  - architecture / Architecture
  - Distributed File System (HDFS) / HDFS
  - MapReduce / MapReduce
  - Yet Another Resource Negotiator (YARN) / YARN
- Hadoop Distributed File System (HDFS)
  - about / Explaining scalability in detail, HDFS
- hard-margin classifiers
  - about / Support Vector Machines
- Harris corner detector function
  - reference / How to do it…
- heart dataset
  - URL / Implementing self-training
- heat maps
  - visualizing / Visualizing heat maps, How to do it…
- hidden layers
  - about / Introduction
- hidden Markov models (HMMs)
  - building / Building Hidden Markov Models, How to do it…
  - URL / Building Hidden Markov Models
  - about / Building Hidden Markov Models for sequential data
  - building, for sequential data / Building Hidden Markov Models for sequential data, How to do it…
  - used, for analyzing stock market data / Analyzing stock market data using Hidden Markov Models, How to do it…
- hierarchical clustering
  - about / Grouping data using agglomerative clustering
- Hierarchical Dirichler Processing (HDP)
  - about / Scaling LDA – memory, CPUs, and machines
- hierarchical grouping
  - about / Stacked Denoising Autoencoders
- hinge loss
  - about / Hinge loss and its variants
  - variants / Hinge loss and its variants
- histogram equalization

    - about / Histogram equalization, How to do it…
- histograms
    - plotting / Plotting histograms, How to do it…
- homogeneity
    - about / Selection of the best K
- homogeneity score
    - about / Kick-starting clustering analysis
- host
    - about / VirtualBox
- housing prices
    - estimating / Estimating housing prices, Getting ready, How to do it…
- hyperparameter optimization
    - about / Neural networks and hyperparameter optimization
- hyperparameters
    - about / Extracting validation curves
    - tuning / Hyperparameter tuning
- hypotrochoid / How to do it…

I

- i-dimensional input
    - about / Introducing the autoencoder
- identity function
    - about / Autoencoders
- image
    - compressing, vector quantization used / Compressing an image using vector quantization, How to do it…
- image classifier
    - training, extremely random forests used / Training an image classifier using Extremely Random Forests, How to do it…
- ImageNet
    - about / Introducing the CNN
- images
    - operating, OpenCV-Python used / Operating on images using OpenCV-Python, How to do it…
- Inception network
    - about / Putting it all together
- income bracket
    - estimating / Estimating the income bracket, How to do it…
- incremental learning
    - about / Deep learning with large files – incremental learning
- incremental PCA
    - about / Incremental PCA
- independent and identically distributed (i.i.d) / Stochastic gradient descent
- input validator
    - URL / Understanding the VW data format
- inverse document frequency (IDF) / Building a text classifier, How it works…
- IPython

    - about / Introducing Jupyter/IPython
    - URL / Introducing Jupyter/IPython
- Iris datasets
    - URL / Understanding the Scikit-learn SVM implementation

J

- Java Development Kit (JDK)
    - about / H2O
- Jupyter
    - about / Introducing Jupyter/IPython
    - URL, for example / Introducing Jupyter/IPython
    - URL, for installing / Introducing Jupyter/IPython
    - URL / Introducing Jupyter/IPython
- Jupyter Notebook Viewer
    - URL / Introducing Jupyter/IPython
- Just-in-Time (JIT) compiler
    - about / Scale up with Python

K

- K-means
    - about / Clustering – K-means
    - initialization methods / Initialization methods
    - assumptions / K-means assumptions
    - selecting / Selection of the best K
    - scaling / Scaling K-means – mini-batch
    - with H2O / K-means with H2O
- k-means algorithm
    - about / Clustering data using the k-means algorithm
    - used, for clustering data / Clustering data using the k-means algorithm, How to do it…
    - reference / Clustering data using the k-means algorithm
- k-means clustering
    - about / How to do it…, Introducing k-means clustering
    - clustering / Clustering – a primer
    - clustering analysis / Kick-starting clustering analysis
    - configuration, tuning / Tuning your clustering configurations
- k-nearest neighbors
    - about / Constructing a k-nearest neighbors classifier
- K-Nearest Neighbors (KNN) / Using bagging algorithms
- k-nearest neighbors classifier
    - constructing / Constructing a k-nearest neighbors classifier, How to do it…, How it works…
- k-nearest neighbors regressor
    - constructing / Constructing a k-nearest neighbors regressor, How to do it…, How it works…
- Kaggle
    - URL / XGBoost

