MANNING

Machine Learning Systems

Designs that scale

Jeff Smith

Foreword by Sean Owen

Machine Learning Systems

Designs that scale

JEFF SMITH

Foreword by SEAN OWEN

MANNING
Shelter Island

For online information and ordering of this and other Manning books, please visit www.manning.com. The publisher offers discounts on this book when ordered in quantity. For more information, please contact

Special Sales Department
Manning Publications Co.
20 Baldwin Road
PO Box 761
Shelter Island, NY 11964
Email: orders@manning.com

Manning Publications Co.
20 Baldwin Road
PO Box 761
Shelter Island, NY 11964

Development editor:	Susanna Kline
Review editor:	Aleksandar Dragosavljević
Technical development editor:	Kostas Passadis
Project editor:	Tiffany Taylor
Copyeditor:	Corbin Collins
Proofreader:	Katie Tennant
Technical proofreader:	Jerry Kuch
Typesetter:	Gordan Salinovic
Cover designer:	Marija Tudor

ISBN 9781617293337
Printed in the United States of America
1 2 3 4 5 6 7 8 9 10 – EBM – 23 22 21 20 19 18

brief contents

contents

foreword

Today's data scientists and software engineers are spoiled for choice when looking for tools to build machine learning systems. They have a range of new technologies that make it easier than ever to build entire machine learning systems. Considering where we—the machine learning community—started, it's exciting to see a book that explores how powerful and approachable the current technologies are.

To better understand how we got here, I'd like to share a bit of my own story. They tell me I'm a data scientist, but I think I'm only here by accident. I began as a software person and grew up on Java 1.3 and EJB. I left the software-engineer role at Google a decade ago, although I dabbled in open source and created a recommender system that went on to be part of Apache Mahout in 2009. Its goal was to implement machine learning algorithms on the then-new Apache Hadoop MapReduce framework. The engineering parts were familiar—MapReduce came from Google, after all. The machine learning was new and exciting, but the tools were lacking.

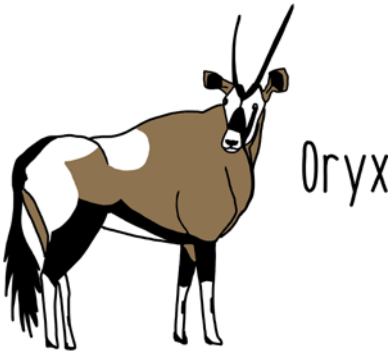
Not knowing any better, and with no formal background in ML, I tried to help build ML at scale. In theory, this was going to open an era of better ML, because more data generally means better models. ML just needed tooling rebuilt on the nascent distributed computing platforms like Hadoop.

Mahout (0.x) was what you'd expect when developers with a lot of engineering background and a little stats background try to build ML tools: JVM-based, modular, scalable, complex, developer-oriented, baroque, and sometimes eccentric in its interpretation of stats concepts. In retrospect, classic Mahout wasn't interesting because it was a better version of stats tooling. In truth, it was much less usable than, say, R (which I admit having never heard of until 2010). Mahout was interesting, because it was built from the beginning to work at web scale, using tooling developed for enterprise software engineering. The collision of stats tooling with new approaches to handling web-scale data gave birth to what became known as data science.

The more I back-filled my missing context about how real statisticians and analysts had been successfully applying ML for decades, thank you very much, the more I realized that the existing world of analytics tooling optimizes for some usages and not others. Python, R, and their ecosystems have rich analytics libraries and visualization tools. They're not as concerned with issues of scale or production deployment.

Coming from an enterprise software world, I was somewhat surprised that the tooling generally ended at building a model. What about doing something with the model in production? I found this was usually viewed as a separate activity for software engineers to undertake. The engineering community hadn't settled on clear patterns for product application around Hadoop-related technologies.

In 2012, I spun out a small company, Myrrix, to expand on the core premise of Mahout and make it into a continuously learning, updating service with the ability to serve results from the model in production—not just a library that output coefficients. This became part of Cloudera and was reimagined again, on top of Apache Spark, as Oryx (https://github.com/OryxProject/oryx).



Spark was another game changer for the Hadoop ecosystem. It brought a higher-level, natural functional paradigm to big data software development, more like you'd encounter in Python. It added language bindings to Python and R. It brought a new machine learning library, Spark MLlib. By 2015, the big data ecosystem at large was suddenly much closer to the world of conventional analytics tools.

These and other tools have bridged the worlds of stats and software engineering such that the two now interact regularly. Today's big data engineer has ready access to Python-only tooling like TensorFlow for deep learning and Seaborn for visualization. The software-engineering culture of version control and testing and strongly typed languages has flowed into the data science community, too.

That brings us back to this book. It doesn't cover just tools but also the entire job of building a machine learning system. It gets into topics that people used to gloss over, like model serialization and building model servers. The language of the book is primarily Scala, a unique language that is both principled and expressive without sacrificing conveniences like type inference. Scala has been used to build powerful technologies like Spark and Akka, which the book shows you how to use to build machine learning systems. The book also doesn't ignore the importance of interoperability with Python technologies or portable application builds with Docker.

We've come a long way, and there's farther to go. The person who can master the tools and techniques in this book will be well prepared to play a role in machine learning's even more exciting future.

Sean Owen
Director of Data Science, Cloudera

preface

I've been working with data for my entire professional career. Following my interests, I've worked on ever-more-analytically sophisticated systems as my career has progressed, leading to a focus on machine learning and artificial intelligence systems.

As my work content evolved from more traditional data-warehousing sorts of tasks to building machine learning systems, I was struck by a strange absence. When I was working primarily with databases, I could rely on the rich body of academic and professional literature about how to build databases and applications that interact with them to help me define what a good design was. So, I was confused and surprised to find that machine learning as a field generally lacked this sort of guidance. There were no canonical implementations of anything other than the model learning algorithms. Huge chunks of the system that needed to be built were largely glossed over in the literature. Often, I couldn't even find a consistent name for a given system component, so my colleagues and I inevitably confused each other with our choices of terminology.

What I wanted was a framework, something like a Ruby on Rails for machine learning, but no such framework seemed to exist.[1] Barring a commonly accepted framework, I wanted at least some clear design patterns for how to build machine learning systems; but alas, there was no *Design Patterns for Machine Learning Systems* to be found, either.

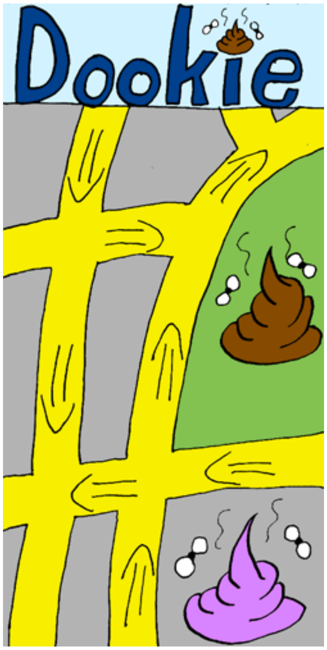
[1] Eventually, I came across Sean Owen's work on Oryx and Simon Chan's on PredictionIO, which were super-instructive. If you're interested in the background of machine learning architectures, you'll benefit from reviewing them both.

So, I built machine learning systems the hard way: by trying things and figuring out what didn't work. When I needed to invent terminology, I just picked reasonable terms. Over time, I tried to synthesize some of my learnings about what worked for machine learning system design and what didn't into a coherent whole. Fields like distributed systems and functional programming offered the promise of adding coherence to my views about machine learning systems, but neither was particularly focused on application to machine learning.

Then, I discovered reactive systems design, via reading the Reactive Manifesto (www.reactivemanifesto.org). It was startling in its simple coherence and bold mission. Here was a complete world view of what the challenge of building modern software applications was *and* a principled way of building applications that met that challenge. I was excited by the promise of the approach and immediately began attempting to apply it to the problems I'd seen in architecting and building machine learning systems.

This inquiry led me to poop—specifically, to dog poop. I tried to imagine how a naive machine learning system could be refactored into something much better, using the tools from reactive systems design. To do this, I wrote a blog post about a dog poop prediction startup (http://mng.bz/9YK8; see figure).

The post got a surprisingly large and serious response from a wide range of people. I learned two things from that response:



Poop prediction

- I wasn't the only one interested in coming up with a principled approach to building machine learning systems.
- People really enjoyed talking about machine learning in terms of cartoon animals.

Those insights led to the book you're reading. In this book, I try to cover a range of issues you're likely to encounter in building real-world machine learning systems that have to keep customers happy. My focus is on all the stuff you won't find in other books. I've tried to make the book as broad as possible, in the hopes of covering the full responsibilities of the modern data scientist or engineer. I explore how to use general principles and techniques to break down the seemingly unique problems of a given component of a machine learning system. My goal is to be as comprehensive as possible in my coverage of machine learning system components, but that means I can't be comprehensive on huge topics like model learning algorithms and distributed systems. Instead, I've designed examples that provide you with experience building various components of a machine learning system.

I firmly believe that to build a truly powerful machine learning system, you must take a system-level view of the problem. In this book, I provide that high-level perspective and then help you build skills around each of the key components in that system. I learned through my experience as a technical lead and manager that understanding the entire machine learning system and the composition of its components is one of the most important skills a developer of machine learning systems can have. So, the book tries to cover all the different pieces it takes to build up a powerful, real-world machine learning system. Throughout, we'll take the perspective of teams shipping sophisticated machine learning systems for live users. So, we'll explore how to build *everything* in a machine learning system. It's a big job, and I'm excited that you're interested in taking it on.

acknowledgments

A book is the opposite of an academic paper when it comes to attribution. In an academic paper, everyone who ever even grabbed lunch at the lab can get their name on the paper; but in a book, for some reason, we only put one or two names on the cover. But it's not that simple to pull a book together; lots of people are involved. Here are all the people who made this book happen.

As I mentioned in the preface, the book grew out of (believe it or not) a blog post about dog poop (http://mng.bz/9YK8). I'm immensely grateful to the serious and accomplished people who took my cartoons about dog poop seriously enough to provide useful feedback: Roland Kuhn, Simon Chan, and Sean Owen.

In the early days of the book, the members of the reactive study group and the data team at Intent Media were invaluable in helping me understand where I was trying to take these ideas about building machine learning systems. I'm also indebted to Chelsea Alburger from Intent Media, who provided great early art direction for the book's visuals.

Thanks go to the team at Manning who took my original ideas and helped them become a book: Frank Pöhlmann, who suggested that there might be a book in this reactive machine learning stuff; Susanna Kline, who dragged me kicking and screaming through the dark forest; Kostas Passadis, who kept me from looking like a complete fool; and Marjan Bace, who green-lit the whole mad endeavor. I also want to thank the technical peer reviewers, led by Aleksandar Dragosavljevic: David Andrzejewski, Jose Carlos Estefania Aulet, Óscar Belmonte-Fernández, Tony M. Dubitsky, Vipul Gupta, Jason Hales, Massimo Ilario, Shobha Iyer, Shanker Janakiraman, Jon

Lehto, Anuja Kelkar, Alexander Myltsev, Tommy O'Dell, Jean Safar, José San Leandro, Jeff Smith, Chris Snow, Ian Stirk, Fabien Tison, Jeremy Townson, Joseph Wang, and Jonathan Woodard.

Once the book really got rolling, the team at x.ai were immensely helpful in providing a test lab for various ideas and supporting me as I took the book's ideas on the road in the form of talks. I thank you, Dennis Mortensen, Alex Poon, and everyone on the tech team.

Also, thanks go to anyone who came out to hear one of the talks associated with the book at conferences and meetups. All the feedback provided, in person and online, was instrumental to helping me understand how the material was evolving.

Finally, I thank my illustrator, yifan, without whom the book wouldn't have been possible. You've brought to life my vision of cartoon animals who do machine learning, and now I'm excited to be able to share it with the world.

P.S. Thanks to my muse: nom nom, the data dog. Who's a good little machine learner? You are!

about this book

This book serves two slightly different audiences. First, it serves software engineers who are interested in machine learning but haven't built many real-world machine learning systems. I presume such readers want to put their skills into practice by actually building something with machine learning. The book is different from other books you may have picked up on machine learning. In it, you'll find techniques applicable to building whole production-grade systems, not just naive scripts. We'll explore the entire range of possible components you might need to implement in a machine learning system, with lots of hard-won tips about common design pitfalls. Along the way, you'll learn about the various jobs of a machine learning system, in the context of implementing systems that fulfill those needs. So, if you don't have a lot of background in machine learning, don't worry that you'll have to wade through pages of math before you get to build things. The book will have you coding all the way through, often relying on libraries to handle the more complex implementation concerns like model learning algorithms and distributed data processing.

Second, this book serves data scientists who are interested in the bigger picture of machine learning systems. I presume that such readers know the concepts of machine learning but may only have implemented simple machine learning functionality (for example, scripts over files on a laptop). For such readers, the book may introduce you to a range of concerns that you've never before considered part of the work of machine learning. In places, I'll introduce vocabulary to name components of a system that are often neglected in academic machine learning discussions, and then I'll show you how to implement them. Although the book does get into some powerful programming techniques, I don't presume that you have deep experience in software engineering, and I'll introduce all concepts beyond the very basic, in context.

For either type of reader, I assume that you have some interest in reactive systems and how this approach can be used to build better machine learning systems. The reactive perspective on system design underpins every part of the book, so you'll spend a lot of time examining the properties your system has or doesn't have, often presuming that real-world problems like server outages and network partitions will occur in your system.

Concretely, this focus on reactive systems means the book contains a fair bit of material on distributed systems and functional programming. The goal of unifying these concerns with the task of building machine learning systems is to give you tools to solve some of the hardest problems in technology today. Again, if you don't have a background in distributed systems or functional programming, don't worry: I'll introduce this material in context with the appropriate motivation. Once you see tools like Scala, Spark, and Akka in action, I hope it will become clear to you how helpful they can be in solving real-world machine learning problems.

How this book is organized

This book is organized into three parts. Part 1 introduces the overall motivation of the book and some of the tools you'll use:

- Chapter 1 introduces machine learning, reactive systems, and the goals of reactive machine learning.
- Chapter 2 introduces three of the technologies the book uses: Scala, Spark, and Akka.

Part 2 forms the bulk of the book. It proceeds component by component, helping you to deeply understand all the things a machine learning system must do, and how you can do them better using reactive techniques:

- Chapter 3 discusses the challenges of collecting data and ingesting it into a machine learning system. As part of that, it introduces various concepts around handling uncertain data. It also goes into detail about how to persist data, focusing on properties of distributed databases.
- Chapter 4 gets into how you can extract features from raw data and the various ways in which you can compose this functionality.
- Chapter 5 covers model learning. You'll implement your own model learning algorithms and use library implementations. It also covers how to work with model learning algorithms from other languages.
- Chapter 6 covers a range of concerns related to evaluating models once they've been learned.
- Chapter 7 shows how to take learned models and make them available for use. In the service of this goal, this chapter introduces Akka HTTP, microservices, and containerization via Docker.

- Chapter 8 is all about using machine learned models to act on the real world. It also introduces an alternative to Akka HTTP for building services: http4s.

Finally, part 3 introduces a few more concerns that become relevant once you've built a machine learning system and need to keep it running and evolve it into something better:

- Chapter 9 shows how to build Scala applications using SBT. It also introduces concepts from continuous delivery.
- Chapter 10 shows how to build artificially intelligent agents of various levels of complexity as an example of system evolution. It also covers more techniques for analyzing the reactive properties of a machine learning system.

How should you read this book? If you have good experience in Scala, Spark, and Akka, then you might skip chapter 2. The heart of the book is the journey through the various system components in part 2. Although they're meant to stand alone as much as possible, it will probably be easiest to follow the flow of the data through the system if you proceed in order from chapter 3 through chapter 8. The final two chapters are separate concerns and can be read in any order (after you've read part 2).

Code conventions and downloads

This book contains many examples of source code, both in numbered listings and in line with normal text. In both cases, source code is formatted in a `fixed-width font like this` to separate it from ordinary text.

In many cases, the original source code has been reformatted; we've added line breaks and reworked indentation to accommodate the available page space in the book. In rare cases, even this was not enough, and listings include line-continuation markers (➥). Additionally, comments in the source code have often been removed from the listings when the code is described in the text. Code annotations accompany many of the listings, highlighting important concepts.

The code used in the book can be found on the book's website, www.manning.com/books/machine-learning-systems, and in this Git repository: http://github.com/jeffreyksmithjr/reactive-machine-learning-systems.

Book forum

Purchase of *Machine Learning Systems* includes free access to a private web forum run by Manning Publications where you can make comments about the book, ask technical questions, and receive help from the author and from other users. To access the forum, go to https://forums.manning.com/forums/machine-learning-systems. You can also learn more about Manning's forums and the rules of conduct at https://forums.manning.com/forums/about.

Manning's commitment to our readers is to provide a venue where a meaningful dialogue between individual readers and between readers and the author can take place. It is not a commitment to any specific amount of participation on the part of the author, whose contribution to the forum remains voluntary (and unpaid). We suggest you try asking him some challenging questions lest his interest stray! The forum and the archives of previous discussions will be accessible from the publisher's website as long as the book is in print.

Other online resources

- For more information about Scala and pointers to various resources on how to learn the language, the language website is the best place to start: www.scala-lang.org.
- The Spark project site contains excellent documentation and pointers to other useful Spark-related resources: http://spark.apache.org.
- Similarly, the Akka project website has invaluable documentation and links to other useful resources: http://akka.io.
- The Reactive Manifesto was the starting point for the recent focus on reactive systems: www.reactivemanifesto.org.
- I maintain a site related to the book. It gathers together talks and other resources about reactive machine learning: http://reactivemachinelearning.com.

about the author

Jeff Smith builds powerful machine learning systems. For the past decade, he has been working on building data science applications, teams, and companies as part of various teams in New York, San Francisco, and Hong Kong. He blogs (https://medium.com/@jeffksmithjr), tweets (jeffksmithjr), and speaks (www.jeffsmith.tech/speaking) about various aspects of building real-world machine learning systems.

about the cover illustration

The figure on the cover of *Machine Learning Systems* is captioned "Japanese Infantry—Musician." The illustration is taken from Auguste Wahlen's *Manners, Customs and Costumes of All Peoples of the World*, a four-volume set published in Brussels by the Librairie Historique in 1843. The books contain scarce and beautiful images: the costumes of the world are finely illustrated with a wonderful series of highly colored full-page plates.

Wahlen, whose real name was Jean-François-Nicolas Loumyer (1801–1875), was an archivist for the heraldic authority of Belgium. Fascination with faraway lands and travel for pleasure were relatively new phenomena in the early nineteenth century, and collections such as this one were popular, introducing both the tourist as well as the armchair traveler to the inhabitants of other countries.

The diversity of the drawings in Wahlen's volumes speaks vividly of the uniqueness and individuality of the world's nations some 200 years ago. Dress codes have changed since then, and the diversity by region and country, so rich at the time, has faded away. It's now often hard to tell the inhabitants of one continent from another. Perhaps, trying to view it optimistically, we've traded a cultural and visual diversity for a more varied personal life—or a more varied and interesting intellectual and technical life.

At a time when it's difficult to tell one computer book from another, Manning celebrates the inventiveness and initiative of the computer business with book covers based on the rich diversity of regional life of two centuries ago, brought back to life by Wahlen's pictures.

Part 1

Fundamentals of reactive machine learning

Reactive machine learning brings together several different areas of technology, and this part of the book is all about making sure you're sufficiently oriented in all of them. Throughout this book, you'll be looking at and building machine learning systems, starting with chapter 1. If you don't have experience with machine learning, it's important to be familiar with some of the basics of how it works. You'll also get a flavor for all of the problems with how machine learning systems are often built in the real world. With this knowledge in hand, you'll be ready for another big topic: reactive systems design. Applying the techniques of reactive systems design to the challenges of building machine learning systems is the core topic of this book.

After you've had an overview of *what* you're going to do in this book, chapter 2 focuses on *how* you'll do it. The chapter introduces three technologies that you'll use throughout the book: the Scala programming language, the Akka toolkit, and the Spark data-processing library. These are powerful technologies that you can only begin to learn in a single chapter. The rest of the book will go deeper into how to use them to solve real problems.

1 Learning reactive machine learning

This chapter covers

- Introducing the components of machine learning systems
- Understanding the reactive systems design paradigm
- The reactive approach to building machine learning systems

This book is all about how to build *machine learning systems*, which are sets of software components capable of learning from data and making predictions about the future. This chapter discusses the challenges of building machine learning systems and offers some approaches to overcoming those challenges. The example we'll look at is of a startup that tries to build a machine learning system from the ground up and finds it very, very hard.

If you've never built a machine learning system before, you may find it challenging and a bit confusing. My goal is to take some of the pain and mystery out of this process. I won't be able to teach you everything there is to know about the techniques of machine learning; that would take a mountain of books. Instead, we'll focus on how to build a system that can put the power of machine learning to use.

I'll introduce you to a fundamentally new and better way of building machine learning systems called *reactive machine learning*. Reactive machine learning represents the marriage of ideas from reactive systems and the unique challenges of machine learning. By understanding the principles that govern these systems, you'll see how to build systems that are more capable, both as software and as predictive systems. This chapter will introduce you to the motivating ideas behind this approach, laying a foundation for the techniques you'll learn in the rest of the book.

1.1 An example machine learning system

Consider the following scenario. Sniffable is "Facebook for dogs." It's a startup based out of a dog-filled loft in New York. Using the Sniffable app, dog owners post pictures of their dogs, and other dog owners like, share, and comment on those pictures. The network was growing well, and the team felt there might be a meteoric opportunity here. But if Sniffable was really going to take off, it was clear that they'd have to build more than just the standard social-networking features.

1.1.1 Building a prototype system

Sniffable users, called *sniffers*, are all about promoting their specific dog. Many sniffers hope that their dog will achieve canine celebrity status. The team had an idea that what sniffers really wanted were tools to help make their posts, called *pupdates*, more viral. Their initial concept for the new feature was a sort of competitive intelligence tool for the canine equivalent of stage moms, internally known as *den mothers*. The belief was that den mothers were taking many pictures of their dogs and were trying to figure out which picture would get the biggest response on Sniffable. The team intended the new tool to predict the number of likes a given pupdate might get, based on the hashtags used. They named the tool *Pooch Predictor*. It was their hope that it would engage the den mothers, help them create viral content, and grow the Sniffable network as a whole.

The team turned to their lone data scientist to get this product off the ground. The initial spec for the minimal viable product was pretty fuzzy, and the data scientist was already a pretty busy guy—he was the entire data science department, after all. Over the course of several weeks, he stitched together a system that looked something like figure 1.1.

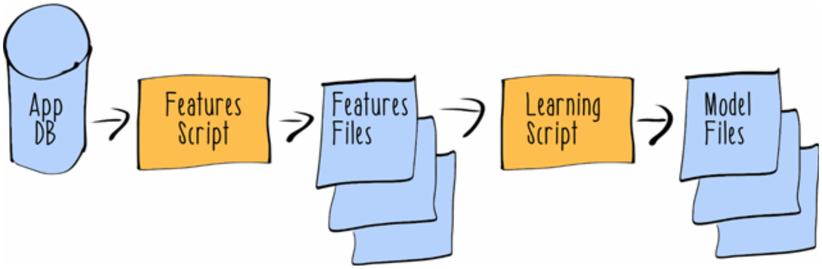


Figure 1.1 Pooch Predictor 1.0 architecture

The app already sent all raw user-interaction data to the application's relational database, so the data scientist decided to start building his model with that data. He wrote a simple script that dumped the data he wanted to flat files. Then he processed that interaction data using a different script to produce derived representations of the data, the features, and the concepts. This script produced a structured representation of a pupdate, the number of likes it got, and other relevant data such as the hashtags associated with the post. Again, this script just dumped its output to flat files. Then he ran his model-learning algorithm over his files to produce a model that predicted likes on posts, given the hashtags and other data about the post.

The team was thoroughly amazed by this prototype of a predictive product, and they pushed it through the engineering roadmap to get it out the door as soon as possible. They assigned a junior engineer the job of taking the data scientist's prototype and getting it running as a part of the overall system. The engineer decided to embed the data scientist's model directly into the app's post-creation code. That made it easy to display the predicted number of likes in the app.

A few weeks after Pooch Predictor went live, the data scientist happened to notice that the predictions weren't changing much, so he asked the engineer about the retraining frequency of the modeling pipeline. The engineer had no idea what the data scientist was talking about. They eventually figured out that the data scientist had intended his scripts to be run on a daily basis over the latest data from the system. Every day there should be a new model in the system to replace the old one. These new requirements changed how the system needed to be constructed, resulting in the architecture shown in figure 1.2.

In this version of Pooch Predictor, the scripts were run on a nightly basis, scheduled by cron. They still dumped their intermediate results to files, but now they needed to insert their models into the application's database. And now the backend server was responsible for producing the predictions displayed in the app. It would pull the model out of the database and use it to provide predictions to the app's users.

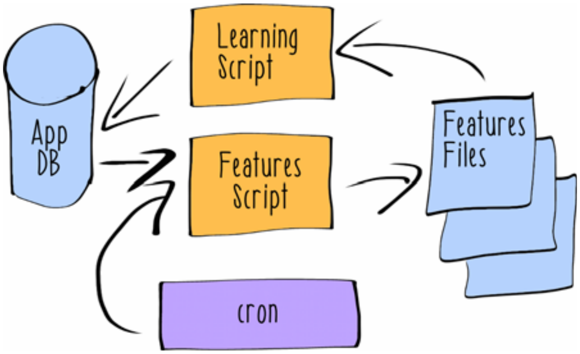


Figure 1.2 Pooch Predictor 1.1 architecture

This new system was definitely better than the initial version, but in its first several months of operation, the team discovered several pain points with it. First of all, Pooch Predictor wasn't very reliable. Often something would change in the database, and one of the queries would fail. Other times there would be high load on the server, and the modeling job would fail. This was happening more and more as both the size of the social network and the size of the dataset used by the modeling system increased. One time, the server that was supposed to be running the data-processing job failed, and all the relevant data was lost. These sorts of failures were hard to detect without building up a more sophisticated monitoring and alerting infrastructure. But even if someone did detect a failure in the system, there wasn't much that could be done other than kick off the job again and hope it succeeded this time.

Besides these big system-level failures, the data scientist started to find other problems in Pooch Predictor. Once he got at the data, he realized that some of the features weren't being correctly extracted from the raw data. It was also really hard to understand how a change to the features that were being extracted would impact modeling performance, so he felt a little blocked from making improvements to the system.

There was also a major issue that ended up involving the entire team. For a period of a couple of weeks, the team saw their interaction rates steadily trend down with no real explanation. Then someone noticed a problem with Pooch Predictor while testing on the live version of the app. For the pupdates of users who were based outside the United States, Pooch Predictor would always predict a negative number of likes. In forums around the internet, disgruntled users were voicing their rage at having the adorableness of their particular dog insulted by the Pooch Predictor feature. Once the Sniffable team detected the issue, they were able to quickly figure out that it was a problem with the modeling system's location-based features. The data scientist and engineer came up with a fix, and the issue went away, but only after having their credibility seriously damaged among sniffers located abroad.

Shortly after that, Pooch Predictor ran into more problems. It started with the data scientist implementing more feature-extraction functionality in an attempt to improve modeling performance. To do that, he got the engineer's help to send more data from the user app back to the application database. On the day the new functionality rolled out, the team saw immediate issues. For one thing, the app slowed down dramatically. Posting was now a very laborious process—each button tap seemed to take several seconds to register. Sniffers became seriously irritated with these issues. Things went from bad to worse when Pooch Predictor began to cause yet more problems with posting. It turned out that the new functionality caused exceptions to be thrown on the server, which led to pupdates being dropped.

At this point, it was all hands on deck in a furious effort to put out this fire. They realized that there were two major issues with the new functionality:

- Sending the data from the app back to the server required a transaction. When the data scientist and engineer added more data to the total amount of data being collected for modeling, this transaction took way too long to maintain reasonable responsiveness within the app.
- The prediction functionality within the server that supported the app didn't handle the new features properly. The server would throw an exception every time the prediction functionality saw any of the new features that had been added in another part of the application.

After understanding where things had gone wrong, the team quickly rolled back all of the new functionality and restored the app to a normal operational state.

1.1.2 Building a better system

Everyone on the team agreed that something was wrong with the way they were building their machine learning system. They held a retrospective to figure out what went wrong and determine how they were going to do better in the future. The outcome was the following vision for what a Pooch Predictor replacement needed to look like:

- The Sniffable app must remain responsive, regardless of any other problems with the predictive system.
- The predictive system must be considerably less tightly coupled to the rest of the systems.
- The predictive system must behave predictably regardless of high load or errors in the system itself.
- It should be easier for different developers to make changes to the predictive system without breaking things.
- The code must use different programming idioms that ensure better performance when used consistently.
- The predictive system must measure its modeling performance better.
- The predictive system should support evolution and change.
- The predictive system should support online experimentation.
- It should be easy for humans to supervise the predictive system and rapidly correct any rogue behavior.

1.2 Reactive machine learning

In the previous example, it seems like the Sniffable team missed something big, right? They built what initially looked like a useful machine learning system that added value to their core product. But all the issues they experienced in getting there obviously had a cost. Production issues with their machine learning system frequently pulled the team away from work on improvements to the capability of the system. Even though they had a bunch of smart people in the room thinking hard about how to predict the dynamics of dog-based social networking, their system repeatedly failed at its mission.

1.2.1 Machine learning

Building machine learning systems that do what they're supposed to do is *hard*, but not impossible. In our example story, the data scientist knew how to *do* machine learning. Pooch Predictor totally worked on his laptop; it made predictions from data. But the data scientist wasn't thinking of machine learning as an *application*—he only understood machine learning as a *technique*. Pooch Predictor didn't consistently produce trustable, accurate predictions. It was a failure both as a predictive system and as a piece of software.

This book will show you how to build machine learning systems that are just as awesome as the best web and mobile applications. But understanding how to build these systems will require you to think of machine learning as an application, and not merely as a technique. The systems that we'll build won't fail at their missions.

In the next section, we'll get into the reactive approach to building machine learning systems. But first I want to clarify what a machine learning system is and how it differs from merely using machine learning as a technique. To do so, I'll have to introduce some terminology. If you have experience with machine learning, some of this might seem basic, but bear with me. Terms related to machine learning can be pretty inconsistently defined and used, so I want to be explicit about what we're talking about.

Functionality vs. implementation

This brief introduction is only focused on ensuring that you're sufficiently oriented in terms of the *functionality* of a machine learning system. This book is focused on the *implementation* of machine learning systems, not on the fundamentals of machine learning itself. Should you find yourself needing a better introduction to the techniques and algorithms used in machine learning, I recommend reading *Real-World Machine Learning* by Henrik Brink, Joseph W. Richards, and Mark Fetherolf (Manning, 2016).

At its simplest, machine learning is a technique for learning from and making predictions on data. At a minimum, to *do* machine learning, you must take some data, learn a model, and use that model to make predictions. Using this definition, we can imagine an even cruder form of the Pooch Predictor example. It could be a program that queries the application database for the most popular breed of dog (French Bulldogs, it turns out) and tells the app to say that all posts containing a French Bulldog will get a lot of likes.

That minimal definition of machine learning leaves out a lot of relevant detail. Most real-world machine learning systems need to do a lot more than just that. They usually need to have all the components, or phases, shown in figure 1.3.

Starting at the beginning, a machine learning system must collect data from the outside world. In the Pooch Predictor example, the team was trying to skip this concern by using the data that their application already had. No doubt about it, that approach was quick, but it tightly coupled the Sniffable application data model to the

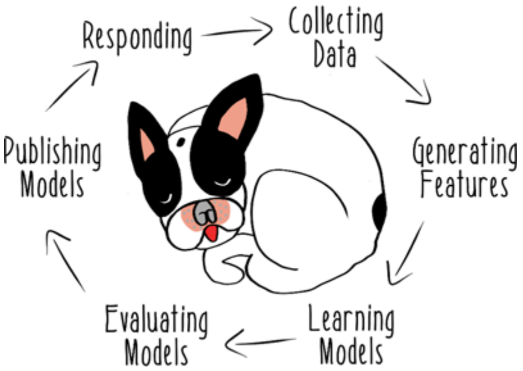


Figure 1.3 Phases of machine learning

Pooch Predictor data model. How to collect and persist data for a machine learning system is a large and important topic, so I'll spend all of chapter 3 showing you how to set up your system for success.

Once the system has data in it, that data is rarely ready to send off to a machine learning algorithm. Most machine learning algorithms are applied to derived representations of the raw data, called *instances*. Figure 1.4 shows the parts of an instance in a common syntax (LIBSVM).

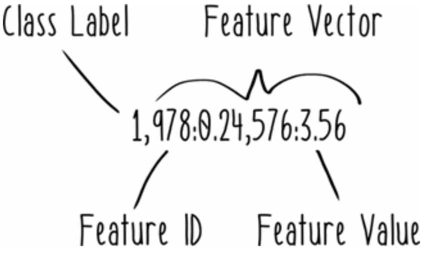


Figure 1.4 The structure of an instance

Many different syntaxes can be used to express instances, so we're not going to worry too much about the specifics of any particular syntax. However they're expressed, instances are always made up of the same components.

Features are meaningful data points derived from raw data related to the entity being predicted on, at the time you're trying to make a prediction. A Sniffable example of a feature would be the number of friends a given dog has. In figure 1.4, features are expressed using a unique ID field and feature value. Feature number 978, which might represent the sniffer's proportion of friends that are male dogs, has a value of 0.24. Typically, a machine learning system will extract many features from the raw data available to it. The feature values for a given instance are collectively called a *feature vector*.

A *concept* is the thing that the system is trying to predict. In the context of Pooch Predictor, a concept would be the number of likes a given post receives. When a concept is *discrete* (not continuous), it can be called a *class label*, and you'll often see just the word *label* used in the relevant parts of machine learning libraries, such as MLlib, which we'll use in this book.

Only some sorts of machine learning problems involve having concepts available in the form of class labels. This sort of machine learning context is known as *supervised learning*, and most of the material in this book is focused on this type of machine learning problem, although reactive machine learning could be applied to unsupervised learning problems as well.

Defining and implementing the best features and concepts to represent the problem you're trying to solve make up an enormous portion of the work of real-world machine learning. From an application perspective, these tasks are the beginning of your data pipeline. Constructing pipelines that do this job reliably, consistently, and scalably requires a principled approach to application architecture and programming style. Chapter 4 is devoted to discussing the reactive approach to this part of machine learning systems under the banner of feature generation.

Using the data prepared as just described, you're now ready to learn a model. You can think of a *model* as a program that maps from features to predicted concepts, as shown in the simple Scala implementation in the following listing.