- KDD99 challenge
  - reference / Spark on the KDD99 dataset
- Keras / Knowing when to use these libraries
  - about / Keras
  - URL / Keras, Keras and TensorFlow installation
  - installing / Keras and TensorFlow installation
  - convolutional neural networks (CNN), in TensorFlow / Convolutional Neural Networks in TensorFlow through Keras
- Kernel Principal Components Analysis
  - performing / Performing Kernel Principal Components Analysis, How to do it…
  - reference / Performing Kernel Principal Components Analysis
- kernels
  - reference / How to do it…
- KSVM
  - about / A few examples using reductions for SVM and neural nets
  - reference link / A few examples using reductions for SVM and neural nets

L

- label encoding
  - about / Label encoding, How to do it…
- Laplacian edge detector
  - about / How to do it…
  - URL / How to do it…
- large scale cloud services
  - URL / Deep learning with large files – incremental learning
- large scale deep learning
  - with H2O / Large scale deep learning with H2O
- large scale machine learning
  - Python / Python for large scale machine learning
- Lasagne
  - about / Introduction to Lasagne, Getting to know Lasagne
- LASSO
  - about / LASSO
- LaSVM
  - URL / Other alternatives for SVM fast learning
  - about / Other alternatives for SVM fast learning
- Latent Dirichlect Allocation (LDA) / Nonlinear and faster with Vowpal Wabbit
- Latent Dirichlet Allocation (LDA)
  - about / How to do it…, Gensim, LDA
  - reference / How it works…
  - scaling / Scaling LDA – memory, CPUs, and machines
- Latent Semantic Analysis (LSA)
  - about / Gensim
- learning curves
  - about / Extracting learning curves
  - extracting / Extracting learning curves, How to do it…
- learning vector quantization (LVQ) neural network / How to do it…

- lemmatization
    - used, for converting text to base form / Converting text to its base form using lemmatization, How to do it…
- LeNet
    - about / Putting it all together
- liblinear-cdblock library
    - URL / Other alternatives for SVM fast learning
- libraries
    - usage, deciding / Knowing when to use these libraries
- Library for Support Vector Machines (LIBSVM)
    - about / Understanding the Scikit-learn SVM implementation
    - URL / Understanding the Scikit-learn SVM implementation
- library for Support Vector Machines (LIBSVM)
    - about / Scale up with Python
- linear classifier
    - building, Support Vector Machine (SVMs) used / Building a linear classifier using Support Vector Machine (SVMs), Getting ready, How to do it…
- linear regression
    - with SGD / Linear regression with SGD
- linear regressor
    - building / Building a linear regressor, Getting ready, How to do it…
- Linux precompiled binaries
    - URL / Installing VW
- Local Binary Patterns Histograms
    - used, for building face recognizer / Building a face recognizer using Local Binary Patterns Histogram, How to do it…
    - about / Building a face recognizer using Local Binary Patterns Histogram
    - reference / Building a face recognizer using Local Binary Patterns Histogram
- logistic regression classifier
    - building / Building a logistic regression classifier, How to do it…

M

- machine learning
    - on TensorFlow, with SkFlow / Machine learning on TensorFlow with SkFlow
    - incremental learning / Deep learning with large files – incremental learning
- machine learning, with Spark
    - about / Machine learning with Spark
    - dataset, reading / Reading the dataset
    - feature engineering / Feature engineering
    - training, giving to learner / Training a learner
    - learner's performance, evaluating / Evaluating a learner's performance
    - ML pipeline, power / The power of the ML pipeline
    - manual tuning / Manual tuning
    - cross-validation / Cross-validation, Final cleanup
- machine learning pipelines
    - building / Building machine learning pipelines, How to do it…, How it works…
- MapReduce

    - about / Explaining scalability in detail, MapReduce
    - data chunker / MapReduce
    - mapper / MapReduce
    - shuffler / MapReduce
    - reducer / MapReduce
    - output writer / MapReduce
- Markov Chain Monte Carlo (MCMC)
    - about / Training
- matplotlib
    - URL / Introduction
- matplotlib package
    - about / The matplotlib package
    - URL / The matplotlib package
- max-pooling
    - about / Understanding pooling layers
- maximization (M)
    - about / Clustering – K-means
- mean-pooling
    - about / Understanding pooling layers
- mean shift
    - about / Building a Mean Shift clustering model
    - reference / Building a Mean Shift clustering model
- mean shift clustering model
    - building / Building a Mean Shift clustering model, How to do it…
- Mel Frequency Cepstral Coefficients (MFCC)
    - about / Extracting frequency domain features
    - URL / Extracting frequency domain features
- memory profiler
    - about / Other useful packages to install on your system
- mini batches
    - about / The neural network architecture
- Minimalist GNU for Windows (MinGW) compiler
    - about / XGBoost
    - URL / XGBoost
- MLlib / Machine learning with Spark
- modeling risk factors
    - longitudinally variant / Identifying modeling risk factors
    - slow change / Identifying modeling risk factors
    - Key parameter / Identifying modeling risk factors
- model persistence
    - achieving / Achieving model persistence, How to do it…
- models
    - using, in dynamic applications / Using models in dynamic applications
    - robustness / Understanding model robustness
    - modeling risk factors, identifying / Identifying modeling risk factors
    - robustness, managing / Strategies to managing model robustness
- momentum