Listing 1.1 A simple model

```
def genericModel(f: FeatureVector[RawData]): Prediction[Concept] = ???
```

Learning models occurs during the latter half of the data pipeline. A model produced by Pooch Predictor would be a program that takes as input the feature representation of the hashtag data and returns the predicted number of likes that a given pupdate might receive, as shown in the following listing.

Listing 1.2 A Pooch Predictor model

```
def poochPredictorModel(f: FeatureVector[Hashtag]): Prediction[Like] = ???
```

During this same phase of the pipeline, you'll need to begin to address several different types of uncertainty that crop up in model building. As a result, the model-learning phase of the pipeline is concerned with more than just learning models. In chapter 5, I discuss the various concerns that you'll need to consider in the model-learning subsystem of a machine learning system.

Next, you'll need to take this model and make it useful by publishing it. *Model publishing* means making the model program available outside of the context it was learned in, so that it can make predictions on data it hasn't seen before. It's easy to gloss over the difficulties that come up in this part of a machine learning system, and the Sniffable team largely skipped it in their original implementation. They didn't even set up their system to retrain the model on a regular basis. Their next approach at implementing model retraining also ran into difficulty, causing their models to be out of sync with their feature extractors. There are better ways of doing this (hint: think immutability), and I discuss them in chapter 6.

Finally, you'll need to implement functionality for your learned model to be used in predicting concepts from new instances, which I call *responding* later in the book. This is ultimately where the rubber meets the road in a machine learning system, and in the Pooch Predictor system it was frequently where the car burst into flames. Given that team Sniffable had never really built a machine learning system like this before, it's not surprising that there were some pain points where their ideas met harsh reality. Some of their problems stemmed from treating their predictive system like a transaction business application that needed to record a purchase. An approach that relies on strong consistency guarantees doesn't work for modern distributed systems, and it's out of sync with the pervasive and intrinsic uncertainty in a machine learning system. Other problems the Sniffable team experienced had to do with not thinking about their system in dynamic terms. machine learning systems must evolve, and they must support parallel tracks for that evolution through experimentation capabilities. Finally, there wasn't much functionality to support handling requests for predictions.

The Sniffable team wasn't unusual in their haphazard approach to architecture. Many machine learning systems look a lot like the architecture in figure 1.5.

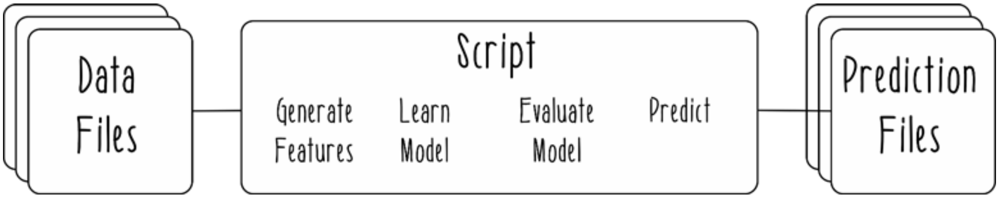


Figure 1.5 A simplistic machine learning system

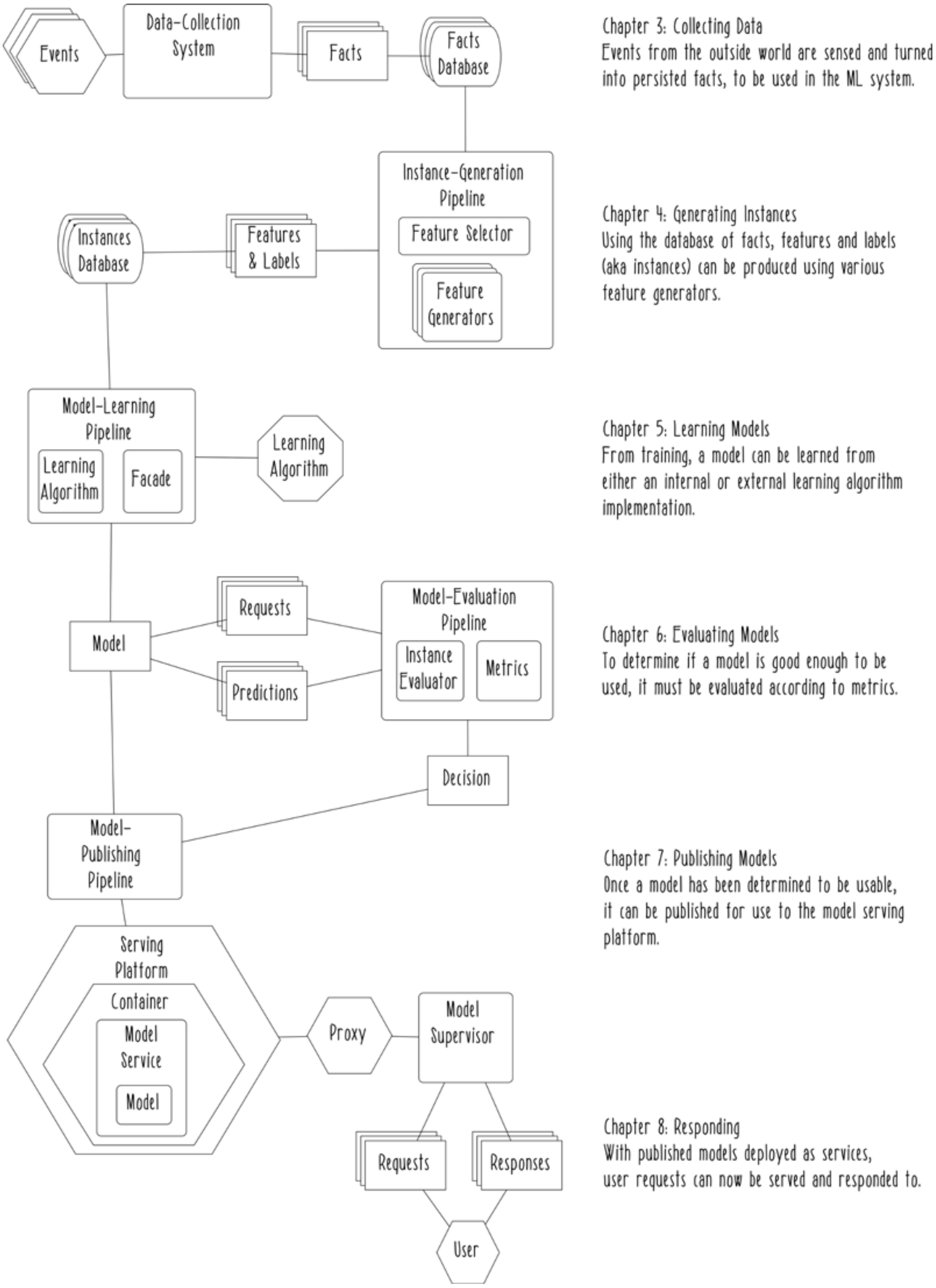
There's nothing wrong with starting with something so simple. But this approach lacks many system components that will eventually be needed, and the ones that are implemented have poor component boundaries. Moreover, not a lot of thought was given to the various properties this system must have, should it ever serve more than a few users. It is, in a word, naive.

This book introduces an approach to building machine learning systems that is anything but naive. The approach is based on a lot of real-world experiences with the challenges of machine learning systems. The sorts of systems that we'll look at in this book are non-trivial and often have complex architectures. At a general level, they will conform to the approach shown in figure 1.6.

It may not be obvious why we need to build machine learning systems using such a complex architecture, but I beg your patience. In each chapter, I'll show you what challenges this portion of the system must address and how a more reactive approach to machine learning will work better. To do that, I should probably give you more background on what reactive systems are.

1.2.2 Reactive systems

Now that you understand a bit more about what machine learning systems are, I want to give you an overview of some of the ideas and approaches that we'll use to build successful ones. We'll begin with the *reactive systems paradigm*. Reactive systems are defined by four traits and three strategies. The paradigm as a whole is a way of codifying an approach to building systems that can serve modern user expectations for things like interactivity and availability.



Chapter 3: Collecting Data
Events from the outside world are sensed and turned into persisted facts, to be used in the ML system.

Chapter 4: Generating Instances
Using the database of facts, features and labels (aka instances) can be produced using various feature generators.

Chapter 5: Learning Models
From training, a model can be learned from either an internal or external learning algorithm implementation.

Chapter 6: Evaluating Models
To determine if a model is good enough to be used, it must be evaluated according to metrics.

Chapter 7: Publishing Models
Once a model has been determined to be usable, it can be published for use to the model serving platform.

Chapter 8: Responding
With published models deployed as services, user requests can now be served and responded to.

Figure 1.6 A reactive machine learning system

Traits of reactive systems

Reactive systems privilege four traits (see figure 1.7).

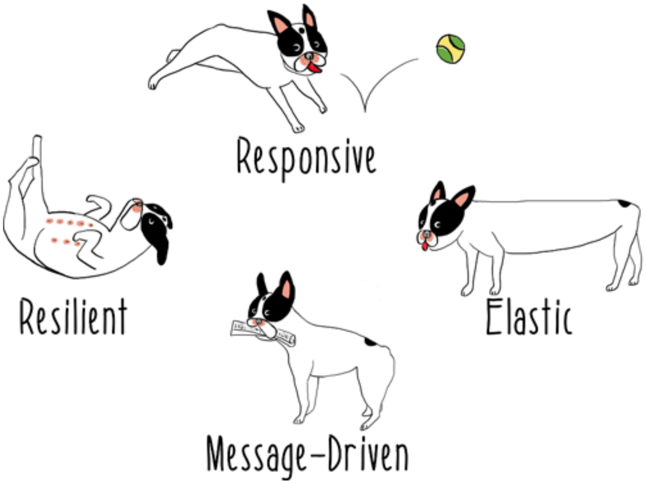


Figure 1.7 The traits of reactive systems

First and most importantly, reactive systems are *responsive*, meaning they consistently return timely responses to users. Responsiveness is the crucial foundation upon which all future development efforts will be built. If a system doesn't respond to its users, then it's useless. Think of the Sniffable team causing a massive slowdown in the Sniffable app due to the poor responsiveness of their machine learning system.

Supporting that goal of responsiveness, reactive systems must be *resilient*; they need to maintain responsiveness in the face of failure. Whether the cause is failed hardware, human error, or design flaws, software always breaks, as the Sniffable team has discovered. Providing some sort of acceptable response even when things don't go as planned is a key part of ensuring that users view a system as being responsive. It doesn't matter that an app is very fast when it's not broken if it's broken half the time.

Reactive systems must also be *elastic*; they need to remain responsive despite varying loads. The idea of elasticity isn't exactly the same as scalability, although the two are similar. Elastic systems should respond to increases *or* decreases in load. The Sniffable team saw this when their traffic ramped up and the Pooch Predictor system couldn't keep up with the load. That's exactly what a lack of elasticity looks like.

Finally, reactive systems are *message-driven*; they communicate via asynchronous, non-blocking *message passing*. The message-passing approach is in contrast with direct intraprocess communication or other forms of tight coupling. It's easy to understand how a more explicit approach to ensuring loose coupling might solve some of the issues in the Sniffable example. A loosely coupled system organized around message passing can make it easier to detect failure or issues with load. Moreover, a design with this trait helps contain any of the effects of errors to just messages about bad news, rather than flaming production issues that need to be immediately addressed, as they were in Pooch Predictor.

The reactive approach could certainly be applied to the problems the Sniffable team were having with their machine learning system. The four principles represent a coherent and complete approach to system design that makes for fundamentally better systems. Such systems fulfill their requirements better than naively designed systems, and they're more fun to work on. After all, who wants to fight fires when you could be shipping awesome new machine learning functionality to loyal sniffers?

These traits certainly sound nice, but they're not much of a plan. How do you build a system that actually has these traits? Message passing is part of the answer, but it's not the whole story. machine learning systems, as you've seen, can be difficult to get right. They have unique challenges that will likely need unique solutions that don't appear in traditional business applications.

REACTIVE STRATEGIES

A key part of how we'll build a reactive machine learning system in this book is by using the three reactive strategies illustrated in figure 1.8.

Figure 1.8 Reactive strategies

First, reactive systems use *replication*. They have the same component executing in more than one place at the same time. More generally, this means that data, whether at rest or in motion, should be redundantly stored or processed.

In the Sniffable example, there was a time when the server that ran the model-learning job failed, and no model was learned. Clearly, replication could have helped here. Had there been two or more model-learning jobs, the failure of one job would have had less impact. Replication may sound wasteful, but it's the beginning of a solution. As you'll see in chapters 4 and 5, you can build replication into your modeling pipelines using Spark. Rather than requiring you to always have two pipelines executing, Spark gives you automatic, fine-grained replication so that the system can recover from failure. This book focuses on the use of higher-level tools like Spark to manage the challenges of distributed systems. By relying on these tools, you can easily use replication in every component of your machine learning system.

Next, reactive systems use *containment* to prevent the failure of any single component of the system from affecting any other component. The term *containment* might get you thinking about specific technologies like Docker and rkt, but this strategy isn't about any one implementation. Containment can be implemented using many different systems, including homegrown ones. The point is to prevent the sort of cascading failure we saw in Pooch Predictor, and to do so at a structural level.

Consider the issue with Pooch Predictor where the model and the features were out of sync, resulting in exceptions during model serving. This was only a problem because the model-serving functionality wasn't sufficiently contained. Had the model been deployed as a contained service communicating with the Sniffable application server via message passing, there would have been no way for this failure to propagate as it did. Figure 1.9 shows an example of this architecture.

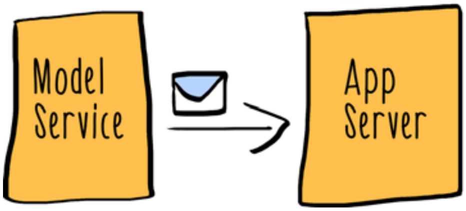


Figure 1.9 A contained model-serving architecture

Lastly, reactive systems rely on the strategy of *supervision* to organize components. When implementing systems using this strategy, you explicitly identify the components that could fail and make sure that some other component is responsible for their lifecycles. The strategy of supervision gives you a point of control, where you can ensure that the reactive traits are being achieved by the true runtime behavior of your system.

The Pooch Predictor system had no system-level supervision. This unfortunate omission left the Sniffable team scrambling whenever something went wrong with the system. A better approach would have been to build supervision directly into the system itself, along the lines of figure 1.10.

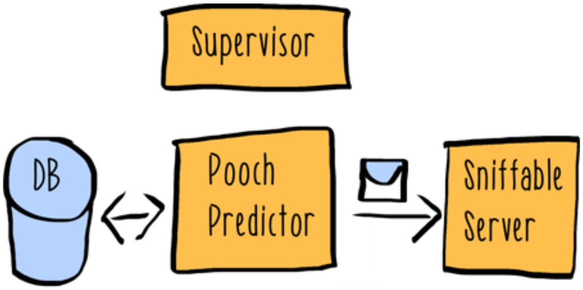


Figure 1.10 A supervisory architecture

In this structure, the published models are observed by the model supervisor. Should their behavior deviate from acceptable bounds, the supervisor would stop sending them messages requesting predictions. In fact, the model supervisor could even completely destroy a model it knows to be bad, making the system potentially self-healing. I'll begin discussing how you can implement model supervision in chapters 6 and 7, and we'll continue exploring powerful applications of the strategy of supervision throughout the remainder of the book.

1.2.3 Making machine learning systems reactive

With some understanding about reactive systems, I can begin discussing how we can apply these ideas to machine learning systems. In a reactive machine learning system, we still want our system to have all the same traits as a reactive system, and we can use all the same strategies. But we can do more to address the unique characteristics of a machine learning system. So far, I've explained a lot of infrastructural concerns, but I haven't yet shown you how this enables new *predictive* capabilities. Ultimately, a reactive machine learning system gives you the ability to deliver value through ever better predictions. That's why reactive machine learning is worth understanding and applying.

The reactive machine learning approach is based on two key insights into the characteristics of data in a machine learning system: it is uncertain, and it is effectively infinite. From those two insights, four strategies emerge, shown in figure 1.11, that will help us build a reactive machine learning system.

To begin, let's think about how much data the Pooch Predictor system might need to process. Ideally, with its new machine learning capabilities, Sniffable will take off and see tons of traffic. But even if that doesn't happen, there's still no way of knowing how many possible pupdates users might want to consider and thus send to the Pooch Predictor system. Imagine having to predict every possible post that a sniffer might make on Sniffable. Some posts would have big dogs; others, small ones. Some posts would use filters, and others would be more natural. Some would be rich in hashtags, and some wouldn't have any annotations. Once you consider the impact of arbitrary parameters on feature values, the range of possible data representations becomes literally *infinite*.

It doesn't matter precisely how much raw data Pooch Predictor ingests. We'll always assume that the amount of data is too much for one thread or one server. But rather than give up in the face of this unbounded scope, reactive machine learning employs two strategies to manage infinite data.

First, it relies on *laziness*, also known as *delay of execution*, to separate the composition of functions to execute from their actual execution. Rather than being a bad habit, laziness is a powerful evaluation strategy that can greatly improve the design of data-intensive applications.

By using laziness in the implementation of your machine learning system, you'll find that it's much easier to conceive of the data flow in terms of *infinite streams* than *finite batches*. This switch can have huge benefits for the responsiveness and utility of your system. I show how laziness can be used to build machine learning pipelines in chapter 4.

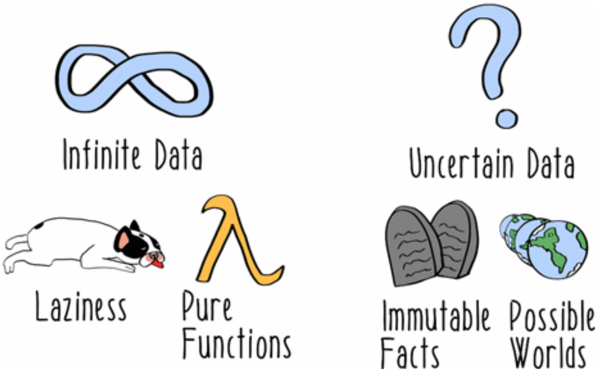


Figure 1.11 Reactive machine learning data and strategies

λ Pure Functions

Similarly, reactive machine learning systems deal with infinite data by expressing transformations as *pure functions*. What does it mean for a function to be pure? First, evaluating the function must not result in some sort of side effect, such as changing the state of a variable or performing I/O. Additionally, the function must always return the same value when given the same arguments. This latter property is referred to as *referential transparency*. Writing machine learning code that maintains this property can make implementations of mathematical transformations look and behave quite similarly to their expression in math.

Pure functions are a foundational concept in a style of programming known as *functional programming*, which we'll use throughout this book. At its heart, functional programming is all about computing with functions. In functional code, functions can be passed to other functions as arguments. Such functions are called *higher-order functions*, and we'll use this idiom throughout the code examples in this book. Functional programming idioms like higher-order functions are a key part of what makes reactive tools like Scala and Spark so powerful.

The emphasis on the use of functional programming in this book isn't merely stylistic. Functional programming is one of the most powerful tools for taming complicated systems that need to reason about data, especially infinite data. The recent increase in the popularity of functional programming has been largely driven by its application to building big data infrastructure. Using the techniques of functional programming, we'll be able to get our system right *and* scale it to the next level. As I discuss in chapters 4 and 6, pure functions can offer real solutions to the problems of implementing feature extraction and prediction functionality.

? Uncertain Data

Next, let's consider what Pooch Predictor knew about what was going on with Sniffable and its users. It had records of sniffers creating, viewing, and liking pupdates. This knowledge came from the main application database. As we saw, the app would sometimes lose sniffers' efforts to like a particular pupdate, due to operational issues, and this loss of data changed the concept that Pooch Predictor was built to learn. Similarly, Pooch Predictor's view of what feature values were seen at a given time was often impeded by bugs in its code or in the main app's code. This is all because *uncertainty* is intrinsic and pervasive in a machine learning system.

Machine learning models and the predictions they make are always approximate and only useful in the aggregate. It wasn't like Pooch Predictor knew *exactly* how many likes a given pupdate might get. Even before making a prediction, a machine learning system must deal with the uncertainty of the real world outside of the machine learning system. For example, do sniffers using the hashtag #adorabull mean the same thing as sniffers using the hashtag #adorable, or should those be viewed as different features?

A truly reactive machine learning system incorporates this uncertainty into the design of the system and uses two strategies to manage it: *immutable facts* and *possible worlds*. It may sound strange to use facts to manage uncertainty, but that's exactly what we're going to do. Consider the location that a sniffer is posting a pupdate from. One way of recording this location data for later use in geographic features is to record the exact location reported by the app, as in table 1.1.

Table 1.1 Pupdate location data model

pupdate_id	Location
123	Washington Square Park

But the location determined by the app at the time of the pupdate was uncertain; it was just the result of a sensor reading on a phone, which has a very coarse level of precision. The sniffer may or may not have been in Washington Square Park. Moreover, if a future feature tries to capture the distinct differences between East and West Greenwich Village, this data model will give a precise but potentially inaccurate view of how far to the east or west this pupdate came from.

A richer, more accurate way of recording this data is to use the raw location reading and the expected radius of uncertainty, as shown in table 1.2.

Table 1.2 Revised pupdate location data model

pupdate_id	Latitude	Longitude	Radius
123	40.730811	-73.997472	1.0

Immutable Facts

This revised data model can now represent *immutable facts*. This data can be written once and never modified; it is written in stone. The use of immutable facts allows us to reason about uncertain views of the world at specific points in time. This is crucial for creating accurate instances and many other important data transformations in a machine learning system. Having a complete record of all facts that occur over the lifetime of the system also enables important machine learning, like model experimentation and automatic model validation.

To understand the other strategy for dealing with uncertainty, let's consider a fairly simple question: how many likes will pupdates about French Bulldogs get in the next hour? To answer this question, let's break it down into pieces.

First, how many pupdates will be submitted in the next hour? There are multiple ways of answering this question. We could just take the historical average rate—say, 6,500. But the number of pupdates submitted varies over time, so we could also fit a line to the data that looks something like figure 1.12. Using this model, we might expect 7,250 pupdates in the next hour.

Beyond that, we need to know how many likes these pupdates will receive. Again, we could take a historical average, which would give us 23 likes per pupdate in this case. Or we could use a model. That model would have to be applied to some recent sample of data to get an idea of the likes that recent traffic has been getting. The result of this model is that the average pupdate will receive 28 likes.

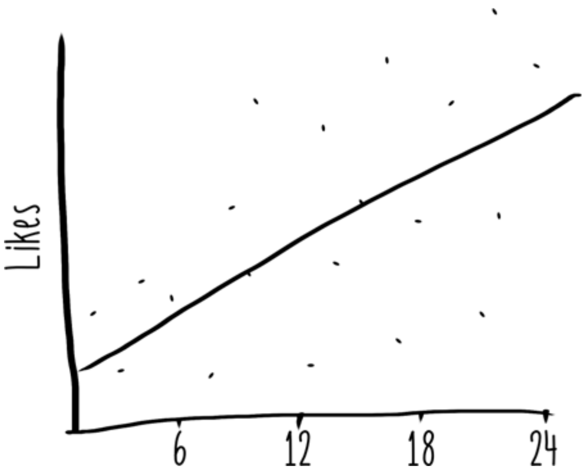


Figure 1.12 Model of likes by hour

Now, we need to combine this information in some way. Table 1.3 shows the predictions we could use in our final prediction.

Table 1.3 Possible prediction values

Model type	Pupdates	Likes/Pupdate
Historical	6,500	23
Machine-learned	7,250	28

We could decide to answer that the expected number of likes in the next hour is 6,500 x 23 = 149,500 using the historical values. Or we could decide to use the machine-learned model and get a value of 7,250 x 28 = 203,300. We could even decide to combine the historical number of pupdates with the model-based prediction of likes per pupdate to get 6,500 x 28 = 182,000. These different views of our uncertain data can be thought of as *possible worlds.*

We don't know which of these worlds we will ultimately find ourselves in during the next hour of traffic on Sniffable, but we can make decisions with this information, such as ensuring that the servers are prepared to handle more than 200,000 likes in the next hour. Possible worlds will form the basis for the queries we'll make of all the uncertain data that is present in our machine learning system. There are limits to the applicability of this strategy, because infinite data can produce infinite possible worlds. But by building our data models and queries with the concept of possible alternative worlds, we'll be able to more effectively reason about the real range of potential outcomes in our system.

Using all the strategies that I've discussed, it's easy to imagine the Sniffable team refactoring the Pooch Predictor system into something much more powerful. The reactive machine learning approach makes it possible to build a machine learning system that has fewer problems and allows for evolution and improvement. It's definitely a different approach than we saw in the original Pooch Predictor example, and this approach is grounded on a firmer footing. Reactive machine learning unites ideas from distributed systems, functional programming, uncertain data, and other fields in a coherent, pragmatic approach to building real-world machine learning systems.

1.2.4 When not to use reactive machine learning

It's fair to ask whether all machine learning systems should be built using the reactive approach. The answer is no.

During the design and implementation of a machine learning system, it's beneficial to consider the principles of reactive machine learning. Machine learning problems by definition have to do with reasoning about uncertainty. Thinking in terms of immutable facts and pure functions is a useful perspective for implementing any sort of application.

But the approach discussed in this book is a way to easily build sophisticated systems, and some machine learning systems don't need to be sophisticated. Some systems won't benefit from using a message-passing semantic that assumes several independently executing processes. A research prototype is a perfect example of a machine learning system that doesn't need the powerful capabilities of a reactive machine learning system. When you're building a temporary system, I recommend bending or breaking all the rules I lay out in this book. The prudent approach to building potentially disposable machine learning systems is to make far more extreme compromises than in the reactive approach. If you're building such a temporary system, see my guide to building machine learning systems at hackathons: http://mng.bz/981c.

Summary

- Even simple machine learning systems can fail.
- Machine learning should be viewed as an application, not as a technique.
- A machine learning system is composed of five components, or phases:
  - The data-collection component ingests data from the outside world into the machine learning system.
  - The data-transformation component transforms raw data into useful derived representations of that data: features and concepts.
  - The model-learning component learns models from the features and concepts.
  - The model-publishing component makes a model available to make predictions.
  - The model-serving component connects models to requests for predictions.
- The reactive systems design paradigm is a coherent approach to building better systems:
  - Reactive systems are responsive, resilient, elastic, and message-driven.
  - Reactive systems use the strategies of replication, containment, and supervision as concrete approaches for maintaining the reactive traits.

- Reactive machine learning is an extension of the reactive systems approach that addresses the specific challenges of building machine learning systems:
  - Data in a machine learning system is effectively infinite. Laziness, or delay of execution, is a way of conceiving of infinite flows of data, rather than finite batches. Pure functions without side effects help manage infinite data by ensuring that functions behave predictably, regardless of context.
  - Uncertainty is intrinsic and pervasive in the data of a machine learning system. Writing all data in the form of immutable facts makes it easier to reason about views of uncertain data at points in time. Different views of uncertain data can be thought of as possible worlds that can be queried across.

In the next chapter, I'll introduce some of the technologies and techniques used to build reactive machine learning systems. You'll see how reactive programming techniques allow you to deal with complex system dynamics without complex code. I'll also introduce two powerful frameworks, Akka and Spark, that you can use to build incredibly sophisticated reactive systems easily and quickly.

2 Using reactive tools

This chapter covers

- Managing uncertainty with Scala
- Implementing supervision and fault tolerance with Akka
- Using Spark and MLlib as frameworks for distributed machine learning pipelines

To get ready to build full-scale reactive machine learning systems, you need to get familiar with a few tools from the Scala ecosystem: Scala itself, Akka, and Spark. In this book, we'll write applications in Scala because it provides excellent support for functional programming and has been used successfully in building reactive systems of all kinds. Sometimes, you'll find that Akka can be useful as a tool for providing resilience and elasticity through its implementation of the actor model. Other times, you'll want to use Spark to build large-scale pipeline jobs like feature extraction and model learning. In this chapter, you'll just start to get familiar with these tools, and beginning with chapter 3, I'll show you how they can be used to build the various components of a reactive machine learning system.

These aren't the only tools that you could use to build a reactive machine learning system. Reactive machine learning is a set of ideas, not a specific implementation. But the technologies shown in this chapter are all very useful for reactive machine learning, in large part because they were designed with strong support for reactive techniques. Even though I'm going to introduce you to the specifics of how these tools work, you can definitely apply these approaches to systems built in other languages using other tools.

I'll introduce you to this book's toolchain in the context of one of the world's most crucial problems: finding the next breakout pop star. *Howlywood Star* is a canine reality singing competition. Each week, unknown dogs from around the country sing in front of a panel of three judges. Then, the viewers at home vote on which dog has what it takes to be the next Howlywood Star. This voting mechanic is key to the runaway success of the show. The audience tunes in each week as much for the competition as for the singing.

A suite of sophisticated apps support this audience participation dynamic, and they're what you'll focus on in this chapter. You'll work primarily on the challenges of handling the voting functionality. There will be some tricky scenarios resulting from the popularity and unpredictability of the competition. Once you've addressed today's votes, we'll try to predict things about future voting patterns using machine learning.

2.1 Scala, a reactive language

In this book, all the examples are in Scala. If you haven't used Scala before, don't worry. If you're competent in Java or a similar mainstream language, you can quickly learn enough Scala to begin to build powerful machine learning systems. It's true that Scala is a large and rich language that could take you quite a while to master. But you'll mostly be using the power of Scala, without having to write terribly sophisticated code yourself. Rather than try to introduce you to all the amazing features in Scala, this section focuses on the features of the language that support reactive programming and reasoning about uncertainty.

To get started with Scala programming, you'll build some of the pieces of the voting application for *Howlywood Star*. The application's architecture is shown in figure 2.1.

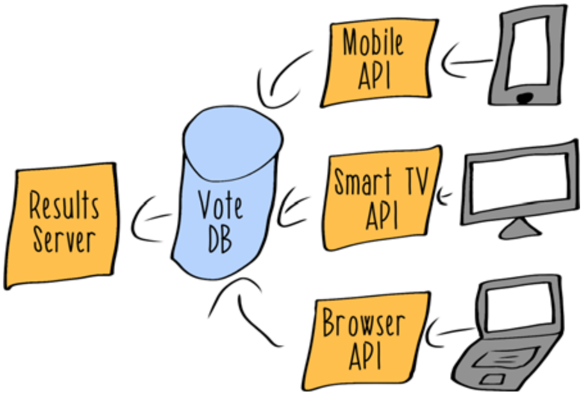


Figure 2.1 Howlywood Star voting application architecture

Various user-facing mobile and web apps are responsible for sending votes from the global *Howlywood Star* audience to backend servers. The servers are responsible for receiving these votes and forwarding them on to the application database. Other visualization apps are then responsible for querying that database and presenting the current results. These apps range from sophisticated internal-analytics dashboards to simple public-facing mobile apps like the one shown in figure 2.2.

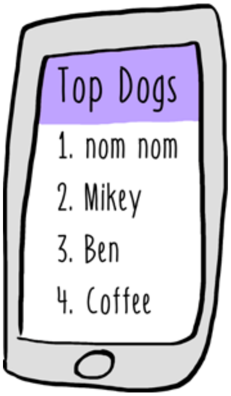


Figure 2.2 Voting results mobile app

This system is very simple, but even a system as simple as this has hidden complexity. Consider the following questions:

- How long will it take to record each vote?
- What will the server be doing while it waits for each vote to be persisted?
- How can the visualization apps be kept as fresh as possible?
- What will happen if load increases dramatically?

Howlywood Star's popularity has recently been exploding thanks to a lot of interest on social media. This voting app needs to be ready for the frenzy of attention that the upcoming Season 2 is expected to produce. When the audience latches on to the next breakout star, it's reasonable to expect that the voting app will be slammed with traffic.

But you can't know in advance how big that traffic spike will be. There's a certain amount of intrinsic uncertainty in trying to predict the future like that.

Nevertheless, the voting app will have to be ready for that uncertain future. Thankfully, Scala has tools for handling uncertainty and reacting appropriately.

2.1.1 Reacting to uncertainty in Scala

Before we get into discussions of more-complex distributed systems, let's discuss some basic techniques you can use to manage uncertainty in Scala. Let's begin with some fairly naive code that will allow you to begin to explore the richness of Scala. Your initial implementation won't represent production-grade Scala code, but rather will be a basic exploration of how different object types work in Scala.

In the following listing, you create a simple collection of Howlers and the number of votes they currently have. Then, you try to retrieve the vote counts for a popular Howler.

Listing 2.1 A map of votes

```
val totalVotes = Map("Mikey" -> 52, "nom nom" -> 105)

val naiveNomNomVotes: Option[Int] = totalVotes.get("nom nom")
```

The collection of votes received thus far

An option that must be "unwrapped" to get the vote count

This trivial example demonstrates Scala's concept of an Option type. In this example, the language will allow you to pass any string key to the map of votes, but it doesn't know whether anyone has voted for nom nom until executing the lookup. Option types can be viewed as a way of encoding the intrinsic uncertainty in an operation. They close over the possibility that a given operation may return a value, Some of a given type, or None.

Because Scala has already told you that there's some uncertainty around the contents of the vote map, you can now write code that handles the different possibilities.

Listing 2.2 Handling no votes using pattern matching

```
def getVotes(howler: String) = {
  totalVotes.get(howler) match {
    case Some(votes) => votes
    case None => 0
  }
}

val nomNomVotes: Int = getVotes("nom nom")
val indianaVotes: Int = getVotes("Indiana")
```

A function that handles the possibility that no one may have voted for a given dog yet

A pattern-match expression to handle the two possibilities

Returns 105

Returns 0

This simple helper function uses *pattern matching* to express the two possibilities: either you've received votes for a given Howler or you haven't. In the latter case, that means the correct number of votes is 0. This allows the type of the votes values to both be Int, even though no one has yet voted for Indiana. Pattern matching is a language feature used to encode what the possible values are that a given operation could produce. In this case, you're expressing the possible cases that the value returned by the get operation could match to. Pattern matching is a common and useful technique in idiomatic Scala, which we'll use throughout the book.

Of course, this very simple form of uncertainty is so common that Scala gives you facilities to address it within the collection. The helper function in listing 2.2 can be eliminated by setting a default value on the votes map.

Listing 2.3 Setting default values on maps

```
val totalVotesWithDefault = Map("Mikey" -> 52, "nom nom" -> 105)
  ➥ .withDefaultValue(0)
```

2.1.2 The uncertainty of time

Building on this line of thinking, let's consider a more relevant form of uncertainty. If the count of votes were stored on a different server than the one you're on, then it would take time to retrieve those votes. The following listing approximates that idea using a random delay.

Listing 2.4 A remote “database”

```
def getRemoteVotes(howler: String) = {
  Thread.sleep(Random.nextInt(1000))
  totalVotesWithDefault(howler)
}

val mikeyVotes = getRemoteVotes("Mikey")
```

A function that retrieves the votes, but with varying amounts of delay

Always returns 52, eventually

This sort of uncertainty is a big problem for the vote visualization app. Its server will be doing nothing, just waiting, while that call is processed. You can imagine that won’t help in the quest to achieve responsiveness at all times.

The source of this performance problem is that the call to `getRemoteVotes` is *synchronous*. The solution to this problem is to use a *future*, which will ensure that this call is no longer made in a synchronous, blocking fashion. Using a future, you’ll be able to return immediately from a remote call like this and collect the result later, once the call has completed. The following listing shows how this can be done to answer the question “Which Howler is currently the most popular?”

Listing 2.5 Futures-based remote calls

```
import scala.concurrent._
import ExecutionContext.Implicits.global

def futureRemoteVotes(howler: String)= Future {
  getRemoteVotes(howler)
}

val nomNomFutureVotes = futureRemoteVotes("nom nom")
val mikeyFutureVotes = futureRemoteVotes("Mikey")
val indianaFutureVotes = futureRemoteVotes("Indiana")

val topDogVotes: Future[Int] = for {
  nomNom <- nomNomFutureVotes
  mikey <- mikeyFutureVotes
  indiana <- indianaFutureVotes
} yield List(nomNom, mikey, indiana).max

topDogVotes onSuccess {
  case _ => println("The top dog currently has" + topDogVotes + "votes.")
}
```

A function that returns a future of the count of votes

The creation of each of these futures returns immediately rather than blocking on the remote call.

This syntax is called a for expression.

A future that will eventually contain the maximum number of votes

This will execute once all values in the for expression have been retrieved.

Prints “The top dog currently has 105 votes.”

In this implementation, the three calls to the remote vote count collection are processed concurrently. Creation of a future doesn’t block on the remote call, waiting for completion of the work. Instead, the creation of the future returns immediately, allowing for the later concurrent processing. Using futures to abstract over time is a foundational technique that you'll use repeatedly to scale up your reactive machine learning systems for handling huge amounts of data and complex operational behavior.

The response time of a given request to a remote data source might, on average, be quite small. But with large amounts of data, it's effectively guaranteed that some response times won't be close to the average. This is an outcome of basic statistics. In a normally distributed dataset, there will be outliers. And in aggregation operations, like the maximum votes calculation in listing 2.5, the average request latency has no effect on the total latency. Instead, the total latency is entirely determined by the single slowest request, as shown in figure 2.3.

The amount of time it takes for that slowest request is often called the *tail latency*. It's a very real problem in large-scale data-processing systems of all kinds, including machine learning systems. But now that you know tail latency is a problem, you can use reactive programming techniques to manage it.

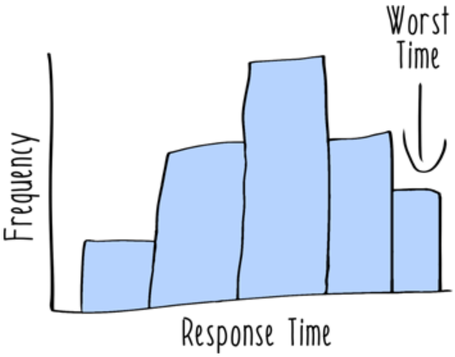


Figure 2.3 Distribution of request times

Listing 2.6 Futures-based timeouts

```
val timeoutDuration = 500
val AverageVotes = 42

val defaultVotes = Future {
  Thread.sleep(timeoutDuration)
  AverageVotes
}

def timeOutVotes(howler: String) = Future.firstCompletedOf(
  List(getRemoteVotes(howler), defaultVotes))
```

Amount of time allowed before returning a value

Historical average number of votes

A future that will complete with the average number of votes after the timeout duration

A function to return the actual number of votes or the default, should the remote call timeout

In this implementation, you accept that life isn't perfect, and some remote calls may exhibit unacceptable latency. Rather than pass that latency on to the user, you choose to return a degraded response, the historical average number of votes. That number isn't literally accurate, but in this case it's better than returning nothing at all. In a real system, you may have several options for what to return as a degraded response. For example, you may have another application to look this value up in, such as a cache. That cache's value may have gotten stale, but that degraded value might be more useful than nothing at all. In other cases, you may want to encode retry logic. It's up to you to figure out what's best for your application.

You may not like planning to fail some of the time, and if so, I can understand your misgivings. As engineers, we're used to building systems that return perfectly correct answers every time. But in machine learning systems, uncertainty is pervasive and intrinsic. It turns out that the same is true of distributed systems as well. If we want our systems to be reactive and responsive, we're going to have to use tactics like these *some* of the time.

As you'll see in chapter 5 and beyond, there are several machine learning–specific scenarios where we need to fall back in some way on a less-than-perfect response. Prudent risk-mitigation tactics like these will help us tame some of the complexity of our machine learning system.

2.2 Akka, a reactive toolkit

The next tool I'm going to introduce is Akka. It's an important tool to understand because it gives you reusable components to construct elastic and resilient systems. As you saw in chapter 1, it can be easy to build a machine learning system that doesn't hold up to the real-world challenges of scale and failure. Akka and the ideas behind it provide solutions to some of those problems.

But first, I'll spend some time familiarizing you with the basics of Akka usage. Compared to the Akka material later in the book, this section is introductory and mostly intended to give you some context and understanding of *how* Akka does what it does. Once you're comfortable with how systems built on Akka achieve their guarantees, then we can move on to using Akka primarily as a dependent library that powers higher-level abstractions. Don't worry if some of Akka remains a mystery to you. We'll only be using Akka for some parts of this book. It's a powerful and complex toolkit; this section only scratches the surface of what it can do. The main goal here is to develop a mental model of how an Akka system honors its guarantees. But before you can understand how Akka works, you need to understand how actors work.

2.2.1 The actor model

The actor model is a way of thinking of the world that identifies each thing as an actor. What's an actor? An actor is a pretty simple thing. In response to a message it receives, it can only do three things:

- Send messages
- Create new actors
- Decide how to behave when it receives its next message

That may sound limiting, but it's quite useful.

First, let's consider communicating via sending messages. You saw this communication style in action in the Sniffable example from chapter 1. I proposed that the model service would have been better able to contain its failure had it communicated with the main app via message passing, illustrated in figure 2.4.

Message passing in itself gives a system some of the benefits of a full actor system. That's because message passing is an effective approach to implementing containment.

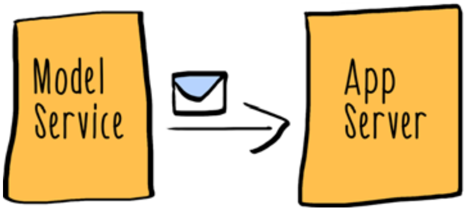


Figure 2.4 A contained model-serving architecture

By implementing strong boundaries that can only be crossed via messages, actors (or services that behave like actors) can't contaminate other components of the system when they fail. In a large system refactoring, often a good place to start is by separating out components so they only communicate via message passing. That would have been a good next step for the developers of the Pooch Predictor system from chapter 1. Well-contained components of a machine learning system are easier to operate and improve on the journey to reactivity.

Next, actors may also have the power to create new actors. That's important for things like creating a supervisory hierarchy. You'll see several examples of supervisory hierarchies in this book, with Akka actors and other related concepts.

You also saw this on a larger scale in the Sniffable example from chapter 1. In that section, I proposed that the system as a whole would benefit from a supervisory hierarchy, shown again in figure 2.5.

In this architecture, the supervisor has the complete power of life and death over supervised services. That's exactly how things work in the actor model. The benefits of supervision are similar to the benefits of containment in this context. By building the concept of failure into the architecture, you now how have system-level solutions for inevitable failures.

Lastly, actors can do something you haven't seen before: they can change their behavior. After receiving a given message, an actor will decide to do something different the next time it receives a message. This means that actors are stateful, a bit like objects in imperative object-oriented programming (as in Java, Python, and so on). Most of the code you've seen thus far has hewed to a functional programming style that tries to avoid explicit state manipulation. But components of the system like remote services have a current state, and you'll find the actor model a useful approach to reasoning about that state. The actor model has a coherent concept of

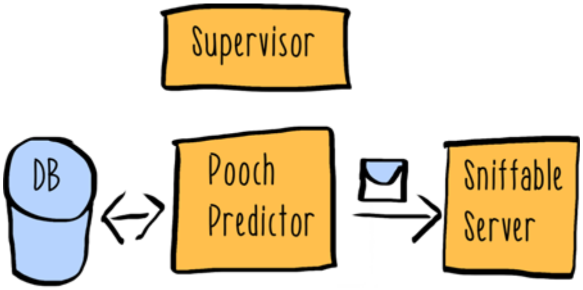


Figure 2.5 A supervisory architecture

how to encapsulate that state while providing a method of interacting with the outside world. That doesn't mean you'll throw out all the advantages of immutable facts and pure functions. Only code that needs to deal with state will be stateful. But a fully reactive machine learning system has many components with many different needs. Thankfully, Scala is a robust and pragmatic language that privileges the benefits of immutability and purity but allows you to handle state when necessary.

2.2.2 Ensuring resilience with Akka

That's enough theory; let's put the actor model to work with a real problem: ensuring reliable records of votes on *Howlywood Star*. You're still within the same overall *Howlywood Star* voting system originally shown in figure 2.1. But now you're concerned specifically with writing received votes from the API servers in the remote vote database. This is obviously a pretty common software development scenario, so I'm going to complicate it slightly. In this scenario, a Chihuahua was put in charge of maintaining the database, and it did a really bad job. Sometimes the database will record the vote, and other times the database will just poop itself by throwing an exception. This fundamental unreliability means you'll need to implement the actor hierarchy shown in figure 2.6.

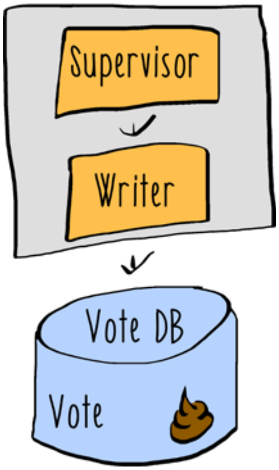


Figure 2.6 Vote-writing actor hierarchy

All remote resources like databases have a certain level of unreliability, but this one is worse than most. The database fails to record a write every other time a vote is sent across.

Listing 2.7 An unreliable database

```
class DatabaseConnection(url: String) {
  var votes = new mutable.HashMap[String, Any]()

  def insert(updateMap: Map[String, Any]) = {
    if (Random.nextBoolean()) throw new Exception

    updateMap.foreach {
      case (key, value) => votes.update(key, value)
    }
  }
}
```

A simple object used to represent a database

A hash map to hold the votes

This will throw an exception half the time.

A pattern-matching expression is used to destructure the map entry into a key and a value.

If I've said it once, I've said it a hundred times: you just can't trust purse dogs with crucial systems administration tasks—they don't have the focus. Database administration is clearly a job for a Mastiff.

Mutable objects

Listing 2.7 uses a `var`, a mutable object. Mutable objects are used very infrequently in idiomatic Scala code. In this example, the use of mutability is just to simplify this example. It's not necessary to use mutability here, and we'll generally avoid the use of `var` in this book in favor of immutable objects. But Scala gives us the choice between using mutable and immutable data, which can be helpful for exploring the trade-offs of the relevant design choices.

This situation is a mess. You want to record instances of votes in this database.

Listing 2.8 A vote case class

```
case class Vote(timestamp: Long, voterId: Long, howler: String)
```

But that Chihuahua has made all this much harder. Some of the votes will get lost by the database. Worse, if you don't protect your vote-writing code, it will fail on the first exception it receives.

I do have some good news, though. Voting on *Howlywood Star* is a large-scale data-processing application. Any individual vote isn't that important. You've been told by management that losing some of these votes is totally acceptable and even expected. What's crucial is that the voting application remain responsive in the face of underlying failures.

When you move on to building large-scale machine learning systems, you'll see similar situations. The value of any one feature or instance won't be that great, because you'll be processing large aggregates to learn models and make predictions. Sometimes data can be discarded. Remember, you assume that data is effectively *infinite* in a reactive machine learning system.

In fact, this technique of judicious prioritization and sacrificing data will come together into an incredibly useful reactive programming technique called the *circuit breaker* pattern, which I discuss later in the book.

Let's see how you can fulfill the limited scope of your mission even with your unreliable vote database. Using Akka, create a vote writer as an actor.

Listing 2.9 A vote-writing actor

```
class VoteWriter(connection: DatabaseConnection) extends Actor {
  def receive = {
    case Vote(timestamp, voterId, howler) =>
      connection.insert(Map("timestamp" -> timestamp,
        "voterId" -> voterId,
        "howler" -> howler))
  }
}
```

A simple actor that receives votes and writes them to the database

The method that will receive messages to the actor

Now create another actor to supervise this actor, as shown in the following listing. It will be responsible for handling failures by the `VoteWriter` due to the unreliable database. This supervisory actor should merely recover from errors and not worry about any data that might have been lost.

Listing 2.10 A supervisory actor

```
class WriterSupervisor(writerProps: Props) extends Actor {
  override def supervisorStrategy = OneForOneStrategy() {
    case exception: Exception => Restart
  }

  val writer = context.actorOf(writerProps)

  def receive = {
    case message => writer forward message
  }
}
```

A supervisory actor instantiated with a Props, a configuration object for an actor

Supervision strategy the supervisory actor will use for the VoteWriter

The strategy is to restart in the event of failure; this will clear any internal state in the actor.

Creates the writer actor in the given context

Passes all messages through to the supervised VoteWriter actor

Note that you use `Restart`, which is an Akka `Directive`, a convenient building block provided by the Akka toolkit. Listing 2.11 shows how all these can be brought together. First, you create a new actor system for the application. Next you connect to your database. Then, you construct your actor hierarchy of a `VoteWriter` supervised by a `WriterSupervisor`. With all these elements in place, you can now send votes to the database using your actor system. This app ends with printing the number of votes that the database has.

Listing 2.11 Full voting app

```
object Voting extends App {
  val system = ActorSystem("voting")

  val connection = new DatabaseConnection("http://remotedatabase")
  val writerProps = Props(new VoteWriter(connection))
  val writerSuperProps = Props(new WriterSupervisor(writerProps))

  val votingSystem = system.actorOf(writerSuperProps)

  votingSystem ! Vote(1, 5, "nom nom")
  votingSystem ! Vote(2, 7, "Mikey")
  votingSystem ! Vote(3, 9, "nom nom")

  println(connection.votes)
}
```

Instantiates a new actor system

Connects to the database

Creates a Props configuration object for the VoteWriter

Creates a Props for the WriterSupervisor

Creates actors from the Props objects

Sending messages in Akka is done using the ! method.

This approach definitely achieves some resilience in the face of failure. Moreover, it shows how you can build the possibility of failure into the application structure from the most trivial of beginnings. But you're probably not satisfied with this solution:

- The database uses mutability.
- The app can and does lose data.
- Explicitly building this actor hierarchy required you to think a lot about exactly how the underlying database might fail.
- Recording data in a database sounds like a common problem that someone else has probably already solved.

If you see these issues as deficiencies in this design, you're 100% right. This simple example isn't the ideal way to record your data in a database. All of chapter 3 is dedicated to a better approach to collecting and persisting data. There, I'll build on the approach you've seen in this chapter as a guide for how we want to interact with databases. We'll still use Akka actors, but we'll get to zoom up to a higher level of abstraction that keeps the focus on our application logic and not on low-level failure-handling concerns. The Akka toolkit is extremely powerful, and many libraries and frameworks make excellent use of it to build reactivity into applications. The next section introduces one of the best of those frameworks: Spark.

2.3 Spark, a reactive big data framework

Spark is the last tool we'll consider in this chapter. It's a framework for large-scale data processing, written in Scala. There are lots of reasons to use Spark to process large amounts of data:

- It's an incredibly fast data-processing engine.
- It can handle enormous amounts of data when used on a cluster.
- It's easy to use, thanks to an elegant, functional API.
- It has libraries that support common use cases like data analysis, graph analytics, and machine learning.

We'll use Spark in chapters 4 and 5 to build out one of the core components of a machine learning system: the feature-generation and model-learning pipeline. Spark is the textbook example of a reactive system, and later chapters explain much of the how and why of Spark's reactivity. But before we get into that deeper exploration of Spark, I want to get you started with a simple example problem: predicting the future.

The Spark framework is one of several high-level tools that we'll use in this book to build reactive machine learning systems. In this example, we'll tackle the problem of predicting the number of votes that the *Howlywood Star* systems will receive in the hours leading up to the close of voting. Historically, the last few hours of voting in a week are the most intense in terms of load, so the *Howlywood Star* engineering team wants to plan for the upcoming season's traffic spikes. You have at your disposal some historical files that describe basic data about the voting activity over time, and with that and Spark you can build a model to predict the number of votes that will be cast over time.

First, you'll need to do some basic setup to begin building a Spark app. Specifically, you'll create a configuration object to hold all your settings, and a context object that defines a specific execution context from those settings.

Listing 2.12 Basic Spark setup

```
val session = SparkSession.builder.appName("Simple ModelExample")
  ➥ .getOrCreate()
import session.implicits._
```

A session for your app

Imports serializers for primitive datatypes in your session

The `SparkSession` object you create here is used as your connection between regular Scala code and objects managed by Spark, potentially on a cluster. The next listings show how a `SparkSession` can be used to deal with things like I/O operations.

These setup steps give you a lot of options that are particularly relevant when you're scaling up to massive clusters. But for the moment, most of those capabilities are irrelevant, so we'll ignore them. In chapter 4 and beyond, we'll dig deeper into taking advantage of the rich possibilities Spark offers. For the moment, we'll take advantage of the simplicity of the API to get started with a minimum of boilerplate.

Load the data for the two datasets that you'll use to build the model.

Listing 2.13 Handling the data path

The path to where you put the example files from the repo

Where you want to write the modeling outputs of this pipeline

```
val inputBasePath = "example_data"
val outputBasePath = "."
val trainingDataPath = inputBasePath + "/training.txt"
val testingDataPath = inputBasePath + "/testing.txt"
val currentOutputPath = outputBasePath + System.currentTimeMillis()
```

Creates a unique path per execution to make rerunning simpler

In a model-learning pipeline, the data used to learn the model is called the *training set*. The data used to evaluate the learned model is called the *test set*.

One important detail to note about these datasets is that the training data comes from older shows than the testing data. In dealing with time series data like this, it's always important to cleanly separate the *in-sample* data used for training from the *out-of-sample* data used for testing. If you fail to do that, then your testing data no longer represents an expectation of the model's behavior when released on truly new data. In this case, I've prepared the data for you, so there's no danger of you committing this data-handling error, unless you mix up the training and testing sets.

Next, you need to load these files into Spark's in-memory representation of datasets, called *resilient distributed datasets* (RDDs). RDDs are Spark's core abstraction. They provide enormous amounts of data in memory, spread across a cluster without you having to explicitly implement that distribution. In fact, RDDs can even handle what happens when some data disappears due to cluster nodes failing, again without you having to concern yourself with handling this failure.

First, you'll load the data using a Spark utility for loading data. Then, you'll pass a function as an argument to the data.

Listing 2.14 Loading training and testing data

```
val trainingData = session.read.textFile(trainingDataPath)
val trainingParsed = trainingData.map { line =>
  val parts = line.split(',')
  LabeledPoint(parts(0).toDouble,  Vectors.dense(parts(1).split(' ')
  ➥ .map(_.toDouble)))
}.cache()

val testingData = session.read.textFile(testingDataPath)

val testingParsed = testingData.map { line =>
  val parts = line.split(',')
  LabeledPoint(parts(0).toDouble, Vectors.dense(parts(1).split(' ')
  ➥ .map(_.toDouble)))
}.cache()
```

Loads the data from a file. Common data formats are supported, including machine learning–specific ones.

The map function applies a higher-order function to the RDD.

A LabeledPoint is a machine learning–specific type designed to encode an instance consisting of a feature vector and a concept.

The cache method tells Spark you intend to reuse this data, so keep it in memory if possible.

Same code as the training set, repeated for the testing set, shown for clarity

Pure Functions

There are a few things worth calling out in this small code listing. First, it uses a higher-order pure function.

> **NOTE** Functions that can be passed as arguments to other functions are known as *higher-order functions*. Using them is a pervasive technique in Scala programming and Spark applications.

This is the standard pattern you'll use to interact with Spark RDDs. Spark uses this functional programming model whereby you pass functions to the data as a way of distributing data-processing workloads across a cluster. This isn't merely a syntax; Spark will literally serialize this function and send it to all the nodes storing the data in the RDD. Shifting from *sending data to functions* to *sending functions to data* is one of the changes you make when using big data stacks like Hadoop and Spark.

Laziness

Also, although it may not be clear, all the preceding Spark code is lazy. No data is read at the time that these commands are issued. Recapping from chapter 1, laziness is the intentional delay of execution. Ensuring the best possible performance in executing your data-processing task is a key part of Spark's strategy. By waiting until the last moment to evaluate, Spark can make informed choices about what data to process and where to send it.

Having loaded the data, you can now learn a linear model for the data using one of MLlib's learning algorithms. The following listing uses linear regression to derive that linear model.

Listing 2.15 Training a model

```
val numIterations = 100
val model = LinearRegressionWithSGD.train(trainingParsed.rdd,
➥ numIterations)
```

Max number of iterations the algorithm should run for

The model learned on the training set

At this point, you've merely learned a model—you have no idea whether that model is good enough for the critical problem of predicting future canine singing sensations. You can't use it to make predictions within the live system.

Linear regression

For those unfamiliar with linear models, they're simply a statistical technique for finding the line that most closely tracks all the data points. We won't get into the details of different model-learning algorithms until chapter 5. If you're interested in linear regression specifically, Wikipedia has a good introduction: https://en.wikipedia.org/wiki/Linear_regression.

To understand how useful this model is, you should now evaluate it on data that it hasn't seen—the testing set.

Listing 2.16 Testing a model

```
val predictionsAndLabels = testingParsed.map {
  ➥ case LabeledPoint(label, features) =>
  val prediction = model.predict(features)
  (prediction, label)
}
```

Another higher-order function, this uses pattern matching to destructure the LabeledPoints in the testing set.

Destructuring

Destructuring is a common technique in Scala and other functional programming languages. It's the inverse operation of instantiating a data structure. When used with pattern matching, it provides a convenient syntax for giving names to the pieces of a data structure.

Now that you've applied this model to new data, you can use more utilities from MLlib to report on the performance of the model.

Listing 2.17 Model metrics

A metrics object that can calculate various performance statistics

The precision of the model on the testing set

The recall of the model on the testing set

```
val metrics = new MulticlassMetrics(predictionsAndLabels.rdd)
val precision = metrics.precision
val recall = metrics.recall
println(s"Precision: $precision Recall: $recall")
```

Prints performance statistics to the console

Finally, you can save this model to disk.

Listing 2.18 Saving a model

```
model.save(session.SparkContext, currentOutputPath)
```

How to save learned models is a tricky topic once you get into it. I discuss it more in chapters 6 and 7. For the moment, having a saved version of the model on disk will be sufficient. It would be useful if you wanted to use this model on other data in the future.

The model could now be used to predict future vote counts. Given a feature vector, the model could be used to predict the expected number of votes. The *Howlywood Star* engineering team could plan around the possibility of peak load getting as high as the predicted value from the model.

That prediction may or may not be correct, but it's still useful for planning purposes. The metrics reported, the precision and the recall, give you some view into how much you can trust the predictions of the model. Later in the book, I discuss more-sophisticated approaches to modeling, and these approaches require you to reason more about the performance of learned models. But those approaches build on the process you've seen here: training a model and then evaluating it on a test set to see how it will perform. You'll even see how you can combine model metrics like these with reactive techniques to make machine learning systems that do some of this work for you.

Predicting the future is definitely hard work, but building the pipeline to do it is quite easy if you use the right tools. With the techniques shown in this chapter, the *Howlywood Star* team was able to build out a pretty impressive backend for a canine reality show. Using Spark, they were able to predict absurdly high levels of viewer engagement, leading to massive load on the voting system, and plan accordingly. But they weren't able to predict the breakout success of Tail-Chaser Swift, the winner of Season 2. Who could? That season, the audience loved poodles. They got to vote on every song, dance, and trick that the aspiring Howlywood Stars had to offer, thanks to a reactive system supporting the apps that they love.

Resources

This chapter has been a quick, shallow introduction to some incredibly complicated and powerful technologies. If you want to learn more about any of these three technologies, Manning has several relevant books for you:

- *Scala—Functional Programming in Scala*: www.manning.com/books/functional-programming-in-scala; *Scala in Action*: www.manning.com/books/scala-in-action; *Scala in Depth*: www.manning.com/books/scala-in-depth
- *Akka—Akka in Action*: www.manning.com/books/akka-in-action
- *Spark—Spark in Action*: www.manning.com/books/spark-in-action

Summary

- Scala gives you constructs to help you reason about uncertainty:
  - Options abstract over the uncertainty of something being present or not.
  - Futures abstract over the uncertainty of actions, which take time.
  - Futures give you the ability to implement timeouts, which help ensure responsiveness through bounding response times.
- With Akka, you can build protections against failure into the structure of your application using the power of the actor model:
  - Communication via message passing helps you keep system components contained.
  - Supervisory hierarchies can help ensure resilience of components.
  - One of the best ways to use the power of the actor model is in libraries that use it behind the scenes, instead of doing much of the definition of the actor system directly in your code.
- Spark gives you reasonable components to build data-processing pipelines:
  - Spark pipelines are constructed using pure functions and immutable transformations.
  - Spark uses laziness to ensure efficient, reliable execution.
  - MLlib provides useful tools for building and evaluating models with a minimum of code.

In the next chapter, we'll begin building out a real, large-scale machine learning system. I'll show you how to address the intrinsic uncertainty of real-world data in your data model. Also, I'll show how to use an immutable database of facts to achieve horizontal scalability. As part of that, I'll introduce a distributed database called Couchbase. Finally, I'll show how you can use reactive programming idioms to handle the uncertainties of processing time and failure.

Part 2

Building a reactive machine learning system

This is the heart of the book. This part will build up your knowledge of the components of a machine learning system, starting from raw data in the wild and looping all the way back around to acting on the real world.

Chapter 3 is about collecting data. It's not a normal chapter for a machine learning book: instead of hand-waving away where data comes from, we'll take a serious look at a range of data issues, concentrating on when that data is big, fast, and hairy.

Chapter 4 explores deriving useful representations of data, called *features*. This is one of the most important skills a machine learning systems developer can have and is often the largest part of the work.

Once you get to chapter 5, you should be ready to do what everyone focuses on in machine learning: learn some models. There are entire books about learning models, but this chapter represents a unique view and will give you an understanding of how this step connects to what came before and what comes after. I'll introduce you to some useful techniques to employ when the pieces of your system aren't as easy to join up as you'd like.

Chapter 6 covers the rich topic of how to make decisions about machine learning models that you've produced. Not all models are created equal. There's a range of common errors that you can make in learning about models, so I'll attempt to arm you with tools you can use to figure out the differences between a good model and a bad one.

Chapter 7 discusses taking the models you've produced and getting them somewhere they can be useful. Models sitting on your laptop probably aren't much use to anyone; they have to be available to be used by your customers, colleagues, and so on. This chapter shows you how to build services that can put your models to use.

Finally, in chapter 8, you get to use your models to affect the real world. This is where the rubber hits the road: you've closed the loop in fulfilling your user's request with a response. Reactive systems design is all about how you meet your users' expectations. So, in chapter 8, we turn our perspective solidly to the user of your machine learning system and see how you can realize those expectations.

3 Collecting data

This chapter covers

- Collecting inherently uncertain data
- Handling data collection at scale
- Querying aggregates of uncertain data
- Avoiding updating data after it's been written to a database

This chapter begins our journey through the components, or phases, of a machine learning system (figure 3.1). Until there's data in your machine learning system, you can't do anything, so we'll begin with collecting data. As you saw in chapter 1, the naive approach for getting data into a machine learning system can lead to all sorts of problems. This chapter will show you a much better way to collect data, one based on recording immutable facts. The approach in this chapter also assumes that the data being collected is intrinsically uncertain and effectively infinite.

Many people don't even mention data collection when they discuss building machine learning systems. At first glance, it doesn't seem as exciting as learning models or making predictions. But collecting data is crucial and a lot harder than it looks. There are no easy shortcuts to building production-grade apps that can collect vast amounts of highly variable data in an environment of change. We need to

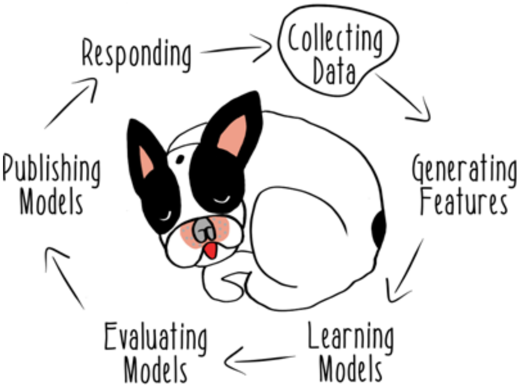


Figure 3.1 Phases of machine learning

bring the full power of reactive machine learning to bear on this problem to ensure that we have good, usable data that can be consumed by other components of our machine learning systems.

To go deeper into the world of reactive machine learning systems, we're going to have to go beyond the problems of mere house pets. We'll have to venture as far as the wilds of Africa. The challenge you'll take on is recording the movements of animals during the largest terrestrial mammal migration on the planet. The Serengeti's Great Migration is full of data that's big, fast, and hairy. Let's see how that can data be collected.

3.1 Sensing uncertain data

In this chapter, you'll play the role of a noble lion queen. As matriarch of your pride, you take your job very seriously.

You have an age-old problem, though: your food doesn't stay still. Every spring, the wildebeests and zebras that you feed on have the annoying habit of leaving the southern grassland plains to migrate north. Then, every fall, these herbivores just turn around and head back south for the rainy season.

As a responsible queen and mother, you must track the movements of this mass migration of your prey. If you didn't, your pride would have nothing to eat. But the data-management problems of this job are severe, as shown in figure 3.2.

To get any sort of handle on this big, fast, and hairy data, you're going to need to deploy some advanced technology. You have a long-term vision that you'll one day be able to use this data to build a large-scale machine learning system that will predict where the prey will be next, so that your lionesses can be there waiting for them. But before you can even consider building systems on top of this data, you need to collect it and persist it somehow.

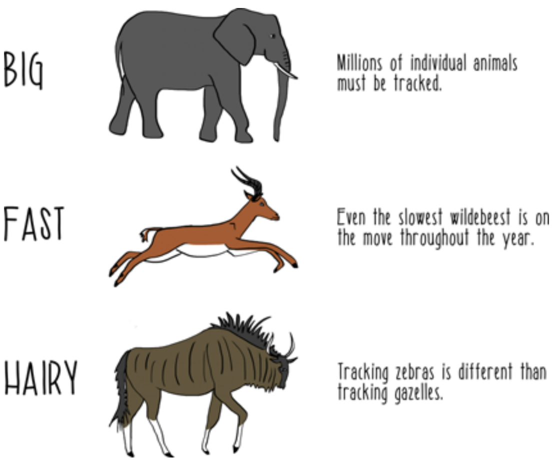
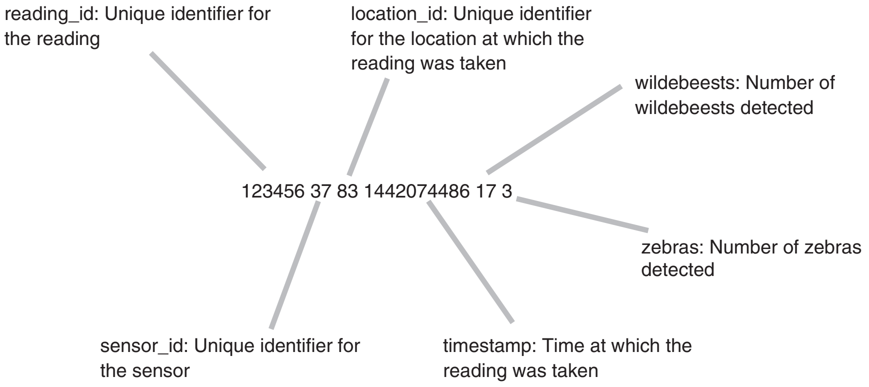

BIG
Millions of individual animals must be tracked.
FAST
Even the slowest wildebeest is on the move throughout the year.
HAIRY
Tracking zebras is different than tracking gazelles.

Figure 3.2 Great Migration data

Thanks to a recently signed contract with technology consultants Vulture Corp., you now have access to some sensor data about the movement of land-bound animals (figure 3.3).

Figure 3.3 Vulture Corp.

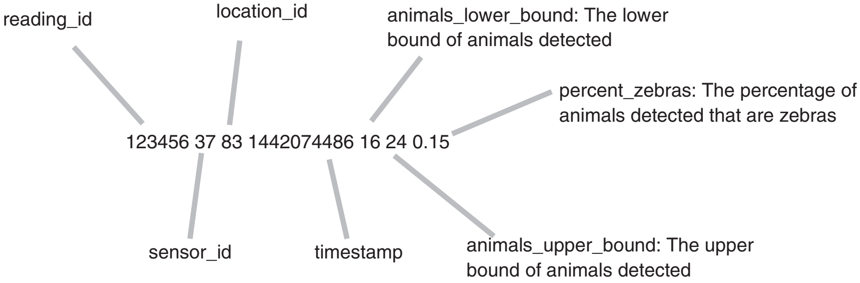
The Vulture Corp. Eyes in the Skies system is based on an aerially deployed, distributed network of sensors. These sensors use a combination of body-heat detection and movement patterns to report back the number and kinds of different animals at any given location. Here's an example of the sort of raw animal-sensor data that this system provides:


reading_id: Unique identifier for the reading
location_id: Unique identifier for the location at which the reading was taken
wildebeests: Number of wildebeests detected
zebras: Number of zebras detected
timestamp: Time at which the reading was taken
sensor_id: Unique identifier for the sensor

```
123456 37 83 1442074486 17 3
```

But this raw schema isn't really what the Eyes in the Skies system understands about the animals below. The sensors only provide an approximate view of what's going on using body heat and movement. There will always be some uncertainty in that process.

After some further negotiation with (intimidation of) your technology consultants, you get access to a richer data feed. That data feed, shown here, is more explicit about the difficulty of being precise in raw sensor data like this:


reading_id
location_id
animals_lower_bound: The lower bound of animals detected
percent_zebras: The percentage of animals detected that are zebras
123456 37 83 1442074486 16 24 0.15
sensor_id
timestamp
animals_upper_bound: The upper bound of animals detected

This data model has far more statistical richness. It expresses that there may have been far fewer or far more than 20 animals at that location. A consultation of the Eyes in the Skies API documentation reveals that these are the upper and lower bounds of the probability distribution of the sensed data, at a 95% confidence level.