    - training / The neural network architecture
- Motor Vehicle Accident (MVA) / Translink Twitter
- movie recommendations
    - generating / Generating movie recommendations, How to do it…
- MrJob
    - about / MapReduce
- Multi-Layer Perceptron (MLP)
    - about / The composition of a neural network
- multicollinearity / Correlation
- music
    - synthesizing / Synthesizing music, How to do it…
    - URL / Synthesizing music
- music recommendation engine
    - references / GPU Computing

N

- n-dimensional input
    - about / Denoising autoencoders
- n-gram tagger
    - about / Sequential tagging
- natural language processing (NLP)
    - about / Introduction
- Natural Language Toolkit (NLTK)
    - about / Introduction, Tagging and categorising words
    - references / Introduction
    - used, for tagging / Tagging with NLTK
- Naïve Bayes classifier
    - about / Building a Naive Bayes classifier
    - building / How to do it…
- nearest neighbors
    - finding / Finding the nearest neighbors, How to do it…
    - about / Finding the nearest neighbors
- Nesterov momentum
    - about / The neural network architecture
- Network In Network (NIN)
    - about / Putting it all together
- network topologies
    - about / Network topologies
- neural network
    - architecture / The neural network architecture
    - softmax, for classification / The neural network architecture
    - forward propagation / The neural network architecture
    - backpropagation / The neural network architecture
    - backpropagation, common problems / The neural network architecture
    - backpropagation, with mini batch / The neural network architecture
    - momentum, training / The neural network architecture
    - Nesterov momentum / The neural network architecture

    - adaptive gradient (ADAGRAD) / The neural network architecture
    - resilient backpropagation (RPROP) / The neural network architecture
    - RMSPROP / The neural network architecture
    - architecture, selecting / What and how neural networks learn, Choosing the right architecture
    - implementing / Neural networks in action
    - sknn, parallelizing / Parallelization for sknn
    - hyperparameter, optimizing / Neural networks and hyperparameter optimization
    - decision boundaries / Neural networks and decision boundaries
    - on GPU, with theanets / Deep learning with theanets
    - performing, in TensorFlow / A neural network from scratch in TensorFlow
- Neural network
    - regularization / Neural networks and regularization
- neural networks
    - reference, for tutorial / Introduction
    - used, for building optical character / Building an optical character recognizer using neural networks, How to do it…
    - about / Neural networks – a primer
    - composition / The composition of a neural network
    - learning process / The composition of a neural network
    - neurons / The composition of a neural network
    - connectivity functions / The composition of a neural network
    - network topologies / Network topologies
- Neural Network Toolbox (NNT)
    - about / Other useful packages to install on your system
- NeuroLab
    - reference / Introduction
    - about / Other useful packages to install on your system
- neurons
    - about / Introduction
- nonlinear classifier
    - building, SVMs used / Building a nonlinear classifier using SVMs, How to do it…
- nonlinear SVMs
    - pursuing, by subsampling / Pursuing nonlinear SVMs by subsampling
- number of clusters
    - estimating automatically, DBSCAN algorithm used / Automatically estimating the number of clusters using DBSCAN algorithm, How to do it…
- NumPy
    - URL / Introduction, NumPy
    - about / Scale up with Python, NumPy
- NVIDIA
    - URL / GPU computing
- NVIDIA CUDA Toolkit
    - URL / GPU computing

O

- object recognizer

    - building / Building an object recognizer, How to do it…
- one-against-all (OAA)
    - about / The covertype dataset crunched by VW
- one-hot encoder
    - reference link / The hashing trick
- one-vs-all (OVA) / Achieving SVM at scale with SGD
- one-vs-all (OVA) strategy / The Scikit-learn SGD implementation
- OpenCL project
    - URL / GPU computing
- OpenCV
    - about / Introduction
    - URL / Introduction
- OpenCV-Python
    - used, for operating on images / Operating on images using OpenCV-Python, How to do it…
- Openssh
    - URL, for Windows / Using the VM
- OpinRank Review dataset
    - about / Applying the SdA
    - URL / Applying the SdA
- optical character recognizer
    - building, neural networks used / Building an optical character recognizer using neural networks, How to do it…
- optimal hyperparameters
    - searching / Finding optimal hyperparameters, How to do it…
- Ordinary Least Squares
    - about / Getting ready
- Ordinary least squares (OLS) / The Scikit-learn SGD implementation
- orthogonalization
    - about / PCA – a primer
- orthonormalization
    - about / PCA – a primer
- out-of-core learning
    - about / Out-of-core learning
    - subsampling, as viable option / Subsampling as a viable option
    - instances, optimizing / Optimizing one instance at a time
    - building / Building an out-of-core learning system
- overcomplete
    - about / Denoising autoencoders
- overshooting
    - about / The neural network architecture