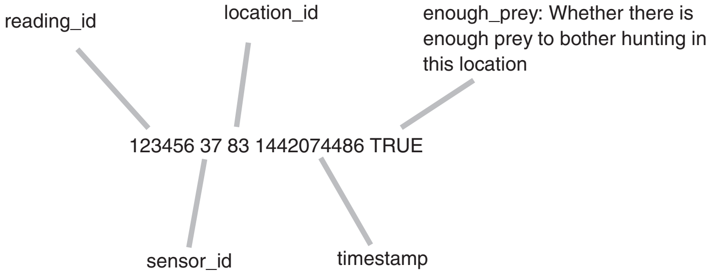
Confidence intervals

The uncertain data model shown in the diagram uses a *confidence interval*. This is a measure of the amount of uncertainty around the sensor reading. The confidence level refers to the percentage of all possible readings that would be expected to include the true count of animals. Right now, you don't need to be concerned with how these values are calculated—we won't spend much time on basic statistical techniques like confidence intervals in this book. Instead, we'll focus on how to build systems that are aware of the need to express and respond to uncertainty. For a more in-depth introduction to statistics, many books and courses are available. *Think Like a Data Scientist* by Brian Godsey (Manning, 2017) is a good book for building a deeper understanding of statistics, in the context of doing data-science work.

The difference between 16 and 24 prey animals might not sound big, and in some contexts it might not be. Some readings have lower bounds that are zero and upper bounds that are nonzero. For those locations, you could send lionesses to them expecting to find one or two wildebeests and find none at all when they arrive. Thanks to this explicitly uncertain data model, you as the lion queen can now make more-informed decisions about where to allocate your scarce hunting resources.

The difference between these two data models is an example of a *data transformation*. In a data-processing system, as in a machine learning system, many of the operations of the system are effectively data transformations. Transforming data is such a common task that I'll spend all of chapter 4 discussing how to do it in a reactive machine learning system. The original sensor data feed from the Eyes in the Skies system was a transformation of a more raw form of the data that was originally kept internal to Vulture Corp. As you saw, transforming this raw data caused you to lose information about the intrinsic uncertainty in the sensor readings. This is a common mistake that people make when implementing data-processing systems: people destroy valuable data by only persisting some derived version of the raw data.

Consider the following proposal from a young lioness working on the transformed sensor data from the Eyes in the Skies system:


reading_id
location_id
enough_prey: Whether there is enough prey to bother hunting in this location
123456 37 83 1442074486 TRUE
sensor_id
timestamp

In this heavily transformed version of the data, the young lioness developer has decided to simplify things down to just a Boolean value representing whether there are more than 10 wildebeests in a given location. That's the cutoff value that you've been using lately to decide whether a location should be hunted. Her thinking is that this is all you really need to know to make a decision anyway.

But you're an experienced lion queen. In a bad year, you might go out to stalk a single zebra foal. The circle of life is always turning, and you can't always know what the future will bring. You may need all the richness of the data model in the previous diagram to make the hard decisions when the time comes.

This is an illustration of a fundamental strategy in data collection. You should always collect data as immutable facts.

A *fact* is merely a true record about something that happened. In the example of the Eyes in the Skies system, the thing that happened was that an aerial sensor sensed some animals. To know when that fact occurred, you should record the time when it occurred, although there are some interesting choices about how that time can be expressed. The example so far has used a simple timestamp to say when a sensor reading was recorded.

Similarly, it's often a good idea to record the entity or entities that the fact relates to. In the animal-sensor data, you recorded both the entity from which the fact originated—the sensor—and the entity being described in the fact: the location. Even though there was some uncertainty around the sensor data collected, the facts will never need to be changed. They will always be a fact about the system's view of the world at that point in time.

Immutable facts are an important part of building a reactive machine learning system. Not only do they help you build a system that can manage uncertainty, they also form the foundation for strategies that deal with data-collection problems that only emerge once your system gets to a certain scale.

3.2 Collecting data at scale

The amazing thing about the Great Migration is its scale. Millions of animals are all on the move at once. The wildebeests are the most numerous of these animals, but there are also hundreds of thousands of gazelles and zebras. Beyond these top three meals on hooves, there's a long tail of lesser animals to consider. From high up on your headquarters on Lion Rock, you can only see so much. The Eyes in the Skies system has given you a starting point for understanding the state of the savannah, but it's just a starting point. It's clear that you need to begin processing this data into more useful representations if you want to be able to take action on any of it.

You'll start with exploring how to build *data aggregations*, derived data produced from multiple data points. The next section shows an initial approach toward building a data-aggregation system that won't be the final strategy you'll use. The techniques used will all work at a certain level of scale but then run into problems as scale increases. These ways of building a distributed data-processing system are the beginnings of a solution to problems of scale, and you'll build on this section to handle even greater scale in section 3.3.

3.2.1 Maintaining state in a distributed system

As a first project, you've decided to try to track the density of prey in each region. A region is a geographically contiguous set of locations, each with its own sensor feed. The density statistic that you'd like maintained could be simply expressed as in the following listing.

Listing 3.1 Calculating location densities

Case class representing a reading of number of animals at a particular location and size of location

Example instance of a reading

Collection of readings

Sum of all animals in a region

```
case class LocationReading(animals: Integer, size: Double)
val reading1 = LocationReading(13, 42.0)
val reading2 = LocationReading(7, 30.5)
val readings = Set(reading1, reading2)
val totalAnimals = readings.foldLeft(0)(_ + _.animals)
val totalSize = readings.foldLeft(0.0)(_ + _.size)
val density = totalAnimals/totalSize
```

Density in terms of animals/square mile —27.6 in this case

Sum of square miles of all locations in a region

This sort of approach is reasonable in some ways. The code in listing 3.1 uses only immutable data structures, so all values used in calculations can be thought of as facts.

Folding

The summing operations in listing 3.1 are implemented using a fold. *Folding* is a common functional programming technique. The `foldLeft` operator used begins with an initial sum of zero. The second argument is the higher-order function to be applied to each item in the set of readings. In a sum, this higher-order function is the addition of the running sum with the next item. But folding is a powerful technique that can be used for more than just summing. You'll see it again, particularly in later chapters that use Spark. If you want to dive deep into why folding is such a powerful technique, check out the paper "A Tutorial on the Universality and Expressiveness of Fold" by Graham Hutton: www.cs.nott.ac.uk/~pszgmh/fold.pdf.

Summing is done using a pure, higher-order function, +, so there's no chance of a side effect in the summing function causing unforeseen problems. But it's not yet time for the lions to sleep tonight. Things are going to quickly get trickier.

Figure 3.4 Cheetah Post

For you to know which regions have the most density of prey without going out to where the sensor readings are being reported, you'll need to aggregate all those readings in a single location—your headquarters at Lion Rock. To that end, you've signed a contract with the Cheetah Post message-delivery company (figure 3.4).

They'll go out to each of the data-collection stations and get the latest reading. Then they'll rush the message about that information back to Lion Rock. That latest sensor reading will then be added to the aggregate view of all the locations.

Anticipating problems with a bunch of cheetahs running back and forth, you decide to do what any seasoned leader would do: you put a pangolin in charge. In exchange for you agreeing not to eat him, the pangolin has agreed to maintain the current state of the density data as part of the system shown in figure 3.5.

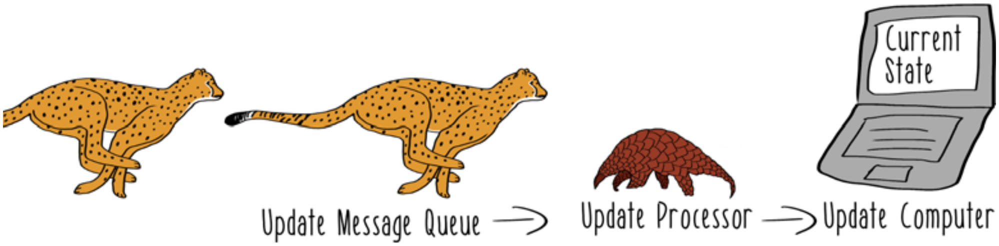


Figure 3.5 Simple density-data system architecture

The pangolin implements the state management process in listing 3.2, which shows an example of how this aggregate view of the savannah can be maintained. The example update scenario is the receipt of a message about there being a high density of animals in Region 4.

Listing 3.2 Aggregating regional densities

```
case class Region(id: Int)

import collection.mutable.HashMap
var densities = new HashMap[Region, Double]()

densities.put(Region(4), 52.4)
```

Case class representing a region

Mutable hash map storing the latest density values recorded by region

This update will overwrite the previous value for Region 4 with a new value.

By putting a single pangolin in charge of this process, you've ensured that the cheetahs will never contend to make updates. Moreover, by making all the cheetahs line up to talk to the pangolin, you've ensured that all updates are processed in the order of their arrival.

But the number of animals constantly on the move leads to there being more cheetahs with more updates than you originally planned for. The process of recording these updates quickly becomes too much for one pangolin to handle.

You decide to hire another pangolin. Now there are two pangolins and two queues that cheetahs can line up in to get their updates applied, as shown in figure 3.6.

At first this seems like a good solution. After all, you can keep hiring more pangolins as your scale of data collection goes up. That gives you some ability to continue applying your updates in the same amount of time despite increasing load.

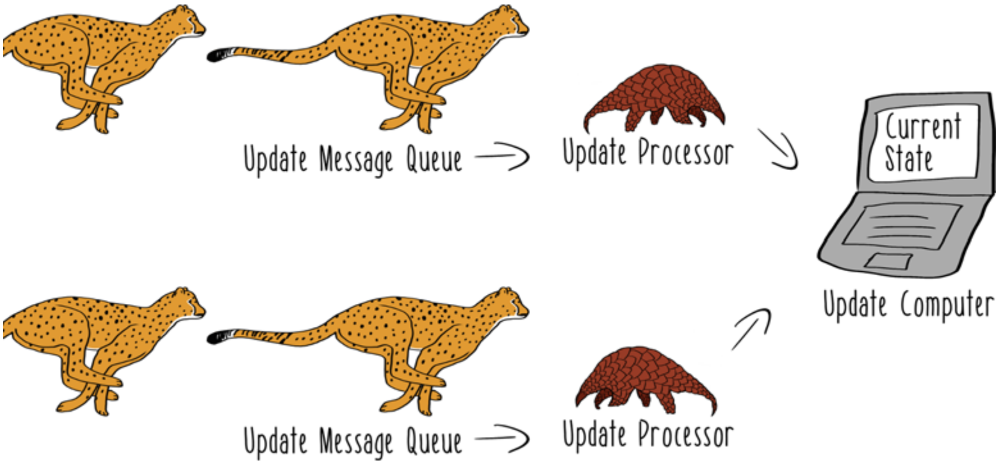


Figure 3.6 Adding a queue

But that initial elasticity quickly fades. Part of the reason is that while one pangolin is making an update to the system, the other pangolin can only wait. Although pangolins do spend some time walking back and forth from queues of cheetahs to the update computer, they soon spend most of their time waiting for access to the single computer. This means updates about one region block updates about another region.

You decide to try adding more pangolins with more computers and implement the system shown in figure 3.7.

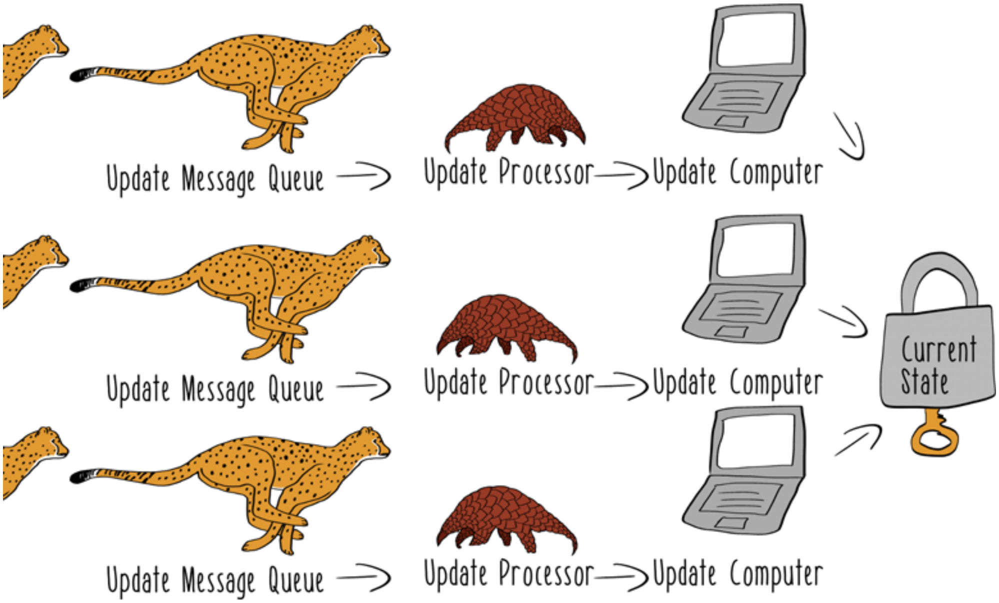


Figure 3.7 Concurrent access to a shared mutable state

To enable multiple pangolins to make updates concurrently, you decide to change the data structure used to store the densities data.

Listing 3.3 Concurrently accessible densities

```
import collection.mutable._

var synchronizedDensities = new LinkedHashMap[Region, Double]()
➥ with SynchronizedMap[Region, Double]
```

This implementation now allows for concurrent access to make updates to densities data using a system of locks that ensures that each thread of execution has the latest view of the data. Different pangolins can be at different computers making different updates, but each pangolin holds a lock for the time it takes to make their update. At first, this looks like an improvement, but the performance eventually is much like the old system. The synchronization process and its locking mechanism turn out to be quite similar to the old single-computer bottleneck. You've merely narrowed the scope of what the scarce resource is down to a lock on the mutable data structure. With this bottleneck, you can no longer add more pangolins to get more throughput; they would just contend for locks on the densities hash map.

There's another unfortunate outcome of this new system. Cheetahs can get in any queue they want. Some pangolins work faster than other pangolins. This system now allows some updates to be processed out of order.

For example, Region 6 of the savannah had a high density of animals this morning before all the zebras moved on. If the updates about these sensor readings are applied in order, you'll have an accurate view of this region, as shown in the following listing.

Listing 3.4 In-order updates

```
densities.put(Region(6), 73.6)
densities.put(Region(6), 0.5)
densities.get(Region(6)).get
```

Morning update

Afternoon update

Returns 0.5

But now it's also possible for updates to be applied out of order. A sequence of out-of-order updates gives you a very different view of the situation.

Listing 3.5 Out-of-order updates

```
densities.put(Region(6), 0.5)
densities.put(Region(6), 73.6)
densities.get(Region(6)).get
```

Afternoon update

Morning update

Returns 73.6

In this second scenario, you're sending out your valuable lionesses to go hunt in an area where you should already know that all the animals have moved on.

If you look back at the first sequence of updates, it also has deficiencies. In the afternoon, you have an accurate view of the lack of prey in Region 6 if the updates are applied as in listing 3.4. But what happened in the morning? There should have been lionesses out there in such a prey-rich region, but they were just lounging around. By the time the afternoon rolled around, all you knew was that there was no more prey in Region 6. You had no idea that a few lazy lionesses missed the day's best opportunity to hunt. There has to be a better way of organizing a hunt.

3.2.2 Understanding data collection

Where did things go wrong with the prey-density project? This system was supposed to answer basic questions about where animals were located on the savannah. Sure, the vulture consultants had proposed a follow-on project where this prey data would be used to create machine learning models of future prey locations. But you can't even begin to consider future projects without figuring out what's wrong with the current system and fixing it.

You gather your team to dissect the prey-density system and figure out what you've learned about collecting data. You reach the following conclusions:

- Data models that simplify real uncertainty throw away valuable data.
- A single data-collection processor can't be scaled out to handle your real workload.
- Distributing your workload doesn't scale much better if you use shared mutable state and locks.
- Using mutation to update data destroys historical knowledge and can even cause you to overwrite new data with older data.

The team had all worked so hard to get this prey-data-collection system online, and yet it had some very real shortcomings. Are they going to be up for the challenges of fixing this system and taking it to the next level? They don't call a group of lions a *pride* for nothing! Of course they can build on what they've learned. They've learned a lot about how to collect data. They're ready for the next phase of building out a data-collection app: storing this data.

3.3 Persisting data

To build the rest of your data-management systems, you'll need a database. As you'll see in the next chapter, machine learning pipelines usually start with some database of raw data. Although it may seem obvious, let's try to understand what you need out of a database. You may already know some of this material, but bear with me while I explain databases from the perspective of reactive systems.

Often people will discuss databases in terms of the operations that they support, so first and foremost, a database should allow you to store your data. In database terminology, this is usually called the *create* operation. For a database to store your data, that data must ultimately be *persisted*—it should still exist in the database after your program shuts down. Persistence also means that your data should survive things like a database server restart.

The other thing a database needs to do for you is return your data when you ask for it. This is referred to as the *read* operation. In reactive terms, this is just responsiveness again.

No other property of a database would matter if it didn't consistently return timely responses to your queries. People have different ideas about how data can and should be read from a database, and we'll consider those options later in this chapter.

There are some other things that databases can do that you won't need to do. Some databases support the `update` operation, which changes data. As you've seen, mutating data can lead to all sorts of problems, so you're going to avoid the `update` operation. Instead of changing data that has already been persisted, you'll rely on writing new immutable facts.

Similarly, some databases support an operation called `delete`. I know! That's just horrifying. In the reactive machine=learning paradigm, we assume that our data is effectively infinite.

You won't need that misguided `delete` operation, because you'll build your system to handle unbounded amounts of data from the very beginning. Now let us never speak of deleting again.

3.3.1 Elastic and resilient databases

Now that you understand some basics of how you're going to use a database, let's get more specific about what will make a database work well in a reactive data-processing system. After all, you have a grand vision that you'll eventually be able to use this prey data to build a large-scale machine learning system that will predict future prey movements. With all this big, fast, hairy data, you need to consider the reactive principles in your technology selection choices if you ever want to get that far.

As you saw in the initial attempt to collect the prey data, achieving elasticity is hard. You can't just add more data-processing units, but that idea is on the right track.

You'll need a *distributed database*, one with multiple servers working in concert to function as a single database. Rather than a bunch of cheetahs contending to write to the same pangolin-controlled computer, you'll ideally use a database where multiple servers can write as well as read, something like figure 3.8. True distribution of both `create` and `read` operations is the only way a database can scale.

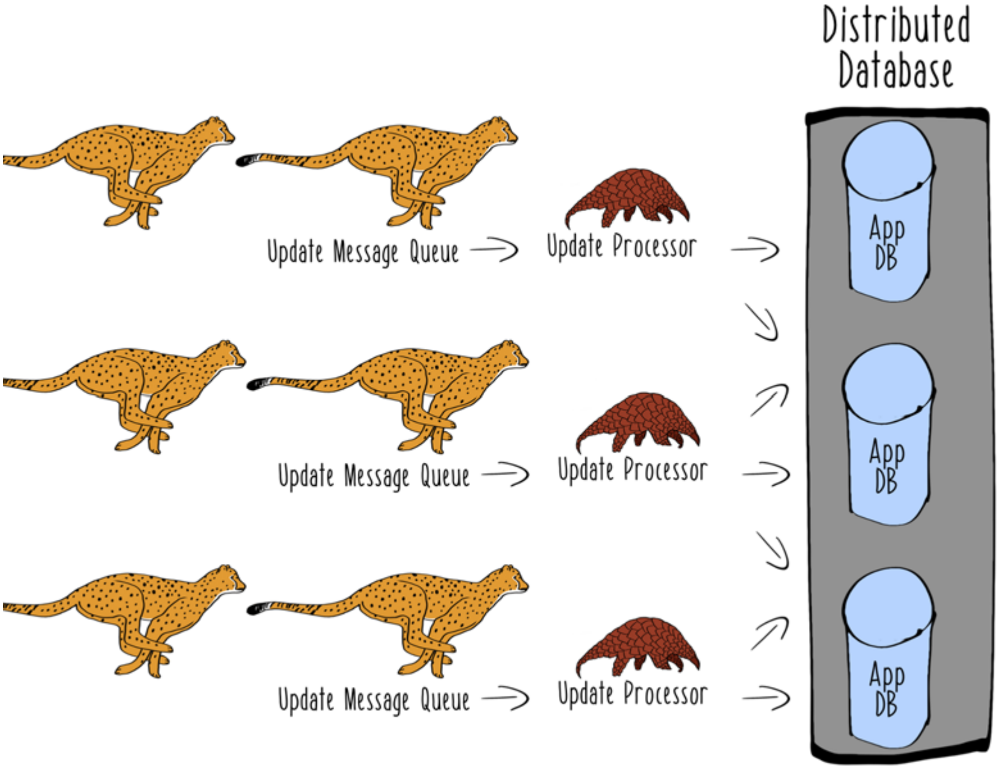


Figure 3.8 A distributed database

The other reactive principle to consider is resilience. In the Serengeti, you know that resources can disappear. Some days the rivers will flood and water will be abundant. Other times, the ground will be scorched by the blazing African sun, and you'll need to walk for days to find water.

Unfortunately, this is exactly how things work for distributed databases. With many servers responsible for storing and retrieving your data, it's inevitable that one of them will fail. Maybe it will just fall asleep for a while like a lazy lioness neglecting her hunting duties for a midday nap. Or maybe it will be worse, and that server will be like Patrick Pangolin, who was last seen in the mouth of a hungry cheetah. That's the circle of life.

You wind up in the same place that you were in your quest for elasticity: you need a distributed database, where your data is stored redundantly on multiple servers. Replication is a consistently relevant strategy when building reactive machine learning systems.

Thankfully, there are all sorts of databases that have many of the properties I just described. The one that we'll use in this book is called Couchbase. It's a distributed database that can both handle the scale of the Great Migration and deal with the inevitable failure of servers along the way. Couchbase has a wealth of capabilities, far beyond your minimum requirement to create and read large numbers of records. In fact, many other databases would work for the example in this chapter. The techniques of reactive machine learning are not tied to any specific technology. But Couchbase will make it easy for you to get started building your database of facts and could easily support any future projects like a prey-movement prediction system. As an additional benefit, Couchbase itself is a very reactive system, implemented using several reactive strategies. Later in this chapter, we'll take a quick look at how Couchbase, as a highly reactive database, can support building reactive data-processing systems.

3.3.2 Fact databases

One of the tools you'll use to scale the prey-movement system is a *fact-based data model.* Earlier in this chapter, I talked about facts as a useful technique for raw-data capture. You can express the enriched data model you saw earlier in the form of a standard case class.

Listing 3.6 Sensor-readings case class

```
case class PreyReading(sensorId: Int,
                       locationId: Int,
                       timestamp: Long,
                       animalsLowerBound: Double,
                       animalsUpperBound: Double,
                       percentZebras: Double)
```

This case class represents an individual sensor reading.

You'll store your data in the form of JSON documents. Databases that support this style of data model are sometimes called *document stores*. Couchbase uses a document data model where the documents are stored inside buckets. *Buckets* serve a similar purpose as tables do in traditional relational databases, but they don't require the definition of the structure of those documents to be defined in the database itself. They'll accept any documents that you choose to write. You don't need to do anything to the database before it can accept documents of your readings for persistence. Not having to plan all aspects of your data model in advance can make it easier to deal with evolution in your data model, such as when you added richer information about sensor uncertainty. Chapter 11 discusses more about handling the evolution of a reactive machine learning system.

To persist instances of the sensor-reading case class, you have to define a formatter that can convert those instances into an equivalent JSON representation that can be stored in the database.

Listing 3.7 Creating sensor-reading documents

```
import play.api.libs.json._
import scala.concurrent.ExecutionContext
import org.reactivecouchbase.ReactiveCouchbaseDriver

val driver = ReactiveCouchbaseDriver()
val bucket = driver.bucket("default")
implicit val ec = ExecutionContext.Implicits.global

implicit val preyReadingFormatter = Json.format[PreyReading]

def readingId(preyReading: PreyReading) = {
  List(preyReading.sensorId,
    preyReading.locationId,
    preyReading.timestamp).mkString("-")
}

val reading = PreyReading(36, 12, System.currentTimeMillis(), 12.0, 18.0, 0.60)

val setDoc = bucket.set[PreyReading](readingId(reading), reading)

setDoc.onComplete {
  case Success(status) => println(s"Operation status: ${status.getMessage}")
  case _ => throw new Exception("Something went wrong")
}
```

Creates a database connection to the default bucket

Execution context to use with futures

Formatter to use to convert the case class into JSON

Helper function to create a composite primary key for PreyReading documents

Example sensor reading

Operation to insert reading as a document that returns a Future

Prints the result of the insertion operation for illustrative purposes

Implicits

You may have noticed that listing 3.7 uses *implicits*. The implicit formatter you created defines a way of converting the `PreyReading` case class into JSON. Had you not created this formatter, the library you use to interact with the database wouldn't know how to perform this conversion and wouldn't be able to save instances of sensor readings to the database. The `implicit` keyword makes that conversion available for use without requiring you to explicitly perform the conversion yourself. The compiler will infer that the formatter should be used to perform this conversion during the compilation of your program and insert the conversion code. Implicits are a unique and powerful feature of Scala. You'll come across them all the time in idiomatically written Scala code. For a more thorough introduction to implicits and their use, check out *Scala in Depth* by Joshua D. Suereth (Manning Publications, 2012).

Separating the definition of an action from the execution of that action is often a helpful idiom. In this case, it helps you encode that the insertion both is going to take time and could possibly fail. For example, you could replace the failure case in the pattern match with retry logic or some meaningful notification. Recognizing and encoding the possibility of failure is a key step toward building resilience into a system.

This style of database interaction relies on futures, a technique you saw in chapter 2. One of the main benefits of this programming style comes out of operations like the database insertion being non-blocking. The call to `bucket.set` returns immediately. Because insertions into a remote database take time, the driver doesn't tie up the main thread of execution of your program waiting on the data to travel to the remote database and a successful insertion message to come back. This future-based, non-blocking programming style works well with the goal of consistent operation under varying load.

There's more in this approach to data collection that supports elasticity. Many different data-collection program instances can be writing to many different database nodes at the same time without any need to lock individual items and coordinate access. This is similar to the final architecture of multiple cheetahs talking to multiple pangolins that you saw in the previous section, but it's even better. Thanks to the power of a non-blocking driver, it's almost like the cheetahs can just drop off their messages and run away. They don't need to wait for the slow pangolin to actually make the update. The cheetahs certainly don't need to wait for the pangolins to coordinate access to the shared mutable state object among themselves, because there's no shared mutable state. But how do you figure out the current state of the savannah from a database full of raw facts?

3.3.3 Querying persisted facts

The easiest way to see data in your database is to define some structure on top of the data you've inserted. Being able to define this structure after you've inserted the data is one of the unique features of modern, flexible databases, in contrast to the more rigid relational databases you may be familiar with. You've been recording your data as JSON documents, and now you need to express some information about the structure of your documents using JavaScript. Don't worry if you're not familiar with JavaScript. You'll only be writing simple JavaScript to define the structure of views on top of raw data. Listing 3.8 defines a view on top of the data you've written to your database instance that allows you to retrieve documents by sensor ID. That view will be defined in terms of a *design document*, which is another way of saying *stored query*.

Listing 3.8 Creating a sensor ID view

```
import scala.concurrent.Await
import scala.concurrent.duration.Duration
import java.util.concurrent.TimeUnit

val timeout = Duration(10, TimeUnit.SECONDS)

Await.result(
  bucket.createDesignDoc("prey",
    """
      | {
      |     "views":{
      |        "by_sensor_id": {
      |            "map": "function (doc, meta) { emit(doc.sensorId, doc); }
      |        }
      |     }
      | }
    """.stripMargin), timeout)
```

Blocks to wait for the future to complete, for illustrative purposes

Creates a design doc (in JavaScript) called prey

Defines views in this design doc

Creates a view by sensor ID

Creates a function to emit all documents with a sensor ID

Blocking

I've been calling out the benefits of non-blocking, futures-based interactions, so it may seem strange to be blocking to see these benefits. The reason you may want to use blocking in a small, exploratory context like this is that it allows you to see the results of the small piece of code you're working on by forcing the future to complete. In a fully implemented and properly composed system, you wouldn't want to rely on calls to `Await.result`. You'd want to use this technique primarily in an exploratory or debugging context.

By defining this view, you've created indexes on documents by their sensor IDs. This will make it simple and fast to look up documents by sensor ID.

In a fully populated database backing a real application, this view creation could take a meaningful amount of time to create the necessary data structures to return the data expressed in the view. In Couchbase, this issue is managed by separating a development view, what you've just created, from a production view. Different distributed databases choose to implement views differently or not at all, so it's worth understanding the specifics of the database that you're using when implementing a real system. But this query-centric workflow is common to a wide range of distributed non-relational databases.

Note that Couchbase views are defined in terms of higher-order functions using a map-reduce syntax, with both the map and reduce phases being expressed as higher-order functions. Defining views in terms of map-reduce operations may feel strange to you if you have experience with views in relational databases. Modern distributed non-relational databases often blur the lines between the data processing done by the application and the data processing done by the database. Certainly, you could implement this exact same view in your application code in Scala, but that would require you to effectively process the entire contents of the database for any given query. When your use of a distributed database reduces to a brute-force full-table scan for a query, usually there's something wrong at the design level in your application (or the database!). It's better to have the database take on this work itself, if for no other reason than that the view-maintenance work will be done only once per view rather than once per query.

Once you have this view, you can retrieve that last reading that you recorded in listing 3.7. To do so, you can just find the reading with the matching sensor ID, using the code in the following listing.

Listing 3.9 All records for a sensor

```
val retrievedReading = Await.result(
    bucket.searchValues[PreyReading]
      ("prey", "by_sensor_id")
      (new Query().setIncludeDocs(true)
        .setKey("36"))
      .toList,
    timeout)
    .head

println(retrievedReading)
```

Searches for PreyReading values

Design document and view to use

Creates a new query

Defines the key of the document queried to be for sensor 36

Forces the result to a list

Takes the first element

Prints the retrieved reading

If you only wrote the sensor reading from listing 3.7, then this query will just operate over that one document. But very similar querying syntax will return all the readings that have ever been recorded for that sensor.

To see how operations on a large sequence of facts work, the next listing creates some synthetic data to play with.

Listing 3.10 Inserting many random records

```
import _root_.scala.util.Random
import play.api.libs.iteratee.Enumerator

val manyReadings = (1 to 100) map { index =>
  val reading = PreyReading(
    36,
    12,
```

Produces 100 random sensor readings

```
    System.currentTimeMillis(),
    Random.nextInt(100).toDouble,
    Random.nextInt(100).toDouble,
    Random.nextFloat())
  (readingId(reading),
    reading)
}

val setManyReadings = bucket.setStream(Enumerator.enumerate(manyReadings))

setManyReadings.map { results =>
    results.foreach(result => {
      if (result.isSuccess)
      ➥ println(s"Persisted: ${Json.prettyPrint(result.document.get)}")
      else println(s"Can't persist: $result")
    }
    )
  }
```

Inserts all the random readings as a stream

Maps over each result and prints the result

Now you should have a lot more facts in your database to work with.

Enumerators

Listing 3.10 uses `Enumerator` from the Play web framework. An *enumerator* is a source of data that pushes input into some recipient. Enumeration is simply operating over each item in the collection one by one. When complete, a Play `Enumerator` will return the final state of the recipient, using a futures-based programming technique called a `Promise` (discussed later in this book). In this case, you're using enumerators as way of sending a stream of values to the database for persistence. In a real system, the data being enumerated would come from multiple sensors sending back sensor-reading data.

You can use this database of facts to answer questions about the current state of the savannah. For example, you can define a time-based view of readings from sensor 36 using the view in the following listing.

Listing 3.11 A time-based view of sensor readings

```
Await.result(
  bucket.createDesignDoc("prey_36",
    """
      |{
      |    "views":
      |    {
      |        "by_timestamp":
      |        {
      |            "map":
      |            "function (doc, meta)  { if (doc.sensorId == 36)
      ➥ { emit(doc.timestamp, doc); }  }"
```

Defines this view for sensor 36 only

```
        |        }
        |    }
        |}
      """.stripMargin), timeout)
```

Because you haven't thrown away any data through mutating some state, it's easy to get a picture of the recent readings and the way things have been trending.

Listing 3.12 The 10 most recent sensor readings

```
val lastTen = Await.result(
  bucket.searchValues[PreyReading]
    ("prey_36", "by_timestamp")
    (new Query().setIncludeDocs(true)
      .setDescending(true)
      .setLimit(10))
    .toList,
  timeout)
```

Orders by the most recent readings

Takes only the 10 most recent readings

Or you can jump back to a specific point in time.

Listing 3.13 An individual old sensor reading

```
val tenth = Await.result(
  bucket.searchValues[PreyReading]
    ("prey_36", "by_timestamp")
    (new Query().setIncludeDocs(true)
      .setDescending(true)
      .setSkip(9)
      .setLimit(1))
    .toList,
  timeout)
```

Skips the nine most recent readings

Only returns the tenth most recent reading

It's important to note that this design gets around the problems shown with out-of-order updates you encountered in listing 3.5. In the previous system, because you had scaled the data-collection system out to have multiple cheetahs bringing updates to multiple pangolins, an old update could get applied over a newer one and destroy any record of that fact. In the revised system, that's not possible because you store all sensor readings without ever throwing any away. The code in the following listing shows how to intentionally insert the records out of order.

Listing 3.14 Inserting out-of-order readings

```
val startOfDay = System.currentTimeMillis()
val firstReading = PreyReading(36, 12, startOfDay, 86.0, 97.0, 0.90)
val fourHoursLater = startOfDay + 4 * 60 * 60 * 1000
val secondReading = PreyReading(36, 12, fourHoursLater, 0.0, 2.0, 1.00)

val outOfOrderReadings = List(
  (readingId(secondReading), secondReading),
```

First reading collected

Second reading collected

```
    (readingId(firstReading), firstReading)) <— Defines a list of readings in the wrong order

val setOutOfOrder = bucket.setStream(Enumerator.enumerate(outOfOrderReadings))  <—

setOutOfOrder.map { results =>                       Inserts readings out of order
  results.foreach(result => {
    if (result.isSuccess)
    ➥ println(s"Persisted: ${Json.prettyPrint(result.document.get)}")
    else println(s"Can't persist: $result")
  }
  )
}
```

Insertion order doesn't matter now, so all the queries shown here work exactly the same whether the records are inserted in order or out of order. You can see this by retrieving the last two sensor readings.

Listing 3.15 Retrieving the last two readings

```
val lastTwo = Await.result(
  bucket.searchValues[PreyReading]
    ("prey_36", "by_timestamp")
    (new Query().setIncludeDocs(true)
      .setDescending(true)
      .setLimit(2))
    .toList,
  timeout)
```

This query returns the same result regardless of whether the records were inserted in order or out of order, because no data is lost and the sortation is determined by the query. You no longer have to risk an out-of-order update corrupting your view of the savannah by writing over old data with new. You're free to scale out your data-collection operation to arbitrary scale without concern for insertion order of updates.