P

- pandas
    - URL / Transforming data into the time series format, Pandas
    - about / Scale up with Python, Pandas
- pandas documentation

- reference link / Using pandas I/O tools
- patterns
  - finding, in stock market data / Finding patterns in stock market data, How to do it…
- Pearson correlation score
  - computing / Computing the Pearson correlation score, How to do it…
- perceptron
  - about / Building a perceptron
  - building / Building a perceptron, How to do it…
- performance report
  - extracting / Extracting the performance report, How to do it…
- Permanent Contrastive Divergence (PCD)
  - about / Training
- pie charts
  - drawing / Drawing pie charts, How to do it…
- Platt calibration
  - about / Implementing self-training
- Platt scaling
  - about / How to do it…
  - reference / How to do it…
- polynomial regressor
  - building / Building a polynomial regressor, Getting ready, How to do it…
- pooling layer
  - about / The pooling layer
- pooling layers
  - about / Understanding pooling layers
- porter stemmer
  - about / Stemming
- Pragmatic Chaos model / Using stacking ensembles
- predictive modeling
  - about / Introduction
- pretraining
  - about / Autoencoders and unsupervised learning
- price-earnings (P/E) ratio / Creating effective derived variables
- principal component analysis (PCA)
  - about / Principal component analysis, Machine learning on TensorFlow with SkFlow, Feature decomposition – PCA
  - features / PCA – a primer
  - employing / Employing PCA
  - reference link / Machine learning on TensorFlow with SkFlow
  - randomized PCA / Randomized PCA
  - incremental PCA / Incremental PCA
  - sparse PCA / Sparse PCA
  - with H2O / PCA with H2O
- Principal Component Analysis (PCA)
  - about / Autoencoders
- Principal Components Analysis (PCA)
  - performing / Performing Principal Components Analysis, How to do it…

- reference / Performing Principal Components Analysis
- Putty
  - URL / Using the VM
- Pylearn2 / Knowing when to use these libraries
- PyPy
  - about / Scale up with Python
  - URL / Scale up with Python
- pySpark
  - about / pySpark
- pySpark, actions
  - reduce(function) / pySpark
  - count() / pySpark
  - countByKey() / pySpark
  - collect() / pySpark
  - first() / pySpark
  - take(N) / pySpark
  - takeSample(withReplacement, N, seed) / pySpark
  - takeOrdered(N, ordering) / pySpark
  - saveAsTextFile(path) / pySpark
- pySpark, methods
  - cache() / pySpark
  - persist(storage) / pySpark
  - unpersist() / pySpark
- pystruct
  - reference / Getting ready
- Python
  - about / Introducing Python
  - advantages / Introducing Python
  - URL / Introducing Python
  - scaling up, with / Scale up with Python
  - scaling out, with / Scale out with Python
  - large scale machine learning / Python for large scale machine learning
  - 2 and Python 3, selecting between / Choosing between Python 2 and Python 3
  - packages, upgrading / Package upgrades
  - scientific distribution / Scientific distributions
  - Jupyter / Introducing Jupyter/IPython
  - IPython / Introducing Jupyter/IPython
  - reference link / Describing the target
  - integrating, with Vowpal Wabbit (VW) / Python integration
- Python-Future
  - URL / Choosing between Python 2 and Python 3
- Python 2
  - and Python 3, selecting between / Choosing between Python 2 and Python 3
- Python 2-3 compatible code
  - URL / Choosing between Python 2 and Python 3
- Python 3
  - and Python 2, selecting between / Choosing between Python 2 and Python 3

    - URL, for compatibility / Choosing between Python 2 and Python 3
- Python packages
    - NumPy / Introduction, NumPy
    - SciPy / Introduction, SciPy
    - scikit-learn / Introduction
    - matplotlib / Introduction
    - about / Python packages
    - pandas / Pandas
    - Scikit-learn / Scikit-learn
- pyvw / Python integration