3.3.4 Understanding distributed-fact databases

Immutable Facts

Now that you've gotten your prey-data collection under control, it might help to zoom out and understand why this alternative approach works so much better. First, you've kept all the richness of your knowledge of the source data. Sensor readings are uncertain, and that uncertainty is persisted in your database.

By not transforming your raw data before persisting it, you maintain a true record of the data the sensors collected. This history of facts is much richer and more useful than a single mutated value. An immutable-fact database can tell you anything you'll ever want to know about the state of the savannah.

Next, you achieved true horizontal scalability thanks to the combination of a good data architecture and a very elastic database. You can continue to add data-collection applications, and your data model and your database will now support arbitrary increases in scale. In this respect, Couchbase gives you even stronger guarantees than

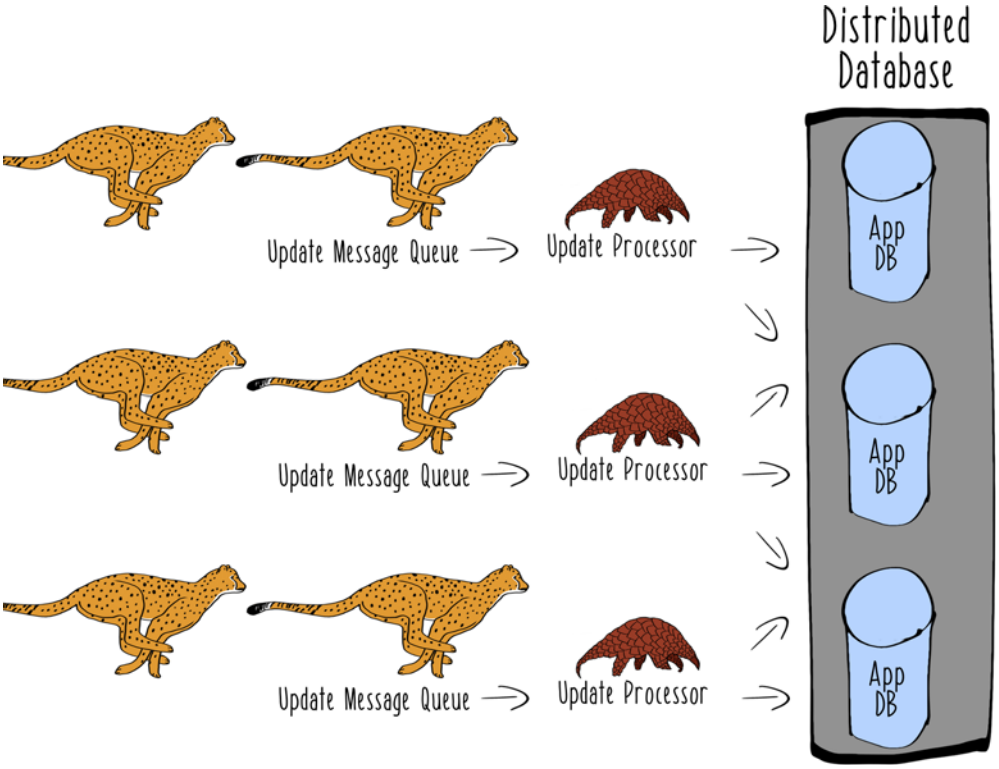


Figure 3.9 A distributed database

many databases. Each node in your database cluster has the same responsibilities and capabilities, so the architecture of the final system is very close to the ideal shown in the original distributed-database diagram, shown in figure 3.9.

There's no single throughput bottleneck for reads or writes. The fact-based data model ensures that multiple operations aren't contending over a shared mutable value, so your ability to scale out is limited only by your infrastructure budget.

This architecture also has important fault-tolerance capabilities. Should the network partition your nodes in some way, your database will remain usable.

Tolerance to network partitions is a difficult property for a database design to support. Strategies for greater partition tolerance have been some of the most important innovations in database development. Moreover, your data model works in tandem with the capabilities provided by the database infrastructure. In the event of a network partition, the fact data that the database returns will be an accurate, if possibly incomplete, view of the data collected. The potential for an incomplete view of the data is unavoidable in a distributed database with partition tolerance. When the network prevents communication across the cluster, some portion of your data is unavailable; no database can change that. Instead of degrading responsiveness, databases like Couchbase choose to return the data that is available. That's not a problem for your application. Consider the sequence of sensor readings about the count of zebras believed to be at Location 55, shown in figure 3.10.

This series of readings relies on the data stored in all the nodes of the database cluster. Due to the uncertainties of the network, you could lose access to some of the nodes in your cluster (figure 3.11).

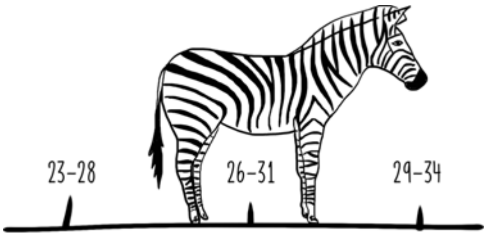


Figure 3.10 Complete zebra-sensor readings

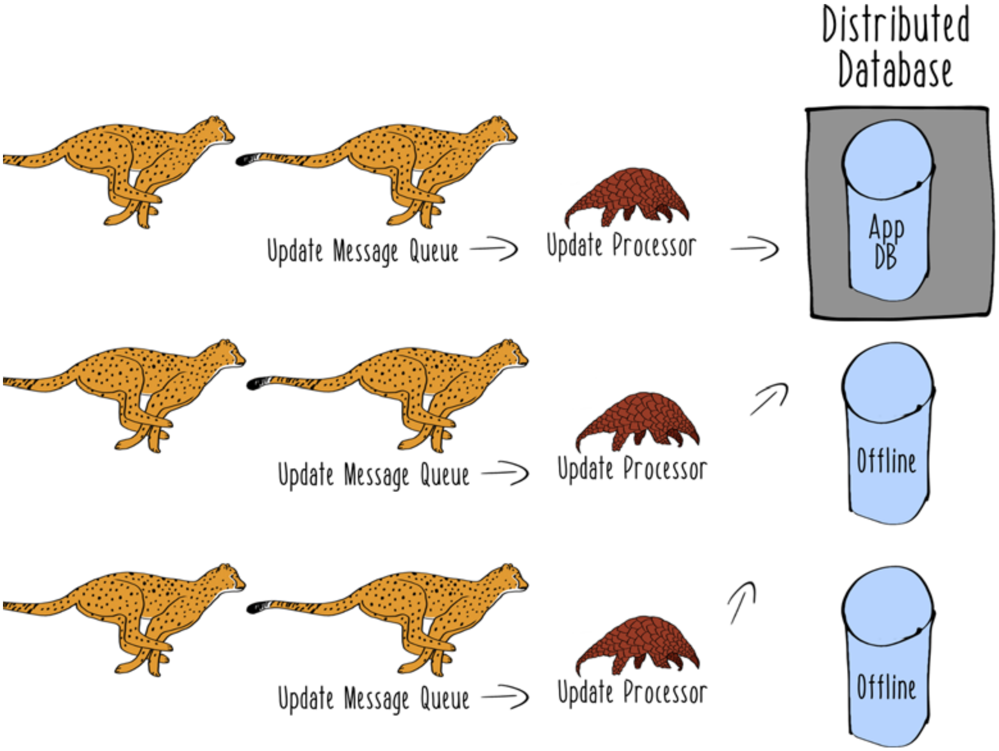


Figure 3.11 A network partition

The database client library used in the update-processor program instances will route to the still reachable nodes. But because your data is distributed throughout the cluster, it's possible that you might only have access to some of the data temporarily. For example, you may only have the readings shown in figure 3.12.

For many applications, this partial view of the database would be entirely usable. After all, the facts recorded don't reflect a current certainty that a definite number of zebras are present at Location 55. You could even use a linear model to interpolate

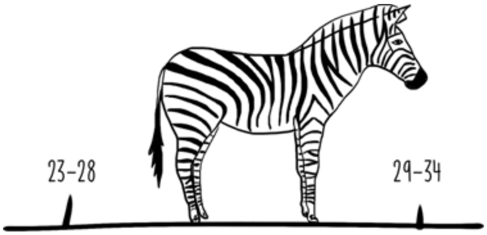


Figure 3.12 Incomplete zebra sensor readings

the missing value if you wanted to. The trend is the same, and the overall picture provided by a query over readings would present a similar view. The actions you would choose to take upon seeing this incomplete data are likely the same that you would take upon seeing the full dataset: you would send out some lionesses to go get you some lunch with stripes on it. By using a fact-based data model with explicit uncertainty, you've already built the ability to handle incomplete views of data into your usage of the database.

There are even more properties of this system that are worth highlighting. A database's internal *concurrency models* are its plan for how to coordinate multiple users. Couchbase uses a concurrency model called *multiversion concurrency control* (MVCC). Systems that use MVCC don't internally lock records for mutation but instead replicate data.

As you've seen, locking gets in the way of scaling out write operations. A Couchbase cluster doesn't have the problems of trying to hold locks across nodes, because within the cluster, actions are coordinated via message passing.

This message passing occurs within an actor system using a supervisory hierarchy. As you've seen before and we'll discuss more in future chapters, using supervisory hierarchies is an excellent strategy for ensuring resilience in the face of failure.

Erlang and OTP

The actor model implementation in Couchbase isn't Akka. Rather, it's built in a language called Erlang using the OTP libraries. Erlang and OTP (the Open Telecom Platform) were originally built for telecommunications applications. More recently, Erlang and OTP have become popular within the implementation of distributed databases like Couchbase and Riak. But the actor-model implementations in both systems are closely related (for example, frequent use of pattern matching). In fact, Jonas Bonér created Akka based on his experiences with Erlang and OTP.

All these characteristics of the database's capabilities align with your larger goals in building a reactive machine learning system. This alignment of vision has a real impact on the performance you can expect from the database and the guarantees it provides. Because your data model no longer relies on shared mutable state that must be maintained with locks, you can take advantage of the extreme scalability capabilities of a database oriented around massive concurrency.

Finally, the main reason why this data architecture works so well is that it's based on ideas that work well in reality, not just in software:

- *Facts are always true*. Future facts don't erase old facts. If there were gazelles down by the river this morning, that then that statement is always true, even after the gazelles have moved on. You don't need to forget the gazelles from the morning just because you no longer go down there and eat them.
- *Central control leads to contention*. Even though you're the Lion Queen, you can't make the impossible possible. A pangolin can only move so fast, and cheetahs don't have the patience for data entry. They can't all coordinate through a single point of control and still be all over the savannah at the same time. It's the same reason you don't have all your lionesses come back to you to ask if they should take down a slow-moving wildebeest. If everything had to go through a single point of control, whether that point of control is you or any other lion, nothing would ever get done.
- *You can't know anything for certain*. The savannah is a sprawling, chaotic, distributed system, and the view from Lion Rock only stretches so far. Maybe the cheetahs didn't really eat Patrick Pangolin. Who's to say? All you or any other lion can do is record all the uncertain facts that come your way and do your best with the knowledge that you have. Being a responsible queen means acknowledging the limits of what you know, because the real world of Africa is uncertain. Acknowledging that you're not omniscient is part of making good decisions on behalf of the pride.

The principled but pragmatic approach you've taken to building out your data-collection capabilities has since paid off. You've been able to feed your pride more consistently and use your time more effectively. Predators across Africa are envious of your all-encompassing view of the savannah. The approaches you've used in this chapter have also put you in an excellent position to build new capabilities in the future. Next up, you're considering building a feature-extraction system to power machine learning predictions. The work of a Lion Queen is never done, but you've done well for your pride. You should be proud.

3.4 Applications

You may not live in the Serengeti. You may not operate a distributed sensor network. You may not even be a lion. Can you still apply the techniques used in this chapter? Of course you can.

All sorts of systems produce massive amounts of data that must be collected to be put to use. Cell phones are full of various sensors, many of which send data back to remote servers. For example, location data is collected by phones and used to guide location-specific recommendations. Even without the use of sensor hardware, most mobile apps send all sorts of data back to remote servers about interactions with the app: swipes, notification dismissals, and so on. This data can be used to determine things like application usage patterns and relate groups of users (for example, subway commuters).

And it's not just phones and wildlife. A huge range of traditional products now produce data that can be collected and used in machine learning systems. We live in a world where robot vacuums and electric cars are real and even common. The amount of data we're producing as a species has exploded over the past decades. As you've seen in this chapter, working with data at scale is different. The techniques demonstrated in this chapter offer real solutions to the problems of working with data at scale. If you're working on an existing system that uses a different approach to handling data, it may not be clear that all the techniques used in this chapter are necessary. You may be working with data that isn't as big, fast, or hairy as the data from a sensor network tracking the Great Migration. In that case, you may not need to use all the techniques shown in this chapter.

But even something as innocuous as a watch can now produce an effectively infinite amount of data of arbitrary complexity. Once you start looking for applications to apply these techniques, it's not hard to find them. Any system with a concept of a user, such as an e-commerce site, could use a fact-based data model to record what users do. Even a modest-velocity business application recording customer transactions could potentially benefit from removing contention over locked data items. The tools in this chapter work incredibly well at scale, but they don't *require* your application to be at scale. Now that you understand the reactive approach to data collection, I expect that you'll find lots of places to put it to use.

3.5 Reactivities

It's time to go off on your own and explore the world of reactive data collection. This section's "reactivities" are intended to point you in the direction of venturing even further beyond the plains of the Serengeti. There are no right answers for these questions; there are merely engaging problems that you can explore even more than I've had the room to discuss in the space of a book:

- *Implement your own reactive data-collection system.* Really. Start with the basics of recording facts. You can use any database you want. Pick whatever database you're most familiar with. Then think about the consequences of that choice:
  - What properties of the database will support making your system more responsive, resilient, elastic, and message-driven? In particular, think through how the database is going to handle massive increases in load.
  - What's going to slow down and why?
  - Think through the data model you've used for your facts—does it include uncertainty?
  - If it didn't before, how hard would it be to add an explicitly uncertain data model?
  - What sort of queries can you write against your database?
  - If you deleted a bunch of records, how would the results of those queries change?

- *Explore other open source data-collection systems*. They don't have to be explicitly reactive. If you want to focus on the design pattern shown in this chapter, it often goes by the name of *event sourcing*. There are even databases specifically built around supporting event sourcing. This idea of collecting lots of fact data is commonly used in monitoring applications. Can you find any monitoring applications that persist all the events in the systems being monitored as databases of immutable facts? Try answering some of the same questions about these other implementations of data-collection systems that you asked of your own implementation:
  - How reactive is this system?
  - If something failed, what would happen?
  - If the system were overwhelmed with traffic, would I still be able to query data?
  - Would I ever want or need to rewrite a "fact" after it was written?
  - What would happen if I kept pushing data into this system forever?

NOTE You got familiar with Couchbase in this chapter as a way of learning about databases in general. You won't need more specific knowledge of this database for the rest of the book, but if you're interested in the technology, there are several books on it.

Summary

- Facts are immutable records of something that happened and the time that it happened:
  - Transforming facts during data collection results in information loss and should never be done.
  - Facts should encode any uncertainty about that information.
- Data collection can't work at scale with shared mutable state and locks.
- Fact databases solve the problems of collecting data at scale:
  - Facts can always be written without blocking or using locks.
  - Facts can be written in any order.
- Futures-based programming handles the possibility that operations can take time and even fail.

In the next chapter, we'll put all this raw fact data to use by deriving semantically meaningful features from it. These features will be the first step in our process of extracting insight from data. We'll be using Spark again to show how to build feature-extraction pipelines that are both elegant and massively scalable.

4 Generating features

This chapter covers

- Extracting features from raw data
- Transforming features to make them more useful
- Selecting among the features you've created
- How to organize feature-generation code

This chapter is the next step on our journey through the components, or phases, of a machine learning system, shown in figure 4.1. The chapter focuses on turning raw data into useful representations called *features*. The process of building systems that can generate features from data, sometimes called *feature engineering*, can be deceptively complex. Often, people begin with an intuitive understanding of *what* they want the features used in a system to be, with few plans for *how* those features will be produced. Without a solid plan, the process of feature engineering can easily get off track, as you saw in the Sniffable example from chapter 1.

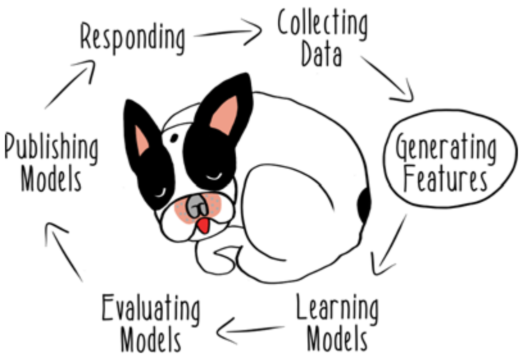


Figure 4.1 Phases of machine learning

In this chapter, I'll guide you through the three main types of operations in a feature pipeline: extraction, transformation, and selection. Not all systems do all the types of operations shown in this chapter, but all feature engineering techniques can be thought of as falling into one of these three buckets. I'll use type signatures to assign techniques to groups and give our exploration some structure, as shown in table 4.1.

Table 4.1 Phases of feature generation

Phase	Input	Output
Extract	`RawData`	`Feature`
Transform	`Feature`	`Feature`
Select	`Set[Feature]`	`Set[Feature]`

Real-world feature pipelines can have very complex structures. You'll use these groupings to help you understand how you can build a feature-generation pipeline in the best way possible. As we explore these three types of feature-processing operations, I'll introduce common techniques and design patterns that will keep your machine learning system from becoming a tangled, unmaintainable mess. Finally, we'll consider some general properties of data pipelines when discussing the next component of machine learning systems discussed in chapter 5, the model-learning pipeline.

Type signatures

You may not be familiar with the use of *types* to guide how you think about and implement programs. This technique is common in statically typed languages like Scala and Java. In Scala, functions are defined in terms of the inputs they take, the outputs they return, and the types of both. This is called a *type signature*. In this book, I mostly use a fairly simple form of signature notation that looks like this: `Grass => Milk`. You can read this as, "A function from an input of type Grass to an output of type Milk." This would be the type signature of some function that behaves much like a cow.

To cover this enormous scope of functionality, we need to rise above it all to gain some perspective on what features are all about. To that end, we'll join the team of Pidg'n, a microblogging social network for tree-dwelling animals, not too different from Twitter.

We'll look at how we can take the chaos of a short-form, text-based social network and build meaningful representations of that activity. Much like the forest itself, the world of features is diverse and rich, full of hidden complexity. We can, however, begin to peer through the leaves and capture insights about the lives of tree-dwelling animals using the power of reactive machine learning.

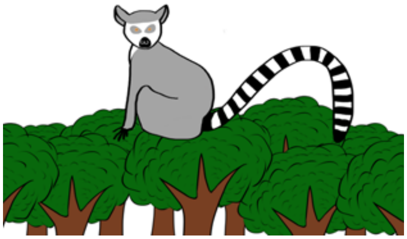
4.1 Spark ML

Before we get started building features, I want to introduce you to more Spark functionality. The `spark.ml` package, sometimes called Spark ML, defines some high-level APIs that can be used to create machine learning pipelines. This functionality can reduce the amount of machine learning–specific code that you need to implement yourself, but using it does involve a change in how you structure your data.

The Spark ML API uses mostly the same nomenclature for feature extraction, transformation, and selection that I use in this chapter, though there are subtle differences. If and when you read the Spark ML documentation, you may see something called a *transformation* operation, which I call an *extraction* operation. These are generally minor, unimportant differences that you can ignore. Different technologies name and structure this functionality differently, and you'll see all sorts of different naming conventions in the machine learning literature. The type signature–based framework for dividing feature-generation functionality that I use in this chapter is just a tool to help you implement and organize your code. Once you've mastered the feature-generation concepts in this chapter, you'll be better equipped to see through differences in nomenclature to the similarities in functionality.

Much of the machine learning functionality in Spark is designed to be used with DataFrames, instead of the RDDs that you've seen up until this point. DataFrames are simply a higher-level API on top of RDDs that give you a richer set of operations. You can think of DataFrames as something like tables in relational databases. They have different columns, which you can define and then query. Much of the recent progress in performance and functionality within Spark has been focused on DataFrames, so to get access to the full power of things like MLlib's machine learning capabilities, you'll need to use DataFrames for some operations. The good news is that they're very similar to the RDDs you've been working with and tabular data structures you may have used in other languages, such as pandas DataFrames in Python or R's data frames.

4.2 Extracting features



Now that I've introduced some of the tools, let's begin to solve the problem. We'll start our exploration of the feature engineering process at the very beginning, with raw data. In this chapter, you'll take on the role of Lemmy, an engineer on the Pidg'n data team.

Your team knows it wants to build all sorts of predictive models about user activity. You're just getting started, though, and all you have are the basics of application data: squawks (text posts of 140 characters or less), user profiles, and the follower relationships. This is a rich dataset, for sure, but you've never put it to much analytical use. To start with, you've decided to focus on the problem of predicting which new users will become *Super Squawkers*, users with more than a million followers.

To start this project, you'll extract some features to use in the rest of your machine learning system. I define the process of feature extraction as taking in raw data of some sort and returning a feature. Using Scala type signatures, feature extraction can be represented like this: `RawData => Feature`. That type signature can be read as, "A function that takes raw data and returns a feature." If you define a function that satisfies that type signature, it might look something like the stub in the following listing.

Listing 4.1 Extracting features

```
def extract(rawData: RawData): Feature = ???
```

Put differently, any output produced from raw data is a potential feature, regardless of whether it ever gets used to learn a model.

The Pidg'n data team has been collecting data since day one of the app as part of keeping the network running. You have the complete, unaltered record of all the actions ever taken by Pidg'n users, much like the data model discussed in chapter 3. Your team has built a few aggregates of that data for basic analytical purposes. Now you want to take that system to the next level by generating semantically meaningful derived representations of that raw data—features. Once you have features of any kind, you can begin learning models to predict user behavior. In particular, you're interested in seeing if you can understand what makes particular squawks and squawkers more popular than others. If a squawker has the potential to become very popular, you want to provide them with a more streamlined experience, free of advertisements, to encourage them to squawk more.

Let's begin by extracting features from the raw data of the text of squawks. You can start by defining a simple case class and extracting a single feature for a few squawks. Listing 4.2 shows how to extract a feature consisting of the words in the text of a given squawk. This implementation will use Spark's `Tokenizer` to break sentences into words. Tokenization is just one of several common text-processing utilities that come built into Spark that make writing code like this fast and easy. For advanced use cases, you may want to use a more sophisticated text-parsing library, but having common utilities easily available can be very helpful.

Listing 4.2 Extracting word features from squawks

```
case class Squawk(id: Int, text: String)

val squawks = session.createDataFrame(Seq(
  Squawk(123, "Clouds sure make it hard to look
➥ on the bright side of things."),
  Squawk(124, "Who really cares who gets the worm?
➥ I'm fine with sleeping in."),
  Squawk(125, "Why don't french fries grow on trees?")))
➥ .toDF("squawkId", "squawk")
```

Case class to hold a basic data model of a squawk

Creates a DataFrame from a sequence

Instantiates example instances of squawks

Names columns to place values in a DataFrame

```
val tokenizer = new Tokenizer().setInputCol("squawk")
➥ .setOutputCol("words")

val tokenized = tokenizer.transform(squawks)

tokenized.select("words", "squawkId").show()
```

Sets up a Tokenizer to split the text of squawks into words and put them in an output column

Executes the Tokenizer and populates the words column in a DataFrame

Prints the results for inspection

The operations in listing 4.2 give you a DataFrame that contains a column called words, which has all the words in the text of the squawk. You could call the values in the words column a *feature*. These values could be used to learn a model. But let's make the semantics of the pipeline clearer using the Scala type system.

Using the code in listing 4.3, you can define what a feature is and what specific sort of feature you've produced. Then, you can take the words column from that DataFrame and use it to instantiate instances of those feature classes. It's the same words that the Tokenizer produced for you, but now you have richer representations that you can use to help build up a feature-generation pipeline.

Listing 4.3 Extracting word features from squawks

```
trait FeatureType {
  val name: String
  type V
}

trait Feature extends FeatureType {
  val value: V
}

case class WordSequenceFeature(name: String, value: Seq[String])
➥ extends Feature {
  type V = Seq[String]
}

val wordsFeatures = tokenized.select("words")
  .map(row =>
  WordSequenceFeature("words",
    row.getSeq[String](0)))

wordsFeatures.show()
```

Defines a base trait for all types of features

Requires feature types to have names

Type parameter to hold the type of values generated by feature

Defines a base trait for all features as an extension of feature types

Requires that features have values of the type specified in the feature type

Defines a case class for features consisting of word sequences

Specifies that the type of features being generated is a sequence of strings (words)

Maps over rows and applies a function to each

Selects a words column from the DataFrame

Creates an instance of WordSequenceFeature named words

Gets extracted words out of a row

Prints features for inspection

With this small bit of extra code, you can define your features in a way that's more explicit and less tied to the specifics of the raw data in the original DataFrame. The resulting value is an RDD of WordSequenceFeature. You'll see later how you can continue to use this `Feature` trait with specific case classes defining the different types of features in your pipeline.

Immutable Facts

Also note that, when operating over the `DataFrame`, you can use a pure, anonymous, higher-order function to create instances of your features. The concepts of purity, anonymous functions, and higher-order functions may have sounded quite abstract when I introduced them in chapter 1. But now that you've seen them put to use in several places, I hope it's clear that they can be very simple to write. Now that you've gotten some Scala and Spark programming under your belt, I hope you're finding it straightforward to think of data transformations like feature extraction in terms of pure functions with no side effects.

You and the rest of the Pidg'n data team could now use these features in the next phase of the machine learning pipeline—model learning—but they probably wouldn't be good enough to learn a model of Super Squawkers. These initial word features are just the beginning. You can encode far more of your understanding of what makes a squawker super into the features themselves.

To be clear, there are sophisticated model-learning algorithms, such as neural networks, that require very little feature engineering on the data that they consume. You *could* use the values you've just produced as features in a model-learning process. But many machine learning systems will require you to do far more with your features before using them in model learning if you want acceptable predictive performance. Different model-learning algorithms have different strengths and weaknesses, as we'll explore in chapter 5, but all of them will benefit from having base features transformed in ways that make the process of model learning simpler. We need to move on to see how to make features out of other features.

4.3 Transforming features

Now that you've extracted some basic features, let's figure out how to make them useful. This process of taking a feature and producing a new feature from it is called *feature transformation*. In this section, I'll introduce you to some common transform functions and discuss how they can be structured. Then I'll show you a very important class of feature transformations: transforming features into concept labels.

What is feature transformation? In the form of a type signature, feature transformation can be expressed as `Feature => Feature`, a function that takes a feature and returns a feature. A stub implementation of a transformation function (sometimes called a *transform*) is shown in the next listing.

Listing 4.4 Transforming features

```
def transform(feature: Feature): Feature = ???
```

In the case of the Pidg'n data team, you've decided to build on your previous feature-engineering work by creating a feature consisting of the frequencies of given words in a squawk. This quantity is sometimes called a *term frequency*. Spark has built-in functionality that makes calculating this value easy.

Listing 4.5 Transforming words to term frequencies

```
val hashingTF = new HashingTF()
  .setInputCol("words")
  .setOutputCol("termFrequencies")

val tfs = hashingTF.transform(tokenized)

tfs.select("termFrequencies").show()
```

Instantiates an instance of a class to calculate term frequencies

Defines an input column to read from when consuming DataFrames

Defines an output to put term frequencies in

Executes the transformation

Prints term frequencies for inspection

It's worth noting that the hashingTF implementation of term frequencies was implemented to consume the DataFrame you previously produced, not the features you designed later. Spark ML's concept of a pipeline is focused on connecting operations on DataFrames, so it can't consume the features you produced before without more conversion code.

Infinite Data

Feature hashing

The use of the term *hashing* in the Spark library refers to the technique of *feature hashing*. Although it's not always used in feature-generation pipelines, feature hashing can be a critically important technique for building large numbers of features. In text-based features like term frequencies, there's no way of knowing a priori what all the possible features could be. Squawkers can write anything they want in a squawk on Pidg'n. Even an English-language dictionary wouldn't contain all the slang terms squawkers might use. Free-text input means that the universe of possible terms is effectively infinite.

One solution is to define a hash range of the size of the total number of distinct features you want to use in your model. Then you can apply a deterministic hashing function to each input to produce a distinct value within the hash range, giving you a unique identifier for each feature. For example, suppose `hash("trees")` returns `65381`. That value will be passed to the model-learning function as the identifier of the feature. This might not seem much more useful than just using `"trees"` as the identifier, but it is. When I discuss prediction services in chapter 7, I'll talk about why you'll want to be able to identify features that the system has possibly never seen before.

Let's take a look at how Spark ML's DataFrame-focused API is intended to be used in connecting operations like this. You won't be able to take full advantage of Spark ML until chapter 5, where you'll start learning models, but it's still useful for feature generation. Some of the preceding code can be reimplemented using a Pipeline from Spark ML. That will allow you to set the tokenizer and the term frequency operations as stages within a pipeline.

Listing 4.6 Using Spark ML pipelines

```
val pipeline = new Pipeline()
  .setStages(Array(tokenizer, hashingTF))

val pipelineHashed = pipeline.fit(squawksDF)

println(pipelineHashed.getClass)
```

Instantiates a new pipeline

Sets the two stages of this pipeline

Executes the pipeline

Prints the type of the result of the pipeline, a PipelineModel

This `Pipeline` doesn't result in a set of features, or even a `DataFrame`. Instead, it returns a `PipelineModel`, which in this case won't be able to do anything useful, because you haven't learned a model yet. We'll revisit this code in chapter 5, where we can go all the way from feature generation through model learning. The main thing to note about this code at this point is that you can encode a pipeline as a clear abstraction within your application. A large fraction of machine learning work involves working with pipeline-like operations. With the Spark ML approach to pipelines, you can be very explicit about how your pipeline is composed by setting the stages of the pipeline in order.

4.3.1 Common feature transforms

Sometimes you don't have library implementations of the feature transform that you need. A given feature transform might have semantics that are specific to your application, so you'll often need to implement feature transforms yourself.

Consider how you could build a feature to indicate that a given Pidg'n user was a Super Squawker (user with more than a million followers). The feature-extraction process will give you the raw data about the number of followers a given squawker has. If you used the number of followers as a feature, that would be called a *numerical* feature. That number would be an accurate snapshot of the data from the follower graph, but it wouldn't necessarily be easy for all model-learning algorithms to use. Because your intention is to express the idea of a Super Squawker, you could use a far simpler representation: a Boolean value representing whether or not the squawker has more than a million followers.

The squirrel, a rather ordinary user, has very few followers. But the sloth is an terrific Super Squawker. To produce meaningful features about the differences between these two squawkers, you'll follow the same process of going from raw data, to numeric features, and then to Boolean features. This series of data transformations is shown for the two users in figure 4.2.

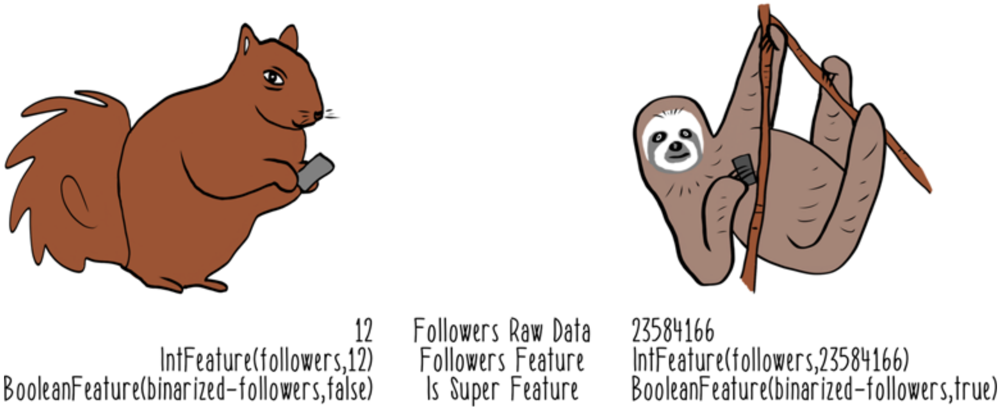

12
IntFeature(followers,12)
BooleanFeature(binarized-followers,false)
Followers Raw Data
Followers Feature
Is Super Feature
23584166
IntFeature(followers,23584166)
BooleanFeature(binarized-followers,true)

Figure 4.2 Feature transformations

The following listing shows how to implement this approach to binarization to produce a Super Squawker feature.

Listing 4.7 Binarizing a numerical feature

```
case class IntFeature(name: String, value: Int) extends Feature {
  type V = Int
}

case class BooleanFeature(name: String, value: Boolean) extends Feature {
  type V = Boolean
}

def binarize(feature: IntFeature, threshold: Double): BooleanFeature = {
  BooleanFeature("binarized-" + feature.name, feature.value > threshold)
}

val SUPER_THRESHOLD = 1000000

val squirrelFollowers = 12
val slothFollowers = 23584166

val squirrelFollowersFeature = IntFeature("followers", squirrelFollowers)
val slothFollowersFeature = IntFeature("followers", slothFollowers)

val squirrelIsSuper = binarize(squirrelFollowers, SUPER_THRESHOLD)
val slothIsSuper = binarize(slothFollowers, SUPER_THRESHOLD)
```

Specifies that these are integer features

Case class representing a numerical feature where the value is an integer

Case class representing a Boolean feature

Specifies that these are Boolean features

Function that takes a numeric integer feature and threshold and returns a Boolean feature

Adds the name of the transform function to the resulting feature name

Constant defining the cutoff for a squawker to be super

Raw numbers of followers for the squirrel and the sloth

Numeric integer feature representing the number of followers

Boolean feature indicating the squirrel is not a Super Squawker

Boolean feature indicating the sloth is a Super Squawker

Pure Functions

The binarize function is a good example of a reusable transform function. It also ensures the resulting feature is somewhat self-describing by appending the name of the transform function to the resulting feature. Ensuring that we can identify the operations that were applied to produce a feature is an idea we'll revisit in later chapters. Finally, note that the transformation function binarize is a pure function.

Using only pure functions in feature transforms is an important part of establishing a coherent structure for feature-generation code. Separating feature extraction and feature transformation within a code base can be difficult, and the boundaries between the two can be hard to draw. Ideally, any I/O or side-effecting operations should be contained in the feature-extraction phase of the pipeline, with all transformations' functionality being implemented as pure functions. As you'll see later, pure transforms are simple to scale and easy to reuse across features and feature-extraction contexts (model learning and predicting).

There's a huge range of commonly used transformation functions. Similar to binarization, some approaches reduce continuous values to discrete labels. For example, a feature designed to express the time of day when a squawk was posted might not use the full timestamp. Instead, a more useful representation could be to transform all times into a limited set of labels, as shown in table 4.2.