Q

- quadratic programming
    - URL / Support Vector Machines

R

- radial basis functions (RBF) / Support Vector Machines
- random forest
    - about / Random forest and extremely randomized forest
- random forest, parameters for bagging
    - n_estimators / Random forest and extremely randomized forest
    - max_features / Random forest and extremely randomized forest
    - min_sample_leaf / Random forest and extremely randomized forest
    - max_depth / Random forest and extremely randomized forest
    - criterion / Random forest and extremely randomized forest
    - min_samples_split / Random forest and extremely randomized forest
- random forest regressor / Estimating bicycle demand distribution
- random forests
    - reference / Training an image classifier using Extremely Random Forests
    - about / Bagging and random forests, Testing our prepared data
    - using / Using random forests
- randomized PCA
    - about / Randomized PCA
- randomized search
    - about / Neural networks and hyperparameter optimization, Fast parameter optimization with randomized search
    - fast parameter, optimizing / Fast parameter optimization with randomized search
    - extremely randomized trees / Extremely randomized trees and large datasets
    - large datasets / Extremely randomized trees and large datasets
- random patches
    - about / Using bagging algorithms
- Random Patches
    - about / Bagging and random forests
- random subspace method
    - about / Stochastic gradient boosting and gridsearch on H2O
- random subspaces

    - about / Using bagging algorithms
- receiver operating characteristic (ROC) / Describing the target
- recommendation engine
    - about / Introduction
- reconstruction error
    - about / Autoencoders
- rectified linear unit (ReLU)
    - about / The neural network architecture
- Rectified Linear Units (ReLU)
    - about / Putting it all together
- recurrent neural network
    - building, for sequential data analysis / Building a recurrent neural network for sequential data analysis, How to do it…
    - reference / Building a recurrent neural network for sequential data analysis
- Recursive Feature Elimination (RFE) / Performing feature selection
    - about / Recursive Feature Elimination
- regression
    - about / Building a linear regressor
- regression accuracy
    - computing / Computing regression accuracy, How to do it…
    - mean absolute error / Getting ready
    - mean squared error / Getting ready
    - median absolute error / Getting ready
    - explained variance score / Getting ready
    - R2 score / Getting ready
- regularization / Getting ready
    - feature selection / Feature selection by regularization
    - about / Neural networks and regularization
- relative importance of features
    - computing / Computing the relative importance of features, How to do it…
- residual network (ResNet)
    - about / GPU Computing
- resilient backpropagation (RPROP)
    - about / The neural network architecture
- Resilient Distributed Dataset (RDD)
    - about / pySpark
- RESTful APIs
    - data, acquiring / Acquiring data via RESTful APIs
    - model performance, testing / Testing the performance of our model
- Restricted Boltzmann Machine (RBM)
    - about / Restricted Boltzmann Machine, Introducing the RBM
    - topology / Topology
    - training / Training
    - applications / Applications of the RBM, Further applications of the RBM
- Ridge Regression / Getting ready
- ridge regressor
    - building / Building a ridge regressor, Getting ready, How to do it…

- RMSPROP
    - about / The neural network architecture
- Root Mean Squared Error (RMSE)
    - about / Genetic models

S

- scalability
    - about / Explaining scalability in detail
    - large scale examples, creating / Making large scale examples
    - Python / Introducing Python
    - Python, scaling up with / Scale up with Python
    - Python, scaling out with / Scale out with Python
- Scale Invariant Feature Transform (SIFT)
    - about / Detecting SIFT feature points
- scientific distribution
    - about / Scientific distributions
- Scikit-learn
    - about / Scikit-learn, Understanding the Scikit-learn SVM implementation
    - URL / Scikit-learn
    - matplotlib package / The matplotlib package
    - Gensim / Gensim
    - H2O / H2O
    - XGBoost / XGBoost
    - Theano / Theano
    - TensorFlow / TensorFlow
    - sknn library / The sknn library
    - theanets / Theanets
    - Keras / Keras
    - useful packages, installing / Other useful packages to install on your system
    - other alternatives / Other alternatives for SVM fast learning
    - Vowpal Wabbit (VW) / Nonlinear and faster with Vowpal Wabbit
- scikit-learn
    - about / Employing PCA
- Scikit-learn documentation
    - reference link / Describing the target
- scikit-neuralnetwork
    - about / The sknn library
- SciPy
    - URL / Introduction, SciPy
    - about / SciPy
- Self-Organizing Map (SOM)
    - about / The composition of a neural network
- self-organizing maps (SOM)
    - about / Self-organizing maps, SOM – a primer
    - employing / Employing SOM
- self-training
    - about / Self-training