Table 4.2 Transforming times into time labels

Time	Label
7:02	Morning
12:53	Midday
19:12	Night

The implementation of a transform to do this is trivial and is naturally a pure function.

There's another variation on reducing continuous data to labels, called *binning*, in which the source feature is reduced to some arbitrary label defined by the range of values that it falls into. For example, you could take the number of squawks a given user has made and reduce it to one of three labels indicating how active the squawker is, as shown in table 4.3.

Table 4.3 Binning

Squawks	Label	Activity level
7	0_99	Least active squawkers
1,204	1000_99999	Moderately active squawkers
2,344,910	1000000_UP	Most active squawkers

Again, an implementation of such a transform would be trivial and naturally a pure function. Transforms *should* be easy to write and should correspond closely to their formulation in mathematical notation. When it comes to implementing transforms, you should always abide by the KISS principle: Keep It Simple, Sparrow. Reactive machine learning systems are hard enough to implement without implementing complicated transforms. Usually, an overly long transform implementation is a smell that someone has laid a rotten egg. In a few special cases, you may want to implement something like a transformer with more involved semantics. We'll consider such circumstances later in this chapter and later in the book.

4.3.2 Transforming concepts

Before we leave the topic of transformations, we need to consider one very common and critical class of feature transformations: the ones that produce concepts. As mentioned in chapter 1, *concepts* are the things that a machine learning model is trying to predict. Although some machine learning algorithms can learn models of continuous concepts, such as the number of squawks a given user will write over the course of the next month, many machine learning systems are built to perform classification. In *classification* problems, the learning algorithm is trying to learn a discrete number of class labels, not continuous values. In such systems, the concept has to be produced from the raw data, during feature extraction, and then reduced to a class label via transformation. Concept class labels aren't exactly the same thing as features, but often the difference is just a matter of how we use the piece of data. Typically, and ideally, the same code that might binarize a feature will also binarize a concept.

Building on the code in listing 4.7, in the next listing, take the Boolean feature about Super Squawkers and produce a Boolean concept label that classifies squawkers into super or not.

Listing 4.8 Creating concept labels from features

```
trait Label extends Feature
```

Defines labels as subtypes of features

```
case class BooleanLabel(name: String, value: Boolean) extends Label {
  type V = Boolean
}
```

Creates a case class for Boolean labels

```
def toBooleanLabel(feature: BooleanFeature) = {
  BooleanLabel(feature.name, feature.value)
}
```

Defines a simple conversion function from Boolean features to Boolean labels

```
val squirrelLabel = toBooleanLabel(squirrelIsSuper)
val slothLabel = toBooleanLabel(slothIsSuper)
```

Converts Super Squawker feature values into concept labels

```
Seq(squirrelLabel, slothLabel).foreach(println)
```

Prints label values for inspection

In this code, you've defined concept labels as a special subtype of features. That's not how features and labels are generally discussed, but it can be a helpful convention for code reuse in machine learning systems. Whether you intend to do so or not, any given feature value could be used as a concept label if it represents the concept class to be learned. The `Label` trait in listing 4.8 doesn't change the underlying structure of the data in a feature, but it does allow you to annotate when you're using a feature as a concept label. The rest of the code is quite simple, and you arrive at the same conclusion again: people just aren't that interested in what squirrels have to say.

4.4 Selecting features

Again, you find yourself in the same situation: if you've done all the work so far, you might now be finished. You could use the features you've already produced to learn a model. But sometimes it's worthwhile to perform additional processing on features before beginning to learn a model. In the previous two phases of the feature-generation process, you produced all the features you *might* want to use to learn a model, sometimes called a *feature set*. Now that you have that feature set, you could consider throwing some of those features in the trash.

The process of choosing from a feature set which features to use is known as *feature selection*. In type-signature form, it can be expressed `Set[Feature] => Set[Feature]`, a function that takes a set of features and returns another set of features. The next listing shows a stub implementation of a feature selector.

Listing 4.9 A feature selector

```
def select(featureSet: Set[Feature]): Set[Feature] = ???
```

Infinite Data

Why would you ever want to discard features? Aren't they useful and valuable? In theory, a robust machine learning algorithm could take as input feature vectors containing arbitrary numbers of features and learn a model of the given concept. In reality, providing a machine learning algorithm with too many features is just going to make it take longer to learn a model and potentially degrade that model's performance. You can find yourself needing to choose among features quite easily. By varying the parameters used in the transformation process, you could create an infinite number of features with a very small amount of code.

Using a modern distributed data-processing framework like Spark makes handling arbitrarily sized datasets easy. It's definitely to your benefit to consider a huge range of features during the feature extraction and transformation phases of your pipeline. And once you've produced all the features in your feature set, you can use some of the facilities in Spark to cut that feature set down to just those features that your model-learning algorithm will use to learn the model. There are implementations of feature-selection functionality in other machine learning libraries; Spark's MLlib is one of many options and certainly not the oldest one. For some cases, the feature-selection functionality provided by MLlib might not be sufficient, but the principles of feature selection are the same whether you use a library implementation or something more bespoke. If you end up writing your own version of feature selection, it will still be conceptually similar to MLlib's implementations.

Using the Spark functionality will again require you to leave behind your feature-case classes and the guarantees of static typing to use the machine learning functionality implemented around the high-level DataFrame API. To begin, you'll need to construct a `DataFrame` of training instances. These instances will consist of three parts: an arbitrary identifier, a feature vector, and a concept label. The following listing shows how to build up this collection of instances. Instead of using real features, you'll use some synthetic data, which you can imagine being about various properties of Squawkers.

Listing 4.10 A `DataFrame` of instances

```
val instances = Seq(
  (123, Vectors.dense(0.2, 0.3, 16.2, 1.1), 0.0),
  (456, Vectors.dense(0.1, 1.3, 11.3, 1.2), 1.0),
  (789, Vectors.dense(1.2, 0.8, 14.5, 0.5), 0.0)
)

val featuresName = "features"
val labelName = "isSuper"

val instancesDF = session.createDataFrame(instances)
  .toDF("id", featuresName, labelName)
```

Defines a collection of instances

Hardcodes some synthetic feature and concept label data

Names for features and label columns

Creates a DataFrame from the instances collection

Sets the name of each column in the DataFrame

Once you have a `DataFrame` of instances, you can take advantage of the feature-selection functionality built into MLlib. You can apply a chi-squared statistical test to rank the impact of each feature on the concept label. This is sometimes called *feature importance*. After the features are ranked by this criterion, the less impactful features can be discarded prior to model learning. The next listing shows how you can select the two most important features from your feature vectors.

Listing 4.11 Chi-squared-based feature selection

```
val selector = new ChiSqSelector()
  .setNumTopFeatures(2)
  .setFeaturesCol(featuresName)
  .setLabelCol(labelName)
  .setOutputCol("selectedFeatures")

val selectedFeatures = selector.fit(instancesDF)
  .transform(instancesDF)

selectedFeatures.show()
```

Creates a new feature selector

Sets the number of features to retain to 2

Sets the column where features are

Sets the column where concept labels are

Sets the column to place results, the selected features

Fits a chi-squared model to the data

Selects the most important features and returns a new DataFrame

Prints the resulting DataFrame for inspection

As you can see, having standard feature-selection functionality available at a library call makes feature selection pretty convenient. If you had to implement chi-squared-based feature selection yourself, you'd find that the implementation was a lot longer than the code you just wrote.

4.5 Structuring feature code

In this chapter, you've written example implementations of all the most common components of a feature-generation pipeline. As you've seen, some of these components are simple and easy to build, and you could probably see yourself building quite a few of them without any difficulty. If you've Kept It Simple, Sparrow, you shouldn't be intimidated by the prospect of producing lots of feature extraction, transformation, and selection functionality in your system. Or should you?

Within a machine learning system, feature-generation code can often wind up being the largest part of the codebase by some measures. A typical Scala implementation might have a class for each extraction and transformation operation, and that can quickly become unwieldy as the number of classes grows. To prevent feature-generation code from becoming a confusing grab bag of various arbitrary operations, you need to start putting more of your understanding of the semantics of feature generation into the structure of your implementation of feature-generation functionality. The next section introduces one such strategy for structuring your feature-generation code.

4.5.1 Feature generators

At the most basic level, you need to define an implementation of what is a unit of feature-generation functionality. Let's call this a *feature generator*. A feature generator can encompass either extraction or both extraction and transformation operations. The implementation of the extraction and transformation operations may not be very different from what you've seen before, but these operations will all be encapsulated in an independently executable unit of code that produces a feature. Your feature generators will be things that can take raw data and produce features that you want to use to learn a model.

Let's implement your feature generators using a trait. In Scala, *traits* are used to define behaviors in the form of a type. A typical trait will include the signatures and possibly implementations of methods that define the common behavior to the trait. Scala traits are very similar to interfaces in Java, C++, and C# but are much easier and more flexible to use than interfaces in any of those languages.

For the purpose of this section, let's say that your raw data, from the perspective of your feature-generation system, consists of squawks. Feature generation will be the process of going from squawks to features. The corresponding feature-generator trait can be defined.

Listing 4.12 A feature-generator trait

```
trait Generator {

  def generate(squawk: Squawk): Feature

}
```

The Generator trait defines a feature generator to be an object that implements a method, generate, that takes a squawk and returns a feature. This is a concrete way of defining the behavior of feature generation. A given implementation of feature generation might need all sorts of other functionality, but this is the part that will be common across all implementations of feature generation. Let's look at one implementation of this trait.

Your team is interested in understanding how squawk length affects squawk popularity. There's an intuition that even 140 characters is too much to read for some squawkers, such as hummingbirds. They just get bored too quickly. Conversely, vultures have been known to stare at the same squawk for hours on end, so long posts are rarely a problem for them. For you to be able to build a recommendation model that will surface relevant content to these disparate audiences, you'll need to encode some of the data around squawk length as a feature. This can easily be implemented using the Generator trait.

As discussed before, the idea of length can be captured using the technique of binning to reduce your numeric data to categories. There's not much difference between a 72-character squawk and a 73-character squawk; you're just trying to capture the approximate size of a squawk. You'll divide squawks into three categories based on length: short, moderate, and long. You'll define your thresholds between the categories to be at the thirds of the total possible length. Implemented according to your Generator trait, you get something like the following listing.

Listing 4.13 A categorical feature generator

```
object SquawkLengthCategory extends Generator {

  val ModerateSquawkThreshold = 47
  val LongSquawkThreshold = 94

  private def extract(squawk: Squawk): IntFeature = {
    IntFeature("squawkLength", squawk.text.length)
  }

  private def transform(lengthFeature: IntFeature): IntFeature = {
```

Defines a generator as an object that extends the Generator trait

Constant thresholds to compare against

Extracting: uses the length of the squawk to instantiate an IntFeature

Transforming: takes the IntFeature of length, returns the IntFeature of category

```
    val squawkLengthCategory = lengthFeature match {
      case IntFeature(_, length) if length < ModerateSquawkThreshold => 1
      case IntFeature(_, length) if length < LongSquawkThreshold => 2
      case _ => 3
    }

    IntFeature("squawkLengthCategory", squawkLengthCategory)
  }

  def generate(squawk: Squawk): IntFeature = {
    transform(extract(squawk))
  }
}
```

Uses a pattern-matching structure to determine which category the squawk length falls into

Returns Int for a category (for ease of use in model learning)

Returns a category of 3, a long squawk, in all other cases

Returns a category as a new IntFeature

Generating: extracts a feature from the squawk and then transforms it to a categorical IntFeature

This generator is defined in terms of a singleton object. You don't need to use instances of a class, because all the generation operations are themselves pure functions.

Internal to your implementation of the feature generator, you still used a concept of extraction and transformation, even though you now only expose a `generate` method as the public API to this object. Though that may not always seem necessary, it can be helpful to define all extraction and transformation operations in a consistent manner using feature-based type signatures. This can make it easier to compose and reuse code.

Reuse of code is a huge issue in feature-generation functionality. In a given system, many feature generators will be performing operations very similar to each other.

A given transform might be used dozens of times if it's factored out and reusable. If you don't think about such concerns up front, you may find that your team has reimplemented some transform, like averaging five different times in subtly different ways across your feature-generation codebase. That can lead to tricky bugs and bloated code.

You don't want your feature-generation code to be messier than a tree full of marmosets! Let's take a closer look at the structure of your generator functionality. The `transform` function in listing 4.13 was doing something you might wind up doing a lot in your codebase: categorizing according to some threshold. Let's look at it again.

Listing 4.14 Categorization using pattern matching

```
private def transform(lengthFeature: IntFeature): IntFeature = {
  val squawkLengthCategory = lengthFeature match {
    case IntFeature(_, length) if length < ModerateSquawkThreshold => 1
    case IntFeature(_, length) if length < LongSquawkThreshold => 2
    case _ => 3
  }
```

You definitely shouldn't be implementing a comparison against thresholds more than once, so let's find a way to pull that code out and make it reusable. It's also weird that you had to define the class label integers yourself. Ideally, you'd just have to worry about your thresholds and nothing else.

Let's pull out the common parts of this code for reuse and make it more general in the process. The code in the next listing shows one way of doing this. It's a little dense, so we'll walk through it in detail.

Listing 4.15 Generalized categorization

```
object CategoricalTransforms {

  def categorize(thresholds: List[Int]): (Int) => Int = {
    (dataPoint: Int) => {
      thresholds.sorted
        .zipWithIndex
        .find {
        case (threshold, i) => dataPoint < threshold
      }.getOrElse((None, -1))
        ._2
    }
  }
}
```

Singleton object to hold a pure function

Only takes a list of thresholds as input

Returns an anonymous categorization function that takes Int as an argument

Ensures that a list of thresholds is sorted, because categorization relies on it

Zips up a list of thresholds and corresponding indices (used as category labels)

Finds an entry that satisfies the case clause predicate

Defines a passing case as being when a data point is less than the threshold

Gets a matching value out of an option or returns a sentinel value of –1 when matching fails

Takes a second element out of a tuple, which is the category label (in integer form)

This solution uses a few techniques that you may not have seen before. For one, this function's return type is `(Int) => Int`, a function that takes an integer and returns an integer. In this case, the function returned will categorize a given integer according to the thresholds previously provided.

The thresholds and categories are also zipped together so they can be operated on as a pair of related values (in the form of a tuple). *Zipping*, or *convolution* as it's sometimes called, is a powerful technique that's commonly used in Scala and other languages in the functional programming tradition. The name *zip* comes from the similarity to the action of a zipper. In this case, you're using a special sort of zip operation that conveniently provides you indices corresponding to the number the elements in the collection being zipped over. This approach to producing indices is far more elegant than C-style iteration using a mutable counter, which you may have seen in other languages, such as Java and C++.

After zipping over the values, you use another new function, `find`, with which you can define the element of a collection you're looking for in terms of a *predicate*. Predicates are Boolean functions that are either true or false, depending on their values. They're commonly used in mathematics, logic, and various forms of programming such as logic and functional programming. In this usage, the predicate gives you a clear syntax for defining what constitutes falling into a category bucket.

This code also deals with uncertainty in external usage in ways that you haven't before. Specifically, it sorts the categories, because they might not be provided in a sorted list, but your algorithm relies on operating on them in order. Also, the `find` function returns an `Option` because the `find` operation may or may not find a matching value. In this case, you use the value `-1` to indicate an unusable category, but how a categorization failure should be handled depends a lot on how the functionality will be integrated in the client generator code. When you factor out common feature transforms to shared functions like this, you should take into account the possibilities of future broad usage of the transform. By implementing it with these extra guarantees, you reduce the chances that someone will use your categorization functionality in the future and not get the results they wanted.

The code in listing 4.15 might be a bit harder to understand than the original implementation in listings 4.13 and 4.14. Your refactored version does more work to give you a more general and robust version of categorization. You may not expect every implementer of a feature generator to go through this much work for a simple transform, but because you've factored out this functionality to shared, reusable code, they don't have to. Any feature-generation functionality needing to categorize values according to a list of thresholds can now call this function. The transform from listings 4.13 and 4.14 can now be replaced with the very simple version in listing 4.16. You still have a relatively complex implementation of categorization in listing 4.15, but now, that complex implementation has been factored out to a separate component, which is more general and reusable. As you can see in the next listing, the callers of that functionality, like this transform function, can be quite simple.

Listing 4.16 Refactored categorization transform

```
import CategoricalTransforms.categorize

private def transform(lengthFeature: IntFeature): IntFeature = {
  val squawkLengthCategory = categorize(Thresholds)
  ➥ (lengthFeature.value)
  IntFeature("squawkLengthCategory", squawkLengthCategory)
}
```

Creates the categorization function and applies it to the value for categorization

Once you have dozens of categorical features, this sort of design strategy will make your life a lot easier. Categorization is now simple to plug in and easy to refactor should you decide to change how you want it implemented.

4.5.2 Feature set composition

You've seen how you can choose among the features you produced, but there's actually a zeroth step that's necessary in some machine learning systems. Before you even begin the process of feature generation, you may want to choose which feature generators should be executed. Different models need different features provided to them. Moreover, sometimes you need to apply specific overrides to your normal usage of data because of business rules, privacy concerns, or legal reasons.

In the case of Pidg'n, you have some unique challenges due to your global scale. Different regions have different regulatory regimes governing the use of their citizens' data. Recently, a new government has come to power in the rainforests of Panama.

The new minister of commerce, an implacable poison-dart frog, has announced new regulation restricting the use of social-media user data for non-rainforest purposes. After consultation with your lawyers, you decide that the new law means that features using data from rainforest users should only be used in the context of models to be applied on recommendations for residents of the rainforest.

Let's look at what impact this change might have on your codebase. To make things a bit more concise, let's define a simple trait to allow you to make simplified generators quickly. This will be a helper to allow you to skip over generator-implementation details that aren't relevant to feature-set composition. The next listing defines a stub feature generator that returns random integers.

Listing 4.17 A stub feature-generator trait

```
trait StubGenerator extends Generator {
  def generate(squawk: Squawk) = {
    IntFeature("dummyFeature", Random.nextInt())
  }
}
```

Implementation of the generate method for implementers of trait to use

Returns random integers

Using this simple helper trait, you can now explore some of the possible impacts that the rainforest data-usage rules might have on your feature-generation code. Let's say the code responsible for assembling your feature generators looks like the following listing.

Listing 4.18 Initial feature set composition

```
object SquawkLanguage extends StubGenerator {}

object HasImage extends StubGenerator {}

object UserData extends StubGenerator {}

val featureGenerators = Set(SquawkLanguage, HasImage, UserData)
```

Normal feature generator about the language the squawk was written in

Normal feature generator about whether the squawk contains an image

User-data feature generator that must be changed

Set of all the feature generators to execute to produce data

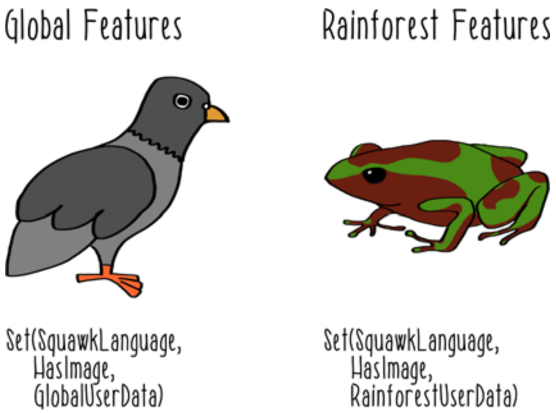


Figure 4.3 Multiple feature-generator sets

Now you need to restructure this code to have one feature set produced for your normal, global models and one feature set for your rainforest models, as shown in figure 4.3. The following listing shows an approach to defining these two different sets of feature generators.

Listing 4.19 Multiple feature sets

User-data feature generator that will only access non-rainforest data

User-data feature generator that will only access rainforest data

```
object GlobalUserData extends StubGenerator {}

object RainforestUserData extends StubGenerator {}

val globalFeatureGenerators = Set(SquawkLanguage, HasImage,
➥ GlobalUserData)

val rainforestFeatureGenerators = Set(SquawkLanguage, HasImage,
➥ RainforestUserData)
```

Set of features available to be used on global models

Set of features available to be used on rainforest models

You could stop with this implementation if you chose. As long as the rainforest feature generators are being used for rainforest models, you've done what the frog asked. But there are reasons to keep working on this problem. Machine learning systems are incredibly complicated to implement. Common feature-generation functionality can get reused in all sorts of places. The implementation in listing 4.19 is correct, but with Pidg'n's rapid growth, new engineers unfamiliar with this data-usage issue might refactor this code in such a way as to misuse rainforest feature data.

Let's see if you can make misusing this data even harder by defining a trait that allows you to mark code as having rainforest user data in it.

Listing 4.20 Ensuring correct usage of rainforest data

```
trait
  ➥ RainforestData {
  self =>
  require(rainforestContext(),
    s"${self.getClass} uses rainforest data outside of a
    ➥ rainforest context.")

  private def rainforestContext() = {
    val environment = Option(System.getenv("RAINFOREST"))
    environment.isDefined && environment.get.toBoolean
  }
}

object SafeRainforestUserData extends StubGenerator
➥ with RainforestData {}

val safeRainforestFeatureGenerators = Set(SquawkLanguage,
➥ HasImage, SafeRainforestUserData)
```

Defines a trait for the usage of rainforest data

Says all instances of this trait must execute the following code

Requires that rainforest environment validation passes

Prints a message explaining disallowed usage in the event of not being in the rainforest context

Validation method ensuring that the code is being called in the rainforest context

Retrieves the rainforest environment variable

Checks that the value exists and is true

Defines a feature generator for the rainforest user data

Assembles feature generators to use for the rainforest data

This code will throw an exception unless you've explicitly defined an environment variable RAINFOREST and set it to TRUE. If you want to see this switch in action, you can export that variable in a terminal window, if you're using macOS or Linux.

Listing 4.21 Exporting an environment variable

```
export RAINFOREST=TRUE
```

Then you can execute the code from listing 4.20 again, in the same terminal window, without getting exceptions. That's similar to how you can use this in your production feature-generation jobs. Using any of several different mechanisms in your configuration, build, or job-orchestration functionality, you can ensure that this variable is set properly for rainforest feature-generation jobs and not set for global feature-generation jobs. A new engineer creating a new feature-generation job for some other purpose would have no reason to set this variable. If that engineer misused the rainforest feature generator, that misuse would immediately manifest the first time the job was executed in any form.

Configuration

Using environment variables is one of many different methods to configure components of your machine learning system. It has the advantage of being simple to get started with and broadly supported.

(continued)

As your machine learning system grows in complexity, you'll want to ensure that you have a well-thought-out plan for dealing with configuration. After all, properties of your machine learning system set as configurations can determine a lot about whether it remains responsive in the face of errors or changes in load. Part 3 of this book addresses most of these issues, where we consider the challenges of operating a machine learning system. The good news is that you'll find a lot of versatile tools from the Scala and big data ecosystems that will help you tame some of the complexity of dealing with configurations.

4.6 Applications

You're probably not an arboreal animal, and you may not even operate a microblogging service. But if you're doing machine learning, you're probably building features at some point.

In advertising systems, you can build features that capture users' past interactions with various types of products. If a user spends all afternoon looking at different laptops, you probably want to show them an ad for a laptop or maybe a case, but an ad for a sweater wouldn't make a lot of sense. That feature about which types of products the user had been looking at would help the machine-learned model figure that out and make the right recommendation.

At a political polling organization, you could build features pertaining to the demographics of different voters. Things like the average income, education, and home property value could be encoded into features about voting districts. Then those features could be used to learn models about which party a given voting district is likely to vote for.

The applications of features are as endless as the applications of machine learning as a technique. They allow you to encode human intelligence about the problem in a way that a model-learning algorithm can use that intelligence. Machine learning systems are not black-box systems that perform magic tricks. You, the system developer, are the one instructing it how to solve the problem, and features are a big part of how you encode that information.

4.7 Reactivities

This chapter covered a lot, but if you're still interested in learning more about features, there's definitely more to explore. Here are some reactivities to take you even deeper into the world of features:

- *Implement two or more feature extractors of your own.* To do this, you'll probably want to choose some sort of base dataset to work with. If you don't have anything meaningful at hand, you can often use text files and then extract features from the text. Spark has some basic text-processing functionality built in, which you may find helpful. Alternatively, random numbers organized into tabular data can work just as well for an activity like this. If you do want to use real data, the UCI Machine Learning Repository at https://archive.ics.uci.edu/ml/index.php is one of the best sources of datasets. Whatever data you use, the point is to decide for yourself what might be some interesting transformations to apply to this dataset.

- *Implement feature-selection functionality.* Using the feature extractors you created in the previous reactivity (or some other extractors), define some basis for including or excluding a given feature within the final output. This could include criteria like the following:
  - Proportion of nonzero values.
  - Number of distinct values.
  - Externally defined business rule/policy. The goal is to ensure that the instances produced by your feature-extraction functionality only include the features that you define as valid.
- *Evaluate the reactivity of an existing feature-extraction pipeline.* If you did the previous two exercises, you can evaluate your own implementation. Alternatively, you can examine examples from open source projects like Spark. As you examine the feature-extraction pipeline, ask yourself questions like the following:
  - Can I find the feature-transform function? Is it implemented as a pure function, or does it have some sort of side effects? Can I easily reuse this transform in other feature extractors?
  - How will bad inputs be handled? Will errors be returned to the user?
  - How will the pipeline behave when it has to handle a thousand records? A million? A billion?
  - What can I discern about the feature extractors from the persisted output? Can I determine when the features were extracted? With which feature extractors?
  - How could I use these feature extractors to make a prediction on a new instance of unseen data?

Summary

- Like chicks cracking through eggs and entering the world of real birds, features are our entry points into the process of building intelligence into a machine learning system. Although they haven't always gotten the attention they deserve, features are a large and crucial part of a machine learning system.
- It's easy to begin writing feature-generation functionality. But that doesn't mean your feature-generation pipeline should be implemented with anything less than the same rigor you'd apply to your real-time predictive application. Feature-generation pipelines can and should be awesome applications that live up to all the reactive traits.
- Feature extraction is the process of producing semantically meaningful, derived representations of raw data.

- Features can be transformed in various ways to make them easier to learn from.
- You can select among all the features you have to make the model-learning process easier and more successful.
- Feature extractors and transformers should be well structured for composition and reuse.
- Feature-generation pipelines should be assembled into a series of immutable transformations (pure functions) that can easily be serialized and reused.
- Features that rely on external resources should be built with resilience in mind.

We're not remotely done with features. In chapter 5, you'll use features in the learning of models. In chapter 6, you'll generate features when you make predictions about unseen data. Beyond that, in part 3 of the book, we'll get into more-advanced aspects of generating and using features.

5 Learning models

This chapter covers

- Implementing model-learning algorithms
- Using Spark's model-learning capabilities
- Handling third-party code

Continuing on our journey through the phases of a machine learning system, we now arrive at model learning (see figure 5.1). You can think of this part as that day when you were very young and looked up at a dark sky and decided that, based on past experience, it just might rain. The model you learned was *dark clouds lead to rain.* Although you may not remember it well, you figured out that model by reasoning about your past experiences with dark and bright days and whether you got rained on.

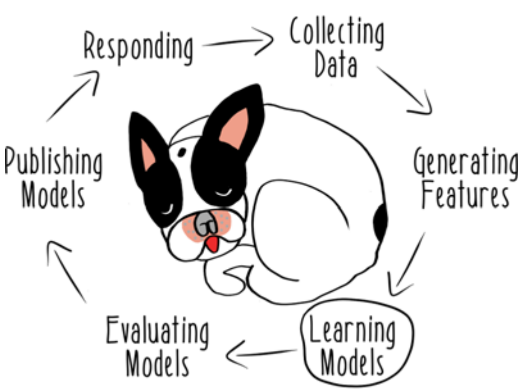


Figure 5.1 Phases of machine learning

That process you went through of reasoning about past experiences to develop a model that could be applied to future situations is analogous to what we do in the model-learning phase of a machine learning system. As I defined it in chapter 1, machine learning is learning from data, and this is the step where we do that learning. We'll run a model-learning algorithm over our features to produce a model. In the context of a machine learning system, a *model* is a way of encoding the mapping from features to concepts. It's a way of generalizing all the information in the training instances.

In software terms, a model is a program that was instantiated with instances that can now return predictions when called with features. This definition is shown in a stub implementation in the following listing.

Listing 5.1 A stub model

```
class Model(features: List[Instance]) {

  def predict(features: Set[Feature]): Label = ???

}
```

Instantiates a model with a list of instances (features and concept labels)

Predicts labels when given new sets of features

The implementations of model-learning algorithms are certainly far more complex than this stub implementation, but at a high level this is all we're doing from a software perspective. There can be incredibly sophisticated algorithms behind the process instantiating a new model, and I'll discuss some of them in this chapter. But the topic of how model-learning algorithms learn from data is huge and is the focus of countless other books. Accordingly, I'll try to cover just enough for you to be able to understand what model-learning algorithms do.

Then we'll do what most engineers do when implementing machine learning systems: call standard library implementations of common model-learning algorithms. Spark has some useful functionality for learning models in MLlib, its machine learning library. We'll build on our usage of MLlib from chapter 4 and take our pipelines all the way to learned models.

MLlib isn't the only machine learning library in the world, so we'll also spend time exploring how to work with libraries that aren't as easy to use from our Spark pipelines written in Scala. This is a common but challenging problem that data teams have to face all the time, and we'll explore some tactics to reduce the pain involved.

Being able to learn models from data is an incredible capability. It's one of the most significant achievements in the history of computer science. That's why, in this chapter, we're going to put it to use in the service of a noble goal: finding love.

5.1 Implementing learning algorithms

Timber is a mobile dating app for bears. Single male and female bears who are looking for love post their pictures in their profile on Timber. Then they can see pictures of other single bears that the app recommends to them. If a bear likes what it sees, the bear swipes right on the picture of that bear.

Behind the scenes of all of this furry romance is a sophisticated recommendation model, built by your data-science team at Timber. This chapter follows along as you begin to build that model. The team's goal is to predict which bears will like each other so that the Timber app can make recommendations to users. Only when two bears swipe right on each other are they connected, so it's crucial for the health of the app that users keep getting introduced to new bears they'd like to meet.

When two bears like each other and both swipe right, indicating that they'd like to meet, this is called a *match* by your Timber data-science team. The team plans to run all possible pairs of active users through the model and get predictions of which ones are a likely match. Only those predicted as matches (paired) will be shown to the users.

To begin building their model, your team only has its historical data about who their users are and which ones ended up matching. They decide to frame the problem of predicting which bears will match as a binary classification problem. Whether or not a given pair of users matched will be used as class label. All the data about the users will be used to build features.

As you saw in chapter 4, building features can be a very complicated endeavor. The Timber team has decided to start by building features around user similarity. When a bear signs up on Timber, they answer various questions to fill out their profile on the app:

- What's your favorite food?
- Do you like to go out or are you more of a cave-body?
- Do you want to have cubs someday?

In the first version of their feature-generation functionality, your team compared the answers of each pair of users to produce features saying whether or not their answers were the same. For example, if two bears both answered that their favorite food was salmon, that would be recorded as a `true` feature value, but if one bear preferred salmon and another preferred berries, then that would be recorded as a `false` feature value. This produces instances like the ones shown in table 5.1, which uses 0 for false and 1 for true.

Table 5.1 Similarity instances

Favorite food	Go out	Cubs	Match
1	0	0	0
1	1	1	1
0	1	1	1
0	1	0	0
1	1	1	0

Using similarity data like this, you could perform all sorts of ad hoc analyses and develop manual rules. But Timber has lots of users and is growing fast. It needs an automated system to reason about this data at scale.

5.1.1 Bayesian modeling

In their first implementation of the model-learning system, the team used a technique called *Naive Bayes*. Before we get into how Naive Bayes works and how it can be implemented, we need to cover *Bayes' rule*, which is the basis for the Naive Bayes technique. Bayes' rule is a foundational technique used in some forms of statistics.

To understand Bayes' rule, let's simplify your data even more. Let's assume that you only know whether two bears share the same favorite food and if they matched. Then your team could talk about the probability that two bears will match if they share the same favorite food.

The following listing introduces some notation to discuss how you could use Bayes' rule in this case.

Listing 5.2 Notation for Bayes' rule

```
F
M
P(F|M)
P(M|F)
P(M|F) = P(F|M)*P(M)÷P(F)
```

Notation for the same favorite food

Notation for a match on the app

Probability of having the same favorite food, given they were a match

Probability that two bears will be a match given they have same favorite food

Calculation for this probability

Bayes' rule states that this probability can be calculated as follows: `P(F|M)*P(M)÷P(F)`, the probability of having the same favorite food, given that they were a match, multiplied by the probability of a match, divided by the probability of having the same favorite food.

Using the data in table 5.1, you can calculate those values.

Listing 5.3 Calculating an application of Bayes' rule

```
2/5 = 0.4
3/5 = 0.6
1/2 = 0.5
0.5 * 0.4 ÷ 0.6 = 0.33
```

Probability of a match in that dataset

Probability of having the same favorite food

Probability of having the same favorite food given they were a match

Probability of being a match given two bears share the same favorite food

This calculation results in a 1/3 probability of being a match given that two bears share the same favorite food.

Bayes' rule can be generalized to apply to many different features using the Naive Bayes technique. Let's use G to represent both bears preferring to go out or stay in the den and C to indicate agreeing on the decision to have cubs. Let's also refer to a specific combination of feature values as V. That's meant to provide a way to talk about the probability of liking the same food and both liking to go out, but not agreeing on raising cubs (true, true, false) as different from bears who agree on everything (true, true, true).

With this notation in place, you can now use Bayes' rule with all your team's original features as P(F|M)*P(G|M)*P(C|M)*P(M)÷P(V). By multiplying all the probabilities together, you can get the probability of a match, given the values of all of your features.

Independence

It's often *not* valid statistically to assume that you can multiply all those probabilities together. Doing so presumes that the probabilities of each feature value are independent from each other—that they never vary together. That's why this technique is referred to as *naive*, because it fails to take into account the possibility of dependence between the features. Despite that limitation, Naive Bayes has been empirically shown to be a useful technique.

Table 5.2 shows what all those probabilities are, based on your dataset.

Table 5.2 Probabilities

Term	Fraction	Probability
P(F|M)	1/2	0.5
P(G|M)	2/2	1.0
P(C|M)	2/2	1.0
P(M)	2/5	0.4

To be able to calculate the probability of a given combination of feature values, P(V), you'll have to try to predict on some data.

Let's say you're trying to compute the probability that Ping, a panda, and Greg, a grizzly bear, will be a match (figure 5.2). They don't agree on food or whether they'd like to have cubs someday, but they are both den-bodies, not liking to go out much. Does the data indicate that these two are likely to be a match on Timber?