  - implementing / Implementing self-training
  - improving / Finessing your self-training implementation
  - selection process, improving / Improving the selection process
  - Contrastive Pessimistic Likelihood Estimation (CPLE) / Contrastive Pessimistic Likelihood Estimation
- semi-supervised algorithms
  - using / Semi-supervised algorithms in action
- semi-supervised learning
  - about / Introduction, Understanding semi-supervised learning
  - self-training / Self-training
- sentiment analysis
  - about / Analyzing the sentiment of a sentence
  - performing / Analyzing the sentiment of a sentence, How to do it…, How it works…
- sequential tagging
  - about / Sequential tagging
- SGD
  - used, for achieving SVM / Achieving SVM at scale with SGD
  - non-linearity, including / Including non-linearity in SGD
  - explicit high-dimensional mappings / Trying explicit high-dimensional mappings
  - linear regression / Linear regression with SGD
- SGD function, parameters
  - lr / Keras and TensorFlow installation
  - decay / Keras and TensorFlow installation
  - momentum / Keras and TensorFlow installation
  - nesterov / Keras and TensorFlow installation
  - optimizer / Keras and TensorFlow installation
- SGD Scikit-learn implementation
  - reference link / Defining SGD learning parameters
- SIFT feature points
  - detecting / Detecting SIFT feature points, How to do it…
- sigmoid
  - about / The neural network architecture
- Silhouette
  - about / Selection of the best K
- Silhouette Coefficient
  - about / Kick-starting clustering analysis
- Silhouette Coefficient score
  - about / Evaluating the performance of clustering algorithms
- simple classifier
  - building / Building a simple classifier, How to do it…, There's more…
- single layer neural network
  - building / Building a single layer neural network, How to do it…
- Singular Value Decomposition (SVD)
  - about / Feature decomposition – PCA
- SkFlow
  - machine learning, on TensorFlow / Machine learning on TensorFlow with SkFlow
- sklearn.svm module, parameters

    - C / Understanding the Scikit-learn SVM implementation
    - kernel / Understanding the Scikit-learn SVM implementation
    - degree / Understanding the Scikit-learn SVM implementation
    - gamma / Understanding the Scikit-learn SVM implementation
    - nu / Understanding the Scikit-learn SVM implementation
    - epsilon / Understanding the Scikit-learn SVM implementation
    - penalty / Understanding the Scikit-learn SVM implementation
    - loss / Understanding the Scikit-learn SVM implementation
    - dual / Understanding the Scikit-learn SVM implementation
- sknn
    - parallelizing / Parallelization for sknn
- sknn library
    - about / The sknn library
    - URL / The sknn library
- sknn package
    - URL / Neural networks in action
- sobel filter
    - about / How to do it…
    - URL / How to do it…
- SofiaML
    - about / Other alternatives for SVM fast learning
    - URL / Other alternatives for SVM fast learning
- softmax
    - for classification / The neural network architecture
- solid squares / Getting ready
- Spark
    - about / Spark, Machine learning with Spark
    - pySpark / pySpark
    - on KDD99 dataset / Spark on the KDD99 dataset
- Spark, methods
    - map(function) / pySpark
    - flatMap(function) / pySpark
    - filter(function) / pySpark
    - sample(withReplacement, fraction, seed) / pySpark
    - distinct() / pySpark
    - coalesce(numPartitions) / pySpark
    - repartition(numPartitions) / pySpark
    - groupByKey() / pySpark
    - reduceByKey(function) / pySpark
    - sortByKey(ascending) / pySpark
    - union(otherRDD) / pySpark
    - intersection(otherRDD) / pySpark
    - join(otherRDD) / pySpark
- Spark DataFrames
    - working with / Working with Spark DataFrames
- sparse autoencoder
    - URL / Autoencoders

- sparse PCA
  - about / Sparse PCA
- sparsity parameters
  - about / Autoencoders
- speech recognizer
  - about / Introduction
  - building / Building a speech recognizer, How to do it…
- Spyder
  - about / Scientific distributions
- SQLite
  - reference link / Working with databases
- stacked denoising autoencoders
  - deep learning / Autoencoders
- stacked denoising autoencoders (SdA)
  - about / Stacked Denoising Autoencoders
  - applying / Applying the SdA
  - performance, assessing / Assessing SdA performance
- stacking ensembles
  - using / Using stacking ensembles
- standalone machine
  - big data, handling / From a standalone machine to a bunch of nodes
  - distributed framework, need for / Why do we need a distributed framework?
- Star feature detector
  - building / Building a Star feature detector, How to do it…
- statistics
  - extracting, from time series data / Extracting statistics from time series data, How to do it…
- steepest descent
  - about / Gradient Boosting Machines
- stemming
  - about / Stemming
- stochastic gradient boosting
  - about / Stochastic gradient boosting and gridsearch on H2O
  - URL / Stochastic gradient boosting and gridsearch on H2O
- Stochastic Gradient Descent (SGD)
  - about / Implementing self-training
- stochastic gradient descent (SGD) / Stochastic gradient descent
- stochastic learning
  - about / Stochastic learning
  - batch gradient descent / Batch gradient descent
  - stochastic gradient descent (SGD) / Stochastic gradient descent
  - Scikit-learn SGD implementation / The Scikit-learn SGD implementation
  - SGD learning parameters, defining / Defining SGD learning parameters
- stock market data
  - patterns, finding in / Finding patterns in stock market data, How to do it…
  - analyzing, hidden Markov models used / Analyzing stock market data using Hidden Markov Models, How to do it…