In this case, P(V) is the sum of 0.2, 0.1, and 0.2 or 0.5. Now you can evaluate Bayes' rule. Plugging in the values from table 5.2, you get 0.5 * 1.0 * 1.0 * 0.4 / (0.2 + 0.1 + 0.2) = 0.4. Bayes' rule is modestly confident that the two of them will hit it off, so the app might recommend them to each other, depending on the scores of their other candidate matches.

Figure 5.2 Timber profile screens

5.1.2 Implementing Naive Bayes

Now that you've seen how Naive Bayes works, let's see how you can implement it so it can be run as part of Timber's production model-learning pipeline. To begin, you need to build up some training instances to use to train your model. You'll build on the techniques you learned in chapter 4 for handling features and extend that code. The following listing shows some of the types you used to manage features and labels before, as well as a way of bringing them together into instances.

Listing 5.4 Features, labels, and instances

```
trait FeatureType[V] {
  val name: String
}

trait Feature[V] extends FeatureType[V] {
  val value: V
}

trait Label[V] extends Feature[V]

case class BooleanFeature(name: String, value: Boolean) extends
➥ Feature[Boolean]

case class BooleanLabel(name: String, value: Boolean) extends Label[Boolean]

case class BooleanInstance(features: Set[BooleanFeature], label: BooleanLabel)
```

Feature-type implementation from chapter 4, requiring a type and a name

Feature implementation from before, requiring a value of a given type

Label implementation from before, defining class labels as a special type of feature

Defines Boolean features

Defines Boolean labels

Defines Boolean instances to contain sets of Boolean features and a Boolean label

This is all review from chapter 4. With these types in place, you'll be able to set up your model-learning algorithm to reason about features and labels in the form of instances.

Now you can implement the Naive Bayes model. First, you'll assume that you can initialize your implementation with some list of training instances. Listing 5.5 creates a simple training instance to allow you to get started. You'll factor this out to a constructor parameter shortly, but this fixtured version of the data will allow you to start writing code to process the instances.

Listing 5.5 Training instances

```
val instances = List(BooleanInstance(
    Set(BooleanFeature("food", true),
      BooleanFeature("goOut", true),
      BooleanFeature("cubs", true)),
    BooleanLabel("match", true)))
```

Creates a list containing only a single instance

Creates a set of all features

Creates a label value

Then you'll operate on the instances within your training set that have `true` labels and calculate `P(M)`, the overall probability of a match.

Listing 5.6 Positive training instances

Filters instances to only the ones with true class labels

```
val trueInstances = instances.filter(i => i.label.value)
val probabilityTrue = trueInstances.size.toDouble / instances.size
```

Calculates P(M), the overall probability of a match

Then you can build up the probabilities of each given feature, given a match (for example, `P(F|M)`, the probability of having the same favorite food given that a couple was a match). Because you could have an arbitrary number of features, you'll build up the unique set of all features first and then produce the probabilities for each one you find.

Listing 5.7 Feature probabilities

Builds a set of all unique feature types in the training set

```
val featureTypes = instances.flatMap(i => i.features.map(f => f.name)).toSet

val featureProbabilities = featureTypes.toList.map {
  featureType =>
    trueInstances.map { i =>
      i.features.filter { f =>
        f.name equals featureType
      }.count {
        f => f.value
      }
    }.sum.toDouble / trueInstances.size
}
```

Maps over all distinct feature types

For each feature, maps over all instances with true class labels

Counts a feature value if it's true

Uses a filter to match to a current feature by name

Sums up all positive examples, dividing by the total number of true instances

Now that you've calculated all the terms in the numerator of the equation, you can calculate the entire numerator. The following listing uses multiplication, `*`, as a higher-order function to reduce over your list of feature probabilities before multiplying by `P(M)`, the probability of a match.

Listing 5.8 Numerator

```
val numerator = featureProbabilities.reduceLeft(_ * _) * probabilityTrue
```

Now you just need to be able to calculate `P(V)`, the probability of a given feature vector. The next listing shows a simple function to perform that calculation for an arbitrary set of features.

Listing 5.9 Feature vector

```
def probabilityFeatureVector(features: Set[BooleanFeature]) = {
  val matchingInstances = instances.count(i => i.features == features)
  matchingInstances.toDouble / instances.size
}
```

Counts the instances with the matching feature values

Divides the number of matching instances over the total number of instances to get the P(V)

With all the pieces now in place, writing the `predict` function is straightforward. Given a new feature vector, calculate the denominator and divide the numerator by it.

Listing 5.10 Prediction function

```
def predict(features: Set[BooleanFeature]) = {
  numerator / probabilityFeatureVector(features)
}
```

Calculates probability by dividing a precomputed numerator by a denominator for a given instance

As a final refactor, let's put all this code inside a class and pass instances at the time of construction. The next listing shows all the modeling code from before, refactored into a class.

Listing 5.11 Naive Bayes model

```
class NaiveBayesModel(instances: List[BooleanInstance]) {

  val trueInstances = instances.filter(i => i.label.value)
  val probabilityTrue = trueInstances.size.toDouble / instances.size

  val featureTypes = instances.flatMap(i => i.features.map(f => f.name)).toSet

  val featureProbabilities = featureTypes.toList.map {
    featureType =>
      trueInstances.map { i =>
        i.features.filter { f =>
          f.name equals featureType
        }.count {
```

Instantiates the model with training instances

```
          f => f.value
        }
      }.sum.toDouble / trueInstances.size
  }

  val numerator = featureProbabilities.reduceLeft(_ * _) * probabilityTrue

  def probabilityFeatureVector(features: Set[BooleanFeature]) = {
    val matchingInstances = instances.count(i => i.features == features)
    matchingInstances.toDouble / instances.size
  }

  def predict(features: Set[BooleanFeature]) = {
    numerator / probabilityFeatureVector(features)
  }

}
```

This implementation doesn't need to do much work when new feature vectors come in for prediction. As part of the instantiation of the class, the model was effectively trained, and the model parameters, the various probabilities, are now held in the internal state of the given model instance.

You can try out this code by writing a simple test.

Listing 5.12 Testing the Naive Bayes model

```
test("It can learn a model and predict") {
  val trainingInstances = List(
    BooleanInstance(
      Set(BooleanFeature("food", true),
        BooleanFeature("goOut", true),
        BooleanFeature("cubs", true)),
      BooleanLabel("match", true)),
    BooleanInstance(
      Set(BooleanFeature("food", true),
        BooleanFeature("goOut", true),
        BooleanFeature("cubs", false)),
      BooleanLabel("match", false)),
    BooleanInstance(
      Set(BooleanFeature("food", true),
        BooleanFeature("goOut", true),
        BooleanFeature("cubs", false)),
      BooleanLabel("match", false)))

    val testFeatureVector = Set(BooleanFeature("food", true),
        BooleanFeature("goOut", true),
        BooleanFeature("cubs", false))

    val model = new NaiveBayesModel(trainingInstances)

    val prediction = model.predict(testFeatureVector)

    assert(prediction == 0.5)
}
```

Sets up a test of model learning and prediction

Creates some training instances to learn the model with

Creates a test feature vector to predict on

Instantiates a class and, thus, trains the model

Predicts on the test feature vector

Asserts that the result is 0.5

This implementation is a pretty good representation of the mathematical process that you went through by hand before. It took your Timber data-science team some time to get this implementation figured out. Though it definitely predicts matches, this implementation isn't perfect. For one thing, it can only handle Boolean feature values and Boolean labels. It also doesn't handle things like not finding the exact feature vector in the training instances that you're trying to predict on now.

There are things to like about this approach to analyzing data. By implementing the mathematics of Naive Bayes as runnable code, you could run it over as many training instances with as many features as your server can handle. Of course, this implementation is consistent in its approach, as contrasted with any ad hoc, manual-analysis approach. Another nice feature of this implementation is that it's verifiable—you can write more tests to demonstrate that this implementation is correct.

I hope you agree that there's a lot of benefit to using machine learning to build a recommendation model. You can certainly take things even further. Naive Bayes is just one algorithm; there's no reason to believe that it's the best algorithm for this problem. If you want to explore more learning algorithms (and I hope you do!), then you may not want to implement them all yourself. There's no need to build everything from scratch. Tools like Spark and MLlib have a lot of functionality that you can use to explore model-learning algorithms even more.

5.2 Using MLlib

In chapters 2 and 4, you got a taste of Spark's machine learning capabilities from MLlib, its machine learning library. It has a wide range of machine learning functionality that can help you explore various modeling techniques.

Infinite Data

For example, MLlib already has a very capable and sophisticated implementation of Naive Bayes. Perhaps most importantly, you can train the MLlib implementation of Naive Bayes over datasets of arbitrary size using Spark's elasticity capabilities. The comparatively simpler implementation in the previous section is limited to a single Java process running on a single server.

Engineers are often excited to port legacy model-learning implementations over to Spark implementations. The code becomes simpler, the job can become much more scalable, and taking advantage of other big data tools becomes easier. Data scientists also stand to gain from this transition. When production machine learning systems use MLlib functionality, data scientists can experiment with MLlib's wide range of algorithms. When a modeling approach works well, incorporating it into the production-data pipelines is easy.

5.2.1 Building an ML pipeline

Let's get back to the problem of building a better recommendation model for bears seeking love. The Timber team begins to build out the second version of their model-learning functionality. This pipeline consumes the features and labels generated by the upstream feature-generation pipeline. The instances for training and testing the model are persisted in LIBSVM format in flat files. In the future, the team hopes to put this data into a database, but for this early stage of development, LIBSVM-formatted files work fine. The next listing shows a sample of that data.

Listing 5.13 Instances in LIBSVM format

```
1 1:3 2:4 3:2
0 1:2 2:4 3:1
0 1:1 2:2 3:2
```

As discussed in chapter 1, instances record the label value in the first position, followed by pairs of feature identifiers to feature values.

The first line of data in listing 5.13 is a true instance with the value `3` for the first feature, the value `4` for the second feature, and the value `2` for the third feature. Formats for storing features like LIBSVM are known as *sparse formats*, meaning features can be absent if their values are `0` or unknown. Sparse formats are an alternative to *dense formats*, where all features must be present, regardless of their value. Sparse formats are intrinsically more flexible, so the Timber team decides to use LIBSVM for now. They hope that will make it easier when they eventually adapt their data architecture to persist this data in a document database, with a sparse representation of their features.

Data formatted in this way can easily be loaded into the new Spark pipeline, once you establish the standard configuration for the job.

Listing 5.14 Loading LIBSVM instances

Creates a new Spark session for the job

```
val session = SparkSession.builder.appName("TimberPipeline").getOrCreate()

val instances = sqlContext.read.format("libsvm").load("/match_data.libsvm")
```

Loads an instance data into a DataFrame

The functionality that loads LIBSVM data understands the structure of the format, so a column named `features` and a column named `label` will be created automatically.

With this data loaded, depending on how you want to learn a model, you can use this `DataFrame` directly. In this case, let's take full advantage of the capabilities of the Spark ML package to perform a bit of preprocessing. You can use two indexers to process the features and the labels in the `DataFrame` of instances. These indexers will analyze how many different values of features and labels it sees and put that metadata back into the `DataFrame`. In this example data, you have all categorical features, so the indexers will detect that and record that in the metadata in the `DataFrame`. These category values as well as the label values will be mapped to an internal representation of the values, which can improve the model-learning process. The values and their associated metadata will later be used by the model-learning algorithm.

Listing 5.15 Detecting feature and label values

```
val labelIndexer = new StringIndexer()
  .setInputCol("label")
  .setOutputCol("indexedLabel")
  .fit(instances)

val featureIndexer = new VectorIndexer()
  .setInputCol("features")
  .setOutputCol("indexedFeatures")
  .fit(instances)
```

Sets up the StringIndexer to process labels in instances of DataFrame

Reads from a label column

Writes transformed labels to the indexedLabel column

Sets the DataFrame to process

Sets up the VectorIndexer to process feature vectors in instances of DataFrame

Reads from the features column created when the LIBSVM file was read and transformed into a DataFrame

Writes processed features to the indexedFeatures column

Sets the DataFrame to process

It's worth noting that nothing has been processed yet. Each of these is a `PipelineStage` that you'll fully compose later and then execute. This is more of Spark's lazy compositional style.

Next, you need to split your data into training and testing sets, as shown in listing 5.16. With the testing set, you'll perform predictions on data that your model hasn't seen before. Chapter 6 discusses the topic of testing models in more detail. For the moment, it's enough to set aside a random 20% of your data for later use.

Listing 5.16 Splitting instances into training and testing sets

```
val Array(trainingData, testingData) = instances.randomSplit(Array(0.8, 0.2))
```

Splits the data: 80% for training and 20% for testing

Now you can finally set up your model-learning algorithm. Your team at Timber decides to start with a decision-tree model. A *decision tree* is a technique that divides the classification decision into a series of decisions about each feature. Decision trees work well with categorical features like the ones you're using, so the team is optimistic that decision trees may be a good place to start.

Listing 5.17 Decision-tree model

```
val decisionTree = new DecisionTreeClassifier()
  .setLabelCol("indexedLabel")
  .setFeaturesCol("indexedFeatures")
```

Sets up a new decision-tree classifier

Sets which column the label is in

Sets which column the features are in

Again, nothing has been learned here; this is just another `PipelineStage`.

Before you put this pipeline together and execute it, you need to put in one last stage to convert from the internal representation of labels to your original one. The indexers you used in listing 5.16 have the ability to come up with new identifiers for the features and labels you provided. But that information isn't lost, so you can turn all predictions made by the model back into your original representation, where 1 corresponded to a match. This stage, where you convert back from those internal label representations into the original labels, is shown in the following listing.

Listing 5.18 Converting labels

```
val labelConverter = new IndexToString()
  .setInputCol("prediction")
  .setOutputCol("predictedLabel")
  .setLabels(labelIndexer.labels)
```

Sets up the transformer to convert from the internal label value back to the original value

Sets the column where predictions can be found

Sets which column to write converted labels to

Sets what were original labels, using the label indexer

Now the pipeline can be composed using Spark ML's `Pipeline` construct. The next listing takes the four `PipelineStages` that you've defined and composes them into a single runnable unit.

Listing 5.19 Composing a pipeline

```
val pipeline = new Pipeline()
  .setStages(Array(labelIndexer, featureIndexer , decisionTree,
  ➥ labelConverter))
```

Sets up a Pipeline

Sets the stages to execute in order

This declarative method of assembling the phases of a pipeline to be run makes it very easy to create reusable pipeline stages and build variant versions of pipelines using those stages. With this composed pipeline, you can now execute the pipeline by processing your features and learning the model.

Listing 5.20 Learning the model

```
val model = pipeline.fit(trainingData)
```

Executes the pipeline and learns the model

After all that structuring and composition, this invocation of `fit` is the step that runs the whole model and eventually learns the model.

Now that you have a model, let's demonstrate that it can do something useful. Typically, at this point in a pipeline, people attempt to evaluate the model over unseen data. This phase is called *testing* the model, as contrasted with *training* the model. Chapter 6 discusses this process more, but let's see a bit of what your model can do by getting it to predict on some of your testing data.

Listing 5.21 Predicting

```
val predictions = model.transform(testingData)

predictions.select("predictedLabel", "label", "features").show(1)
```

Uses the model to predict on testing data

Prints one of the predictions, the true label, and the features used

This shows that your model has been learned and you can now predict on unseen data. At this point, you may be curious about what exactly this model is. Not all model-learning algorithms produce models that are easy for humans to reason about. But the decision tree that you used is something you can look at and understand the learned structure of decisions that the model will make.

Listing 5.22 Inspecting the model

Gets a decision-tree model from the executed pipeline

```
val decisionTreeModel = model.stages(2)
  .asInstanceOf[DecisionTreeClassificationModel]

println(decisionTreeModel.toDebugString)
```

Casts it to an instance of the DecisionTreeClassificationModel

Prints the resulting model

Printing the model should produce something like the model shown in the following listing.

Listing 5.23 Inspecting the model

Describes the type and structure of the model

```
DecisionTreeClassificationModel (uid=dtc_f0924358349f) of depth 1 with 3 nodes
  If (feature 0 in {0.0,1.0})
   Predict: 0.0
  Else (feature 0 not in {0.0,1.0})
   Predict: 1.0
```

Explains the decision the tree model will make when predicting on new data

This is a simple model; you're not working with much data, and you haven't defined too many features. But all the code you just wrote could be used to scale up to more-complex models learned over more features, as you'll see in the next section.

Beyond being a flexible implementation that can support easy changes in modeling approaches, this pipeline is also quite reactive, thanks to being built on top of all the powerful infrastructure provided by Spark. You could easily and quickly deploy this pipeline to production at scale in the same way you could for any other Spark application.

5.2.2 Evolving modeling techniques

The other aspect of this design that the team is excited about is that evolving their approach to a given modeling problem is quick and easy. After this first decision-tree model-learning pipeline is completed, your team decides to try a related but more sophisticated technique: *random forests*. This technique still uses decision trees, but it uses many of them in combination. Using multiple models in combination with each other, called an *ensemble*, is one of the most powerful techniques in all of machine learning. Rather than trying to get a single model learned on a single dataset using a single set of model-learning parameters, ensemble modeling techniques improve performance by creating different models, potentially with different strengths in combination. This is a sort of possible-worlds technique, where you assume that different models might be right about different aspects of the concept.

Even though this technique is more sophisticated, and the implementation of the algorithm is more complicated, this isn't any harder for you to use than the basic decision-tree model from before. The code is nearly identical to what you wrote earlier.

Listing 5.24 Learning a random-forest model

```
val randomForest = new RandomForestClassifier()
  .setLabelCol("indexedLabel")
  .setFeaturesCol("indexedFeatures")
```

Sets up a random-forest classifier

Creates a slightly different pipeline, using a random-forest classifier

```
val revisedPipeline = new Pipeline()
  .setStages(Array(labelIndexer, featureIndexer, randomForest,
  ➥ labelConverter))
```

```
val revisedModel = revisedPipeline.fit(trainingData)
```

Executes the pipeline

```
val randomForestModel = revisedModel.stages(2)
  .asInstanceOf[RandomForestClassificationModel]
```

Extracts the learned model from the pipeline for inspection

```
println(randomForestModel.toDebugString)
```

Prints a representation of the learned model

If you execute the code in listing 5.24, you should see a much larger model printed than you saw for the simple decision-tree model.

Listing 5.25 A random forest model

Describes the type and structure of the model

```
RandomForestClassificationModel (uid=rfc_fbfe64b4a427) with 20 trees
  Tree 0 (weight 1.0):
    If (feature 1 in {0.0})
     Predict: 0.0
    Else (feature 1 not in {0.0})
     Predict: 1.0
  Tree 1 (weight 1.0):
```

Explains the various decision-tree models within the larger random-forest model

Given decision tree within the forest, weighted equally to the other trees

Predicts 0, or false

First branch of a given decision tree, testing whether the feature is 0

Second branch of the decision tree, when the feature is 1

Predicts 1, or true

```
    If (feature 0 in {0.0})
     Predict: 0.0
    Else (feature 0 not in {0.0})
     Predict: 1.0
  Tree 2 (weight 1.0):
    If (feature 0 in {1.0})
     Predict: 0.0
    Else (feature 0 not in {1.0})
     Predict: 1.0
  Tree 3 (weight 1.0):
    If (feature 0 in {0.0,1.0})
     Predict: 0.0
    Else (feature 0 not in {0.0,1.0})
     Predict: 1.0
  Tree 4 (weight 1.0):
    Predict: 1.0
  Tree 5 (weight 1.0):
    If (feature 1 in {0.0})
     Predict: 0.0
    Else (feature 1 not in {0.0})
     If (feature 0 in {1.0})
      Predict: 0.0
     Else (feature 0 not in {1.0})
      Predict: 1.0
  Tree 6 (weight 1.0):
    If (feature 2 in {0.0})
     Predict: 0.0
    Else (feature 2 not in {0.0})
     Predict: 1.0
  Tree 7 (weight 1.0):
    Predict: 1.0
  Tree 8 (weight 1.0):
    Predict: 0.0
  Tree 9 (weight 1.0):
    If (feature 0 in {1.0})
     Predict: 0.0
    Else (feature 0 not in {1.0})
     Predict: 1.0
  Tree 10 (weight 1.0):
    If (feature 0 in {0.0,1.0})
     Predict: 0.0
    Else (feature 0 not in {0.0,1.0})
     Predict: 1.0
  Tree 11 (weight 1.0):
    If (feature 1 in {0.0})
     Predict: 0.0
    Else (feature 1 not in {0.0})
     Predict: 1.0
  Tree 12 (weight 1.0):
    If (feature 0 in {0.0,1.0})
     Predict: 0.0
    Else (feature 0 not in {0.0,1.0})
     Predict: 1.0
```

```
Tree 13 (weight 1.0):
  Predict: 1.0
Tree 14 (weight 1.0):
  Predict: 0.0
Tree 15 (weight 1.0):
  Predict: 0.0
Tree 16 (weight 1.0):
  If (feature 0 in {0.0})
   Predict: 0.0
  Else (feature 0 not in {0.0})
   Predict: 1.0
Tree 17 (weight 1.0):
  Predict: 0.0
Tree 18 (weight 1.0):
  Predict: 0.0
Tree 19 (weight 1.0):
  If (feature 2 in {0.0})
   Predict: 0.0
  Else (feature 2 not in {0.0})
   If (feature 1 in {0.0})
    Predict: 0.0
   Else (feature 1 not in {0.0})
    Predict: 1.0
```

That's a lot more modeling sophistication for a trivial change in your code! This is awesome power to have in a fast-growing dating-app startup. You can easily try out a wide range of modeling strategies without having to implement them all yourself, and when you find something that works, it can be rapidly deployed over your whole dataset. That should add up to a lot more happy ursine couples!

5.3 Building facades

With all these improvements in your model-learning pipeline, the team is looking to take on more-significant technical challenges. They decide they want to be able to use cutting-edge techniques like *deep learning*. Deep learning is a recently developed technique in machine learning that builds on earlier work in *neural networks*, a technique for building learning algorithms that mimic the structure of the brains of animals.

Although MLlib has support for some forms of neural networks, it doesn't offer a lot of the deep learning functionality you may want out of the box, so you'll need to figure out how to make that work. There are some efforts to enable deep learning on Spark, but your team really wants to be able to take advantage of the latest advances as they happen. The current fashion in deep learning research is to use technologies accessed via Python, so, ideally, you should find a way to use those technologies in cooperation with your existing Scala applications. Getting your Scala code to drive bleeding-edge Python research code certainly could be tricky, but you'll be able to pull it off using a pattern called a *facade*.

The facade pattern (also known as a *wrapper*, an *adapter*, or a *connector*) is a relatively well-established technique for integrating third-party libraries. The idea is to write some new code in your application with the sole responsibility of acting as an intermediary between your application and a third-party library. This intermediary is the facade, and it's in charge of harmonizing the API of the underlying library with the expectations of your application (figure 5.3).

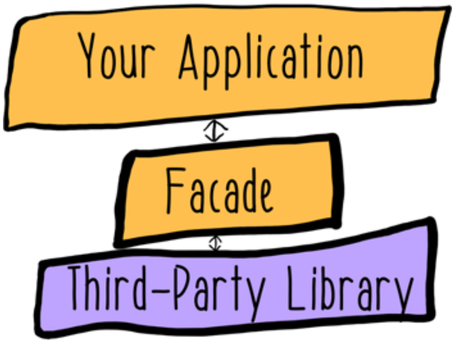


Figure 5.3 Facade pattern

5.3.1 Learning artistic style

For your first foray into the world of deep learning, you'll transform user photos into the style of famous paintings. The idea is to help bears get beyond superficial first impressions and consider what inner beauty might be lurking just below the fur. To perform these image transformations, you'll use an exciting new algorithm called a *neural algorithm of artistic style*, or just *style net* for short. What the style-net technique does is transfer the artistic style of a given image to the content of another image.

The open source implementations of the style-net algorithm all rely on various deep learning frameworks. Your team has chosen to use an implementation that relies on the TensorFlow framework. TensorFlow is powerful machine learning framework with particularly good support for deep learning. It was originally developed at Google as part of the company's research on machine learning, and deep learning, specifically.

To install TensorFlow, follow the latest instructions at www.tensorflow.org. Once you've installed TensorFlow, you'll want to clone the implementation of the style-net algorithm I've provided (https://github.com/jeffreyksmithjr/neural-art-tf). It's an adaptation of an implementation called neural-art-tf (https://github.com/woodrush/neural-art-tf) with some important changes. If you want to clone both repositories for comparison's sake, you can.

To begin learning new models, you first need to download an existing model. You'll use this model as part of the input to learning a new model of a given style image—a famous painting, in this case. The approach you'll use relies on a pretrained model from the Visual Geometry Group (VGG) at Oxford. You can download this model from http://mng.bz/iYlL. You'll need the .caffemodel file, the model itself, and the .prototxt file, which describes the structure of the model.

This model, a large, deep model originally intended for use in image recognition, is provided in Caffe format. Caffe is another deep learning framework, with different strengths and weaknesses than TensorFlow.

To make this model usable by the TensorFlow implementation, you'll need to use a simple conversion utility in the neural-art-tf project.

Listing 5.26 Converting a Caffe model for TensorFlow

```
> python ./kaffe/kaffe.py [path.prototxt] [path.caffemodel] vgg
```

Converts the model and saves it to a file called vgg

Once you've converted the model file, you can begin learning the model of artistic style for a given painting to apply to a picture of a bear, using the original neural-art-tf implementation. It was originally intended to be used by being called as a command-line utility with various arguments for things like the path to the model, the file, and the number of iterations. Rather than take those arguments from the command line, let's first move them up to be parameters in a method.

Listing 5.27 Method parameters

```
def produce_art(content_image_path, style_image_path, model_path,
  ➥ model_type, width, alpha, beta, num_iters):
```

Then you need to find a way to make this method available to your Scala code. Let's try to turn this simple script to run the code into a service that can be called by your primary pipeline. To do that, you'll use a Python library called Pyro. Like most Python libraries, it's pretty easy to install using `pip`.

Listing 5.28 Installing Pyro

```
> pip install Pyro4
```

With Pyro, you can take Python functionality in the scripts in neural-art-tf and make it available to clients accessing it via the network.

Listing 5.29 Setting up

```
class NeuralServer(object):
    def generate(self, content_image_path, style_image_path, model_path,
    ➥ model_type, width, alpha, beta, num_iters):
        produce_art(content_image_path, style_image_path, model_path,
        ➥ model_type, width, alpha, beta, num_iters)
        return True

daemon = Pyro4.Daemon()
ns = Pyro4.locateNS()
uri = daemon.register(NeuralServer)
ns.register("neuralserver", uri)
daemon.requestLoop()
```

Creates a class to represent your server

Defines a method to expose to clients

Passes arguments to the underlying produce_art method

Returns True as the status once the job has completed

Sets up the Pyro daemon

Locates the name server

Registers a server class with the Pyro daemon as a Pyro object

Registers the Pyro object with the name server

Starts the event loop to wait for calls to the Pyro object

All of listing 5.29 makes it simple for other users of Pyro to find and use the method that runs the model-learning jobs. The script will run as a *daemon*, a small utility running in the background waiting to serve requests. You've named it `neuralserver` and attempted to register it with a name server. The name server will be responsible for routing requests for this Pyro object when it's asked for by name. For all this to work, you'll need to start up a Pyro name server and then start the revised server script.

Listing 5.30 Starting a name server

```
> pyro4-ns

> python neural-art-tf.py
```

Starts a name server

Starts a job-running server

Now let's figure out how to send requests to run jobs from your Scala client application to this Python server application. One of the nice features of Pyro is that it provides client libraries for use with other runtimes. You can use the Java Pyrolite client library in your Scala code (https://pythonhosted.org/Pyro4/pyrolite.html). This will allow you to build a facade around the Python code and the problems inherent in long-running jobs, using all the same reactive techniques you've used elsewhere in this book.

To build a more reactive facade, the first thing you can do is ensure more type safety around the parameters being passed. For example, the command-line version of neural-art-tf uses a bunch of strongly typed arguments for things such as the type of model being used. The number of valid model types is tiny compared to the huge range of possible strings. In Scala, you can capture those arguments that can only take one of a known set of values as an enumeration.

Listing 5.31 A model-type enumeration

```
object ModelType extends Enumeration {
  type ModelType = Value
  val VGG = Value("VGG")
  val I2V = Value("I2V")
}
```

Creates an Enumeration named ModelType

Defines a ModelType value for use in the type system

Defines VGG as one valid type of model

Defines I2V (for Image2Vector) as another type of model

You can then use type safety to create a better definition of what constitutes a valid configuration for your job. Let's use a case class to encapsulate what is a valid job configuration. Moreover, in the following listing, let's set some default values so you don't have to pass all the knowledge about the job setup every time the job is run.

Listing 5.32 A job-configuration case class

```
case class JobConfiguration(contentPath: String,
                            stylePath: String,
                            modelPath: String,
                            modelType: ModelType,
                            width: Integer = 800,
```

Path to the content image, user picture

Path to the style image, famous painting

Path to the pretrained model to use

Type of pretrained model

Width of the resulting image, defaulting to 800 pixels

Beta parameter, for weight of style, defaulting to 200.0

Alpha parameter, for weight of content, defaulting to 1.0

```
alpha: java.lang.Double = 1.0,
beta: java.lang.Double = 200.0,
iterations: Integer = 5000)
```

Number of iterations to learn the model over, defaulting to 5000

Note that because the Pyrolite client library is a Java library, your case class needs to use Java types for doubles and integers. But beyond specifying that in the case-class type signatures, there's nothing else you need to do; Scala will automatically perform the conversions from Scala types to the underlying Java types.

Using this configuration case class makes it easier for you to ensure that your configurations are valid. With a valid configuration, such as the one in the next listing, you can submit your job with more confidence that the arguments are sufficient for the Python-server application.

Listing 5.33 A job configuration

Creates a configuration with content image of an attractive sloth bear

Sets the style image to a famous painting

```
val jobConfiguration = JobConfiguration("./sloth_bear.png",
  "./senecio.jpg",
  "./vgg",
  ModelType.VGG,
  iterations = 1000)
```

Model file

Type-safe model type

Overrides the number of iterations

Then locate the name server and connect to the Pyro object for your job server. In this example, you're only using the networking capabilities of Pyro to give you a way of communicating between your Python program and your Scala program, so you won't need to deal with any networking complexity, although the process would be similar in a distributed implementation.

Listing 5.34 Connecting to the server

Finds the name server on the same machine, passing in a null host

Looks up the Pyro object by its name, neuralserver

```
val ns = NameServerProxy.locateNS(null)
val remoteServer = new PyroProxy(ns.lookup("neuralserver"))
```

Now, you can send your configuration off to be processed—but let's think through what that means first. That request is to a separate running process, and it could have just as easily been a separate process on another machine. The job might take quite a while to run. Deep learning techniques are famous for requiring very powerful hardware and taking a long time to execute. If you really want your application to be reactive, you should implement some sort of timeout. The duration of that timeout should be based on your expectation of the normal runtime of the model-learning algorithm. In the example in listing 5.35, I've arbitrarily chosen one hour. With a timeout in place, you can detect whether the model-learning process has taken too long and should be treated as a failure.

Listing 5.35 Using a timeout

```
val timeoutDuration = 60 * 60 * 1000

def timedOut = Future {
  Thread.sleep(timeoutDuration)
  false
}

def callServer(remoteServer: PyroProxy, jobConfiguration:
➥ JobConfiguration) = {
  Future.firstCompletedOf(
    List(
      timedOut,
      Future {
        remoteServer.call("generate",
          jobConfiguration.contentPath,
          jobConfiguration.stylePath,
          jobConfiguration.modelPath,
          jobConfiguration.modelType.toString,
          jobConfiguration.width,
          jobConfiguration.alpha,
          jobConfiguration.beta,
          jobConfiguration.iterations).asInstanceOf[Boolean]
      }))
}
```

Sets up a one-hour timeout in milliseconds

Creates a timeout future

Returns a false value to indicate failure to complete in time

Creates a function to wrap calling of the Python server application

Sets a timeout as the first future completed of the two

The timeout future

Sets up a future to call the method on the Pyro object

Casts the resulting value to a Boolean

Because you're dealing with a Python program on the other end of this call, you don't get any type guarantees about what it's going to send back, so you'll need to typecast the return value to a `Boolean`. Because the model-learning process is going to occur in an entirely separate process, this Scala program will only ever know if that Python program sends back a true value to say that it's been successful. Given that limited knowledge, this timeout mechanism prevents your Scala application from waiting forever in the event that something has gone wrong with the model-learning process.

Finally, you can call this function:

```
val result = callServer(remoteServer, jobConfiguration)
```

Invokes the server-calling function with a sample configuration

Containment

The value in `result` is a `Future` of a `Boolean`, so you can then integrate the value of the future when it's completed, with the rest of your reactive Scala application. All the details of how the Python implementation works are abstracted down to a small interface. The Scala pipeline code understands what valid job configurations are and how long they should take. Should any errors occur in the execution of the TensorFlow implementation of the style-net algorithm, the Scala application will detect them using a timeout and react accordingly. Those errors in the model-learning process will be fully contained with an entirely separate process. Any errors in the Python application can't propagate back to the facade or any of its consumers.

5.4 Reactivities

- *Implement a model-learning algorithm.* Machine learning textbooks often have detailed descriptions of various learning algorithms that you can use, in either pseudocode or some other language. For example, many decision-tree-based algorithms can be quite simple to implement. Once you have a rudimentary implementation in place, you can start to think about how reactive your implementation is or could be. Will your learning algorithm always complete within a given time? If not, how could you change that?
- *Dive deeper into building facades.* You built a simple facade for a machine learning algorithm in this chapter. In particular, you used a Python model-learning algorithm from an otherwise Scala codebase. That's a pretty common real-world scenario. Lots of development on machine learning technologies occurs using Python. One of the most powerful technologies you can use for building machine learning models is TensorFlow, from Google. Although large portions of it are written in C++, the primary user API is written in Python.

  Because TensorFlow is so popular, many people have attempted to implement different ways of calling it from Scala. Examine one of the several libraries that allow you to access TensorFlow from Scala. Some of these may be focused on Spark specifically. Notable examples include TensorFlowOnSpark from Yahoo!, TensorFrames from Databricks, and MLeap. When you look into these implementations, ask yourself what guarantees hold about the behavior of the model-learning implementation. How does the use of multilanguage runtimes change what you can say confidently about the behavior of a model-learning pipeline written using one of these tools? Can you write a test that proves something about the library's response to errors or high load?

If you're interested in learning more about TensorFlow specifically, check out *Machine Learning with TensorFlow* by Nishant Shukla (Manning, 2018) for a much deeper look into that technology (www.manning.com/books/machine-learning-with-tensorflow).

Summary

- A model is a program that can make predictions about the future.
- Model learning consists of processing features and returning a model.
- Model learning must be implemented with an expectation of failure modes (for example, timeouts).

- Containment, using the facade pattern, is a crucial technique for integrating third-party code.
- Contained code wrapped in a facade can be integrated with the rest of your data pipeline using standard reactive-programming techniques.