- stream handling
    - reference link / Datasets to try the real thing yourself
- stride
    - about / Understanding convolution layers, The convolution layer
- subsample
    - URL / Extremely randomized trees and large datasets
- subsampling
    - nonlinear SVMs, pursuing / Pursuing nonlinear SVMs by subsampling
- subsampling layer
    - about / The pooling layer
- subtaggers
    - about / Backoff tagging
- sum-pooling
    - about / Understanding pooling layers
- Support Vector Classification (SVC)
    - about / Recursive Feature Elimination
- support vector machine / How to do it…
- Support Vector Machine (SVMs)
    - used, for building linear classifier / Building a linear classifier using Support Vector Machine (SVMs), Getting ready, How to do it…
    - references, for tutorials / Building a linear classifier using Support Vector Machine (SVMs)
    - used, for building nonlinear classifier / Building a nonlinear classifier using SVMs, How to do it…
- Support Vector Machines (SVMs)
    - about / Support Vector Machines
    - hinge loss / Hinge loss and its variants
    - Scikit-learn / Understanding the Scikit-learn SVM implementation
    - nonlinear SVMs, pursuing / Pursuing nonlinear SVMs by subsampling
    - achieving, with SGD / Achieving SVM at scale with SGD
- support vector machines (SVMs) / The Scikit-learn SGD implementation

T

- tagging
    - with, Natural Language Toolkit (NTLK) / Tagging with NLTK
    - sequential tagging / Sequential tagging
    - backoff tagging / Backoff tagging
- tanh
    - about / The neural network architecture
- TB-scale datasets
    - about / Clustering – a primer
- TDM-GCC x64
    - URL / Theano
- tensor / Understanding convolution layers
- TensorFlow
    - about / Introduction to TensorFlow, Getting to know TensorFlow, TensorFlow
    - using / Using TensorFlow to iteratively improve our models

- URL / TensorFlow
- references / TensorFlow
- installing / TensorFlow installation, Keras and TensorFlow installation
- operations / TensorFlow operations
- machine learning, with SkFlow / Machine learning on TensorFlow with SkFlow
- convolutional neural networks (CNN), through Keras / Convolutional Neural Networks in TensorFlow through Keras

- TensorFlow, operations
  - GPU, computing / GPU computing
  - linear regression, with SGD / Linear regression with SGD
  - neural network, performing / A neural network from scratch in TensorFlow
- TensorFlow library
  - about / Understanding convolution layers
- term frequency (TF) / Building a text classifier, How it works…
- text
  - converting, to base form with lemmatization / Converting text to its base form using lemmatization, How to do it…
  - dividing, chunking used / Dividing text using chunking, How to do it…
  - patterns, identifying with topic modeling / Identifying patterns in text using topic modeling, How to do it…, How it works…
- text analysis
  - about / Introduction
- text classifier
  - building / Building a text classifier, How to do it…
- text data
  - stemming / Stemming text data, How it works…
  - cleaning / Cleaning text data
  - cleaning, with BeautifulSoup / Text cleaning with BeautifulSoup
  - punctuation, managing / Managing punctuation and tokenizing
  - tokenisation, managing / Managing punctuation and tokenizing
  - words, categorizing / Tagging and categorising words
  - words, tagging / Tagging and categorising words
  - features, creating / Creating features from text data
- text feature engineering
  - about / Text feature engineering
  - text data, cleaning / Cleaning text data
  - stemming / Stemming
  - bagging / Bagging and random forests
  - random forests / Bagging and random forests
  - prepared data, testing / Testing our prepared data
- tf-idf
  - about / Building a text classifier
  - URL / Building a text classifier
- theanets
  - about / Theanets, Deep learning with theanets
  - URL / Theanets, Deep learning with theanets
  - neural network, on GPU / Deep learning with theanets

- Theano
  - about / Denoising autoencoders, Theano
  - URL / Theano, Theano – parallel computing on the GPU, Installing Theano
  - URL, for installing / Theano
  - used, for parallel computing on GPU / Theano – parallel computing on the GPU
  - installing / Installing Theano
- time series data
  - about / Introduction
  - slicing / Slicing time series data, How to do it…
  - operating on / Operating on time series data, How to do it…
  - statistics, extracting from / Extracting statistics from time series data, How to do it…
- time series format
  - data, transforming into / Transforming data into the time series format, How to do it…
- tokenisation
  - about / Managing punctuation and tokenizing
- tokenization
  - about / Preprocessing data using tokenization
  - used, for preprocessing data / Preprocessing data using tokenization, How to do it…
- topic modeling
  - about / Identifying patterns in text using topic modeling
  - used, for identifying patterns in text / Identifying patterns in text using topic modeling, How to do it…, How it works…
- traffic
  - estimating / Estimating traffic, How to do it…
- training_classification_error (.089)
  - about / Large scale deep learning with H2O
- transforming autoencoder
  - about / Understanding pooling layers
- translation-invariance
  - about / Understanding pooling layers
- Translink Twitter
  - about / Translink Twitter
- trigram tagger
  - about / Sequential tagging
- Twitter
  - using / Twitter
  - Translink Twitter, using / Translink Twitter
  - consumer comments, analyzing / Consumer comments
  - Bing Traffic API / The Bing Traffic API

U

- U-Matrix
  - about / Employing SOM
- UCI Machine Learning Repository / Datasets to try the real thing yourself
- Uniform Resource Identifier (URI)
  - about / HDFS
- unigram tagger

    - about / Sequential tagging
- universal approximation theorem
    - about / What and how neural networks learn
- University of California, Irvine (UCI) / Datasets to try the real thing yourself
- unsupervised learning
    - about / Introduction
    - autoencoders / Autoencoders and unsupervised learning
- unsupervised methods
    - about / Unsupervised methods
- unsupervised pretraining
    - about / Deep learning and unsupervised pretraining
- US Census dataset
    - URL / Scaling K-means – mini-batch

V

- v-fold cross-validation
    - about / Tuning your clustering configurations
- Vagrant
    - about / Vagrant
    - URL / Vagrant
- validation curves
    - extracting / Extracting validation curves, How to do it…
- validation_classification_error (.0954)
    - about / Large scale deep learning with H2O
- validity measure (v-measure)
    - about / Kick-starting clustering analysis
- vanishing gradient problem
    - about / The neural network architecture
- variables
    - sharing, across cluster nodes / Sharing variables across cluster nodes
- vector quantization
    - about / Compressing an image using vector quantization, How to do it…, Creating a vector quantizer
    - used, for compressing image / Compressing an image using vector quantization, How to do it…
    - reference / Compressing an image using vector quantization, How to do it…
    - used, for creating features / Creating features using visual codebook and vector quantization, How to do it…
- vector quantizer
    - creating / Creating a vector quantizer, How to do it…
- video
    - capturing, from webcam / Capturing and processing video from a webcam, How to do it…
    - processing, from webcam / Capturing and processing video from a webcam, How to do it…
- VirtualBox
    - about / VirtualBox

    - URL / VirtualBox
- virtual machine
    - setting up / Setting up the VM for this chapter
- virtual machines (VM)
    - setting up / Setting up the VM
    - VirtualBox / VirtualBox
    - about / VirtualBox
    - Vagrant / Vagrant
    - using / Using the VM
- visual codebook
    - used, for creating features / Creating features using visual codebook and vector quantization, How to do it…
    - about / Creating features using visual codebook and vector quantization
    - references / Creating features using visual codebook and vector quantization
- Vowpal Wabbit (VW)
    - about / Scale up with Python, Nonlinear and faster with Vowpal Wabbit
    - installing / Installing VW
    - URL, for compiling / Installing VW
    - data format / Understanding the VW data format
    - URL, for dataset / Understanding the VW data format
    - Python, integration / Python integration
    - examples / A few examples using reductions for SVM and neural nets
    - URL, for neural networks / A few examples using reductions for SVM and neural nets
    - faster bike-sharing example / Faster bike-sharing
    - covertype dataset / The covertype dataset crunched by VW
- vowpal_porpoise / Python integration

W

- Wabbit Wappa / Python integration
- weak learners / Getting ready
- weather API
    - creating / The weather API
- Whitening
    - about / Machine learning on TensorFlow with SkFlow
- wholesale vendor and customers
    - reference / Building a customer segmentation model
- wide networks
    - about / The hidden layer
- WinPython
    - URL / Scientific distributions
    - about / Scientific distributions
- word2vec
    - about / Gensim

X

- XGBoost

    - about / Scale up with Python, XGBoost
    - URL / XGBoost
    - URL, for installing / XGBoost
    - URL, for code / XGBoost

Y

- Yahoo Weather API
    - about / Acquiring data via RESTful APIs
- Yet Another Resource Negotiator (YARN)
    - about / Scale out with Python, YARN

Z

- zero-padding
    - about / The convolution layer
- Zipf distribution
    - about / Reinterpreting non-numeric features