In the next chapter, we'll take the models you've learned and analyze their performance so you can decide whether to use them.

6 *Evaluating models*

This chapter covers

- Calculating model metrics
- Training versus testing data
- Recording model metrics as messages

We’re over halfway done with our exploration of the phases of a machine learning system (figure 6.1). In this chapter, we’ll consider how to evaluate models. In the context of a machine learning system, to *evaluate* a model means to consider its performance before making it available for use in predictions. In this chapter, we’re going to ask a lot of questions about models.

Much of the work of evaluating models may not sound that necessary. If you’re in a hurry to build a prototype system, you might try to get by with something quite crude. But there’s real value in understanding the output of the upstream

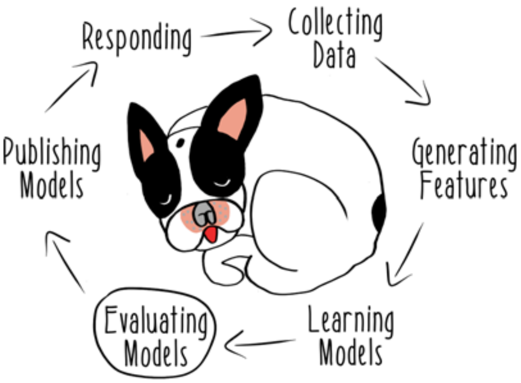


Figure 6.1 Phases of machine learning

components of your machine learning system before you use that output. The data in a machine learning system is intrinsically and pervasively uncertain.

When you contemplate whether you want to use a model to make predictions, you face that uncertainty head on. There are no answers to look up from somewhere else. You need to implement the components of your system such that they make the right decisions—or suffer the consequences.

Using machine-learned models can be high-stakes stuff. Machine learning systems are used to handle decisions of real consequence where failure could mean real losses to their users. We're going to consider just such a problem, one with some very concrete financial consequences for success or failure: fraud detection.

6.1 Detecting fraud

Kangaroo Kapital is the largest credit card company in Australia. Animals across the continent use Kangaroo Kapital credit cards to make all their daily purchases, racking up points in the company's reward system. Because Australian animals have traditionally not worn much clothing, the challenges of carrying around cash are substantial. Only having to keep track of a single credit card is a big help for your average working wallaby; nevertheless, Australian animals have problems keeping track of even a single credit card. Cards are often misplaced, leading to a problem with theft and fraudulent use.

The fraud team at Kangaroo Kapital is charged with detecting these cases of fraud. They use sophisticated analytical techniques to try to determine when a customer's card has been stolen. If they can determine with sufficient confidence that a card has been stolen, they lock down the card and contact the customer. The benefits of being fast *and* correct are substantial. In the best-case scenario, the fraud team's systems detect fraud the first time a card is misused, lock down the card, and then suffer no further losses (figure 6.2).

But if the system is too slow, the company eats the cost of those fraudulent transactions, not the customer. That can get expensive, because no one loves to spend quite like a duck-billed platypus with a stolen credit card (figure 6.3).

Being too eager can also be bad for the company. If the team is wrong, then they've locked down a card that a customer is attempting to use, greatly inconveniencing them.

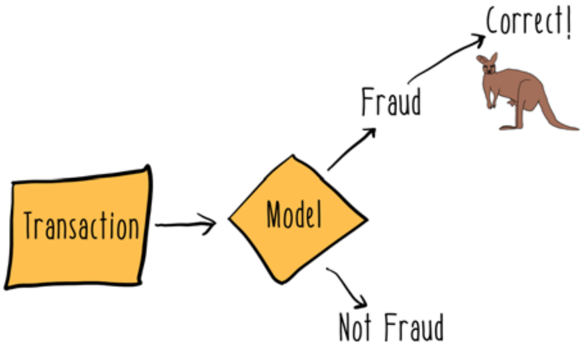


Figure 6.2 Successful fraud detection

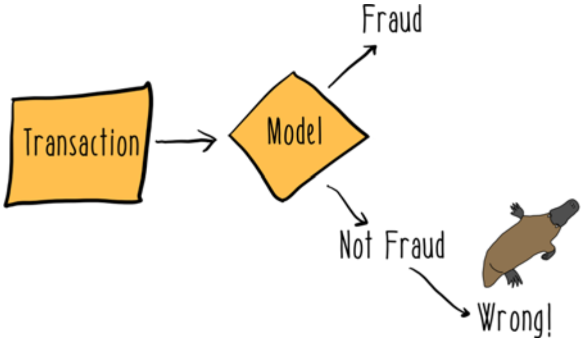


Figure 6.3 Failing to detect fraud

A Tasmanian devil who can't pay his dinner bill due to a declined credit card is one unhappy customer. He's likely to cancel his credit card, which would also lead to lost money for the company (figure 6.4).

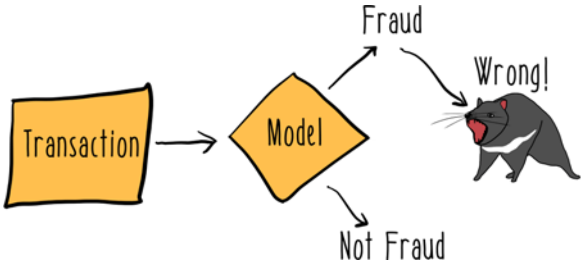


Figure 6.4 Inaccurately detecting fraud

This trade-off is often discussed in machine learning contexts using various metrics that can be calculated for a given model. Much of this chapter focuses on different model metrics, because metrics are a key part of how real-world machine learning systems are operated.

6.2 Holding out data

To understand model metrics for evaluating your models, you need to start by using some data that's different than you've ever used before. In previous chapters, you saw how to collect and use data to train a machine learning model. But you typically don't use all the data you've collected just for training the model. Instead, you usually leave some data aside for other purposes. That data is called *hold-out data*, meaning it wasn't included in the data used to learn the model. Depending on what you're trying to do with your machine learning system, you might do several different things with that held-out data, but usually you'll need it for something.

There are big dangers in deciding which data to hold out and which data to use to learn your model. If you're not careful about how you handle your data *at the system level*, you may find that your models will do terrible things when used to make live predictions in your production system. The 'roos at Kangaroo Kapital are a pretty conservative bunch, so they've tried to safely use hold-out data as much as possible. Their approach relies on a global concept of what is hold-out data and what is training data.

To understand the Kangaroo Kapital implementation, let's first set up a basic version of the domain model for the credit card–processing systems. The following listing shows some of the basics you'll need to build code that could handle credit card transactions.

Listing 6.1 Credit card transactions utils

```
type TransactionId = Long
type CustomerId = Long
type MerchantId = Long

case class Transaction(transactionId: TransactionId,
                       customerId: CustomerId,
                       merchantId: MerchantId,
                       dateTime: DateTime,
                       amount: BigDecimal)
```

Type alias for a transaction identifier

Type alias for a customer identifier

Type alias for a merchant identifier

Case class for a transaction

Type aliases

The identifiers of transactions, customers, and merchants are typed using *type aliases*. We haven't used them before, but type aliases are simple utilities. They allow you to define arbitrary numeric identifiers using meaningful names for their types, without changing any of the underlying properties of the base type. A `CustomerId` is just a `Long`. You can perform all the same operations on a `CustomerId` that you can on a `Long`. But you can implement your code to describe when a given `Long` is in fact the identifier for a customer versus the identifier for a merchant. Being able to assign these descriptive types is often helpful for building up richer descriptions of the problem domain using types, as shown in listing 6.1.

Using the basic domain model established in listing 6.1, you can now implement your version of the code to split out training data from hold-out data. The implementation applies a deterministic hashing function, based on a stable identifier (the customer's account number). All transactions for a given customer are always in either the training data or the held-out data. In less data-rich systems, that may not be the right approach. An alternative would be to split transactions between training data and held-out data. But Kangaroo Kapital has enormous market share in Australia, so splitting by customers is an acceptable choice for them.

The next listing shows how a given transaction is assigned to either the training data or the held-out data.

Listing 6.2 Assigning customers to the training set

```
val TrainingPercent = 80

def trainingCustomer(id: CustomerId): Boolean =
  val hashValue = id.hashCode() % 100
  hashValue < TrainingPercent
}

val sampleTransaction = Transaction(123, 456, 789,
➥ DateTime.now(), 42.01)

val datasetResult = trainingCustomer(sampleTransaction.customerId)

println(s"Is the sample transaction in the training set?
➥ $datasetResult")
```

Percentage of customers to assign to training sets

Function to determine whether a customer should be used in a training set

Uses the modulo 100 value of a hash value of a customer ID to produce a hash value

Compares the hash value for a customer with a constant percent

Sample transaction to use for testing

Resulting dataset for the sample transaction

Prints the results for inspection

The `trainingCustomer` function could be implemented in several different places. At the moment, you're concerned with transactions, so you could implement it on the `Transaction` class. Because it's information about a customer, you could put it on the `Customer` class that presumably exists somewhere (but that you haven't implemented). But `trainingCustomer` is a pure function that could be widely used across the system.

The type signature ensures that it will only ever operate on customer IDs, as you intend, so let's leave it on a utility object and allow consumers to import it as needed. If you have experience with object-oriented programming, that might strike you as bad style. But Scala unites both the object-oriented and functional programming paradigms, so this is entirely acceptable in Scala. In functional programming–style code, it's not uncommon to have "bags of functions," where an object may be a container for some functions that might all be used independently of each other. The object merely serves as a *namespace*—an identifier used for organizing code. This is in contrast to traditional object-oriented programming, which creates objects that are strongly cohesive units meant to be used as a whole.

λ Pure Functions

Because your hashing function holds to certain properties, there's no harm in allowing the function to be passed around throughout your codebase. It's clearly *pure*, meaning it causes no side effects. The function is also *referentially transparent*, meaning it will always return the same value when called with the same argument. That's an important property of mathematical functions that your functions must hold to. When you can structure your code as pure, referentially transparent functions, that can make code reuse quite easy and natural, as you can see with your hashing function.

Note that your hashing function is implemented in such a way as to randomly assign instances to training or testing according to the proportions set by the training-proportion parameter. This strategy for splitting up the training and testing datasets avoids various common data-preparation problems that can result in poorly performing models.

6.3 Model metrics

Now that you have the ability to divide up the data, you can use some of it to train your model and the rest to test or evaluate any learned models. You've already seen how models can be trained in chapter 5. Listing 6.3 recaps that model-learning process, again using Spark's MLlib. You'll start without using much new functionality from MLlib. Instead, you'll focus on how the model-learning process from chapter 5 connects to the work at hand. In this example, you'll learn a binary classification model using logistic regression.

Logistic regression

Although you haven't seen it before in this book, *logistic regression* is a common model-learning algorithm. It's a regression model used to predict categorical variables (for example, fraudulent versus nonfraudulent credit card charges). A deeper discussion of the details of the algorithm is beyond the scope of this book, but as usual, Wikipedia has a good introduction (https://en.wikipedia.org/wiki/Logistic_regression).

When it comes to building machine learning systems, logistic regression has several advantages: it's widely implemented, there are efficient distributed implementations, the model size scales linearly with the number of features, the importance of features on the model is easily analyzable, and so on. In this case, using logistic regression allows you to use even more library functionality from MLlib to evaluate your learned model than is available for less popular or more sophisticated model-learning algorithms.

Listing 6.3 Learning a model

```
val session = SparkSession.builder.appName("Fraud Model").getOrCreate()
import session.implicits._

val data = session.read.format("libsvm")
➥ .load("src/main/resources/sample_libsvm_data.txt")


val Array(trainingData, testingData) = data.randomSplit(Array(0.8, 0.2))

val learningAlgo = new LogisticRegression()

val model = learningAlgo.fit(trainingData)

println(s"Model coefficients: ${model.coefficients}
➥ Model intercept: ${model.intercept}")
```

Creates a new session

Imports some useful implicit conversions for use with DataFrames

Loads some sample data, stored in LIBSVM format

Randomly splits sample data into training and testing sets

Instantiates a new instance of a logistic-regression classifier

Learns a model over a training set

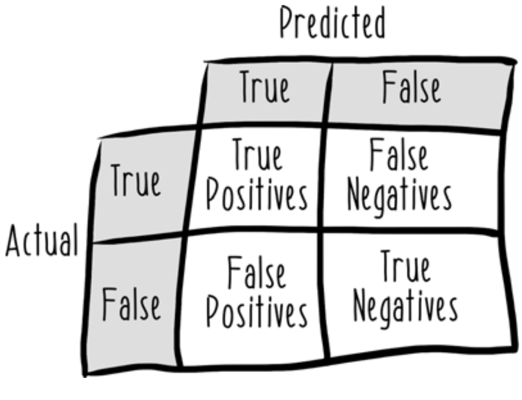
Prints the parameters of the model for inspection

In this case, you'll use some standard sample data to stand in for the Kangaroo Kapital credit card data. You can refactor this code later to ingest from your statically typed transactional data. With this sample data, you can get the basics of your training and testing process set up quickly. Note that you're also using a less sophisticated method of splitting the data between training and testing than you did before. Again, this is just to give you a simple but runnable prototype that you can refactor to use the credit card data later. Both the sample data and the random train/test splitting function are provided by the Spark project to make it easier to get started building models.

At the end of listing 6.3, you produced an instance of a `LogisticRegressionModel`. You can now use some library functionality to inspect and reason about your model. At this point in the process, you have absolutely no idea what your model is like. The outcome of the model-learning process is by definition uncertain—you could have a very useful model or complete garbage.

First, we can understand some of the metrics that can be computed about the model's performance on the training set, but to do that, we need to cover how to measure the performance of a classifier. Reviewing a bit from chapter 5, in binary classification problems, we often refer to the two classes as *positive* and *negative*. In the case of Kangaroo Kapital, the positive case would be that fraud had occurred, and the negative case would be that no fraud had occurred. A given classifier can then be scored on its performance within those classes. This is true whether the classifier is a machine-learned model, a dingo making decisions based on smell, or just a flip of an Australian one-dollar coin. Conventional terminology calls correct predictions *true* and incorrect predictions *false*. Putting all this together yields the two-by-two matrix shown in figure 6.5, known as a *confusion matrix*.

A *true positive* is when the model predicts a fraud correctly. A *false positive* is when the model predicts a fraud incorrectly. A *true negative* is when the model predicts a normal (not fraudulent) transaction correctly. Finally, a *false negative* is when the model incorrectly predicts a normal transaction, when there was in fact fraud.



		Predicted	
		True	False
Actual	True	True Positives	False Negatives
	False	False Positives	True Negatives

Figure 6.5 Confusion matrix

With these four statistics, we can calculate a number of statistics to help us evaluate models. First, we can evaluate the *precision* of a model, which is defined as the number of true positives divided by the sum of all positive predictions:

precision = true positives / (true positives + false positives)

Precision is important for the Kangaroo Kapital team. If their fraud model's precision isn't high enough, they'll spend all their fraud investigation budget investigating normal, nonfraudulent transactions.

There's another statistic, called *recall*, that's also important for the kangaroos. If the kangaroos' fraud model's recall isn't high enough, it will be too easy for animals to commit credit card fraud and never get caught, and that will get expensive.

Recall is defined as the number of true positives divided by the sum of all positives in the set:

recall = true positives / (true positives + false negatives)

Depending on the context, recall also goes by other names, such as the *true positive rate*. There's another statistic related to recall called the *false positive rate*, or *drop-out*, which is defined as the number of false positives divided by the sum of all negatives in the set:

false positive rate = false positives / (true negatives + false positives)

You can visualize how a model trades off the true-positive rate (recall) versus the false positive rate using a plot called an *ROC curve*.

> **NOTE** *ROC* stands for *receiver-operating characteristic*. The technique and the name originated in work on radar during World War II. Although the technique is still useful, the name has no relationship to its current common usage, so it's rarely referred to by anything other than its acronym, ROC.

A typical ROC curve plot might look something like figure 6.6.

The false positive rate is on the x-axis, and the true positive rate is on the y-axis. The diagonal line $x = y$ represents the expected performance of a random model, so a usable model's curve should be above that line. MLlib has some nice, built-in functionality to calculate the ROC curve for a binary classifier.

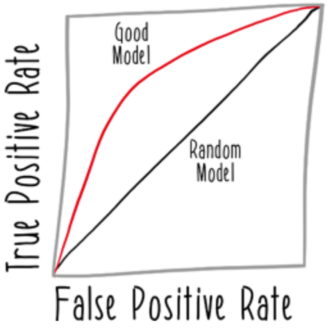


Figure 6.6 ROC curve

Listing 6.4 Training-performance summary

```
val trainingSummary = model.summary

val binarySummary = trainingSummary.asInstanceOf
➥ [BinaryLogisticRegressionSummary]

val roc = binarySummary.roc

roc.show()
```

Produces a summary of a model's performance

Casts that summary to an appropriate type, BinaryLogisticRegressionSummary

ROC curve for the model

Prints the ROC curve for inspection

The model summary is relatively new functionality in MLlib, so it's not available for all classes of models. There are also limitations to its implementation, such as the one that requires you to use `asInstanceOf` to cast the summary to the correct type. Make no mistake, using `asInstanceOf` like this is bad Scala style; it represents a subversion of the type system. But MLlib is still being rapidly developed, so this cast operation is just a sign of an incomplete implementation within MLlib. Development on MLlib is very active, but machine learning is an enormous domain for any one library to support. New functionality is being added at a rapid pace, and the overarching abstractions are being dramatically improved. Look for rough edges like this class cast to disappear in future versions of Spark.

We're building massively scalable machine learning systems that operate largely autonomously, so who has time to look at a graph and make a decision about what constitutes a good-enough model? Well, one of the uses of an ROC curve is to get a single number about the performance of a model: the area under the ROC curve. The higher this number, the better the model's performance. You can even make strong assertions about a model's utility using this calculation. Remember, a random model would be expected to perform according to the line $x = y$ on the ROC curve. The area under that is 0.5, so any model with an area under the curve (AUC) of less than 0.5 can safely be discarded as being worse than a random model.

Figures 6.7, 6.8, and 6.9 show the differences in the area under the curve of a good, random, and worse-than-random model.

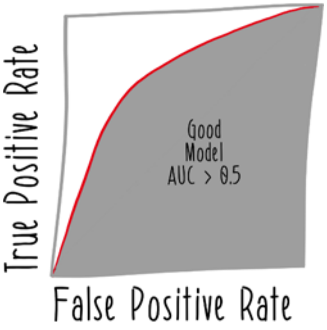


Figure 6.7 Good model

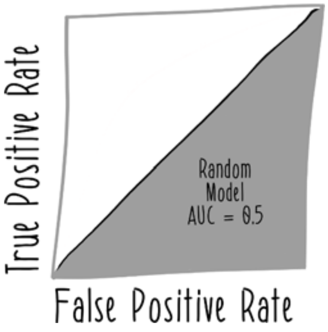


Figure 6.8 Random model

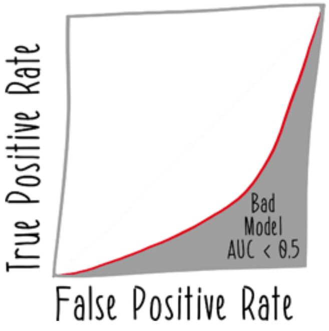


Figure 6.9 Bad model

The next listing shows the implementation of validating for performance better than random.

Listing 6.5 Validating training performance

```
def betterThanRandom(model: LogisticRegressionModel) = {
  val trainingSummary = model.summary

  val binarySummary = trainingSummary.asInstanceOf
  ➥ [BinaryLogisticRegressionSummary]

  val auc = binarySummary.areaUnderROC

  auc > 0.5
}

betterThanRandom(model)
```

Defines a function to validate that a model is better than random

Training summary

Class casting

Area under the ROC curve

Tests whether the area under the curve is greater than the random model

Example call to validate a model

This validation can serve as a useful safety feature in a machine learning system, preventing you from publishing a model that could be deeply detrimental. In the Kangaroo Kapital example, because fraud is so much rarer than normal transactions, a model that failed this test would very likely be falsely accusing a lot of angry animals of credit card fraud.

This technique can be extended beyond basic sanity checks like this. If you record the historical performance of your published models, you can compare the performance of your newly trained models to them. Then a logical validation would be to not publish a model with meaningfully different performance than the current published model. I'll discuss some more techniques for model validation a bit later.

You're not done asking questions about your model yet. You can consider other model metrics. The metrics you've seen so far try to capture an aspect of a model's performance. In particular, it's not hard to imagine a model that does a bit better on precision but not on recall or vice versa. An *F measure* (or sometimes *F1 score*) is a statistic that tries to combine the concerns of precision and recall in the same metric. Specifically, it's the harmonic mean of the precision and the recall. The next listing shows two ways of formulating the F measure.

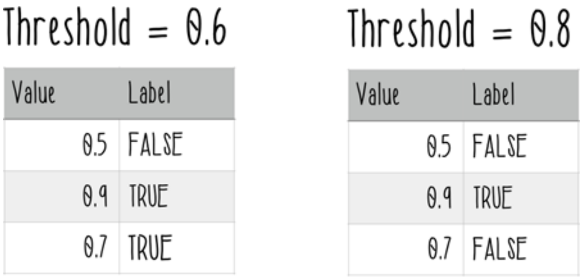
Listing 6.6 F measure

```
F measure = 2 * (precision * recall) / (precision + recall)

F measure = (2 * true positives) /
➥ (2 * true positives + false positives + false negatives)
```

Using the F measure as a model metric may not always be appropriate. It trades off precision versus recall evenly, which may not correspond to the modeling and business objectives of the situation. But it does have the advantage of being a single number that can be used to implement automated decision making.

For example, one use of the F measure is to set the *threshold* that a logistic-regression model uses for binary classification. Internally, a logistic-regression model is producing probabilities. To turn them into predicted class labels, you need to set a threshold to divide positive (fraud) predictions from negative (not fraud) predictions. Figure 6.10 shows some example prediction values from a logistic-regression model and how they could be divided into positive and negative predictions using different threshold values.



Threshold = 0.6

Value	Label
0.5	FALSE
0.9	TRUE
0.7	TRUE

Threshold = 0.8

Value	Label
0.5	FALSE
0.9	TRUE
0.7	FALSE

Figure 6.10 Threshold setting

The F measure isn't the only way of setting a threshold, but it's a useful one, so let's see how to do it. The following listing shows how to set a threshold using the F measure of the model on the training set.

Listing 6.7 Setting a threshold using the F measure

```
val fMeasure = binarySummary.fMeasureByThreshold

val maxFMeasure = fMeasure.select(max("F-Measure"))
➥ .head().getDouble(0)

val bestThreshold = fMeasure.where($"F-Measure" === maxFMeasure)
  .select("threshold").head().getDouble(0)

model.setThreshold(bestThreshold)
```

Finds the maximum F measure

Retrieves the F measure for every possible threshold

Finds the threshold corresponding to the maximum F measure

Sets that threshold on the model

Now the learned model will use the threshold selected on the basis of the F measure to distinguish between positive and negative predictions.

6.4 Testing models

Back in listing 6.3, as part of preparing your data for learning, you set aside some of the data for the testing process. Now it's time to use that hold-out data to test your model. When we test models, our goal is to get an accurate picture of the model's performance in the wild. To do that, we must use data that the model has never seen before: our hold-out data.

You can use the existing model that you've already learned and set a threshold to make predictions on the hold-out data. The next listing shows how to produce predictions and inspect them.

Listing 6.8 Predicting on hold-out data

```
val predictions = model.transform(testingData)

predictions.show(5)
```

Predicts over each row in a testing dataset

Prints a few predictions for inspection

Next, you'll do something similar to what you did before: calculate some metrics about the model's performance. In this case, let's look at precision and recall again. To recap: a model with low precision resulted in a lot of angry Tasmanian devils who got their cards declined for totally normal, nonfraudulent usage. A model with low recall will result in a lot of happy platypi swimming away with all of Kangaroo Kapital's money, because their fraud went undetected. Both are important, so we want a model that trades off both concerns well. To visualize how a model does with respect to precision and recall, we can look at another plot: the precision-recall curve, shown in figure 6.11.

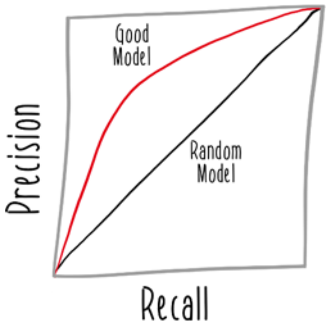


Figure 6.11 Precision-recall curve

Compared to the ROC curve we looked at before, the only difference is the metrics on the axes. Recall is on the x-axis, and precision is on the y-axis. Again, the diagonal line $x = y$ represents the expected performance of a random model, so a usable model's curve should be above that line.

As before, you want to be sure that the learned model is better than a random model for your chosen model metrics, so you'll calculate the area under the precision-recall curve.

Listing 6.9 Using an evaluator

```
val evaluator = new BinaryClassificationEvaluator()
  .setLabelCol("label")
  .setRawPredictionCol("rawPrediction")
  .setMetricName("areaUnderPR")

val areaUnderPR = evaluator.evaluate(predictions)
```

Instantiates a new evaluator

Sets a column containing a class label

Sets a column containing predictions

Sets a metric to be calculated, the area under the precision-recall curve

Executes the evaluator

The area under the precision-recall curve for a random model is 0.5, the same as a random model in an ROC curve, so you can define your validation function in the same way.

Listing 6.10 Validating the model

```
def betterThanRandom(area: Double) = {
  area > 0.5
}
```

Defines a function to check the area under the precision-recall curve

Ensures that the area under the curve is greater than the random model

From the perspective of the code you had to write, the metrics on the hold-out data worked pretty much identically to calculating metrics on the training data. But it's important to note that metrics calculated on training data give you a very different sort of picture than metrics calculated on hold-out data. The metrics you calculated on the training set represent *the best possible* performance of the model. During the training process, the model-learning algorithm had access to both the features and the class labels. Depending on the model-learning algorithm and the training dataset, it's possible for a model to have 100% accuracy on a training dataset.

The hold-out data gives you a very different and more realistic view of how good your model is. Because the model has never seen any of this data, it should behave much the same as it would if you published it and used it in the real world. You can think of this as early access to the model's true performance, much like the Manning Early Access Program (www.manning.com/meap-program), which gives readers access to books like this one before they're actually printed on paper.

This early access to a model's performance is crucial. It allows you to protect your production systems from fundamentally broken models that could wreak havoc with your overall system. It's important that you don't corrupt the integrity of the testing process. If you fail to adequately separate the data you use for training and testing processes, you can get a fundamentally inaccurate picture of your model's performance. Otherwise, you can end up with the problem discussed in the next section.

6.5 Data leakage

A common data-handling error is called *data leakage*. It works something like this. You've separated your data for use in training and testing, but there's a subtle problem. Knowledge about what's in the hold-out data—in particular, the class labels—has *leaked* into the training data somehow. The result will be good performance on the training and testing data but potentially very bad performance on real-world data.

In the case of Kangaroo Kapital, consider a data scientist building a model on the transactions of long-term customers, for detecting fraud. Because the scientist has a lot of historical data on these customers, he decides to build a feature about the history of fraud for a certain user. The rationale is that past incidences of fraud on a customer's account imply that she might not be very good at keeping her card in her pouch. The data scientist writes a feature to query the historical number of fraud reports for a customer, which looks something like the stub implementation in the following listing.

Listing 6.11 Past-fraud-reports feature

```
def pastFraudReports(customer: Customer): Int = ???
```

The problem is that in the data scientist's implementation, he has no date-range restriction on his query. The backing database that stores the fraud-report data that he's querying employs a mutable data model. The result is that recently reported frauds are included in this feature, so the model can see those frauds and bias itself toward fraud in the training process. This approach continues to work just fine in the test set, where that feature about "past" frauds continues to do a good job of predicting where "future" frauds will occur. All the model metrics we've looked at will appear to indicate that this is a highly performant model—but they'll be wrong. Once the model is published, its performance will be much lower than any previously calculated metrics would have implied. That's because the model can no longer see the future data in the feature, so it can no longer rely on that feature to artificially inflate its performance.

Data leakage can also take even more subtle forms than that. Remember, Kangaroo Kapital separates customers into either training or testing customers. That's a generally sound strategy, but it's not necessarily all that's needed to ensure that data is handled properly.

There was an incident a while ago involving a ring of koala fraudsters. They slowly accumulated the account credentials of many of Kangaroo Kapital's customers. Then, all at once, they rang up a huge amount of charges, costing the company millions. That's a lot of eucalyptus!

The problem with working with this historical dataset is that it happened at a particular point in time, for a particular subset of users. For some reason, the koala fraudsters targeted dingoes primarily. When the data is separated between training and testing usage by customer, plenty of dingoes are going to wind up in the training set, and it's probable that the model will learn that accounts held by dingoes are likely to be targeted for fraud. The problem is that this knowledge is useless for the future. It was a single incident, at a single point in time. All the perpetrators have since been locked up (after a low-speed police chase). This knowledge is useless, and a dingo-centric model will now perform quite poorly in the wild.

In cases of large events in the underlying dataset like this, you can use various techniques to mitigate the impact of this anomalous data. One approach is to divide your data by time, using the earlier data for training, while holding out the later data. This approach may result in quite poor performance during the testing phase, but that will be accurate. Machine learning models have a hard time predicting the unpredictable.

Or you can discard this time period entirely on the grounds that you're trying to build a model for the concept label of "normal-scale fraud," which would typically be more evenly distributed across various species of customers. That doesn't have to mean ignoring such forms of fraud entirely. You can still implement other types of models or even deterministic systems (for example, rate limits) to detect major fraud events like this. This approach lets your "normal-scale fraud" focus on the small and regular incidents of fraud that you were originally focusing on anyway.

Regardless of the approaches you take, you'll always want to make sure that any relevant knowledge from the hold-out data is kept *entirely* out of the training data. If you can succeed at that, your model-evaluation process should be accurate and useful.

6.6 Recording provenance

Now that you've asked all sorts of questions about your models to ensure that they're predictive, you need to put those models to use. In chapter 7, we'll focus on the process of publishing models, making them available for use in predictions. But before we do that, let's take a quick look at how to capture all the useful information that we considered in this chapter.

Uncertain Data

If you look back at chapter 4, section 4.5, you can see that you needed to make decisions about which feature data could be used based on contextual information that couldn't necessarily be established within source code and verified at compile time. We have a similar problem with models. Due to the intrinsic uncertainty of the model-learning process, we don't know how good our model is going to turn out to be until after we've learned it and evaluated its results.

This chapter has shown several techniques for addressing this uncertainty through using statistical techniques to evaluate the performance of models. The results of those calculations need to be consumed by something, though. In this book, we'll consider two downstream consumers of the data produced during model evaluation: the publishing process and the supervisory component of the prediction-serving application.

The information that these other systems need to consider is a form of *provenance* (also known as *lineage*). In this context, the provenance of a model is the information about how the model was produced, including things like the performance metrics used to determine that a model should be published.

One way to pass around this information is to attach the calculated metrics to the model itself inside a wrapper object of some kind. The next listing shows one way to do this for some of the statistics you've calculated in this chapter.

Listing 6.12 Evaluation-results case class

```
case class Results(model: LogisticRegressionModel,
                   evaluatedTime: DateTime,
                   areaUnderTrainingROC: Double,
                   areaUnderTestingPR: Double)
```

The model itself

Time at which the model was evaluated

Area under the ROC curve on the training set

Area under the precision-recall curve on the testing set

These may not be precisely the metrics you want to record for your particular model-learning pipeline. Area under the curve is useful for deciding *if* we want to use a model, but downstream systems may be more concerned with *how* that model will behave. You might want to record the precision and recall on the testing set at the chosen threshold.

Additionally, you may not be passing around the model object with its metadata. Sometimes you may choose to only refer to a model by a unique identifier and store the metadata in a different place than the model, such as in a database. The next listing shows an alternative way of modeling your evaluation results.

Listing 6.13 Refactored evaluation-results case class

```
case class ResultsAlternate(modelId: Long,
                            evaluatedTime: DateTime,
                            precision: Double,
                            recall: Double)
```

Model identifier

Time at which the model was evaluated

Precision of the model on the testing set

Recall of the model on the testing set

This alternative approach is a bit closer to the message-oriented ideal preferred in a reactive machine learning system. This case class could easily be transmitted as a message via any number of technologies: queue, event bus, database, and so forth. There's no need for any of the downstream systems needing to consume this data to be coupled any more tightly than simple message-passing. From the perspective of any consumers (receivers of `ResultsAlternate` messages), the model could have been learned by any library, in any language.

If our model-evaluation process turned up that we had learned a useless model, we could send that information as a message about a model with low precision and/or recall. This message-passing form of communication keeps the failure of our model-learning process nicely contained. As you'll see in the next chapter, we can build model-publishing systems that can act on messages about models and do the right thing to maintain reactivity in our machine learning systems.

6.7 Reactivities

- *Calculate a different performance statistic for the performance of a given model on a dataset.* Believe it or not, there are more performance statistics that you can calculate. You can try calculating the positive and negative likelihood ratios, the G-measure, or something else. You'll find descriptions of different statistics in various statistics references, in books and online. Then you can compare those values to the results of the other calculations we've explored in this chapter. What intuitions does this give you about the performance of the model?
- *Try to build a "perfect" model on the training set.* This one is easier than it sounds. With a little work, many techniques can get perfect or near-perfect performance for a given dataset. Generally speaking, if your model has the same number or more of parameters as it does training instances, it can be possible for each parameter to end up effectively being a representation of a given instance. If you give every instance in your dataset an arbitrary identifier as a feature, then several model-learning techniques can use this single feature as a way of "remembering" which instance has which class label. If you get a model that's been trained in this way, the interesting thing to do with it is to test it. Apply the model to your test set, and calculate its performance. Because you've engineered an overfit model, the performance is likely (but not guaranteed) to be pretty terrible. How does your model perform on the test set? If its performance isn't too bad, how else could you detect that it might not do well on future unseen data?

Summary

- Models can be evaluated over hold-out data to assess their performance.
- Statistics like accuracy, precision, recall, F-measure, and area under the curve can quantify model performance.

- Failing to separate data used in training from testing can result in models that lack predictive capability.
- Recording the provenance of models allows you to pass messages to other systems about their performance.

In the next chapter, you'll see how to make your learned models available for use in predictions.

7 Publishing models

This chapter covers

- Persisting learned models
- Modeling microservices using Akka HTTP
- Containerization of services using Docker

In this chapter, we'll consider how to publish models (see figure 7.1). Throughout this book, you've been learning and using models, but making models available for use in a real machine learning system can involve some complexities that you haven't yet seen. When you're exploring models in a REPL like the Spark shell, you can directly call methods on the instance of a model already in memory. But in real-world systems, it's common for a model to be learned in a pipeline, as you saw in chapters 4 and 5, before being used in a completely different application. This chapter will show you how to make models

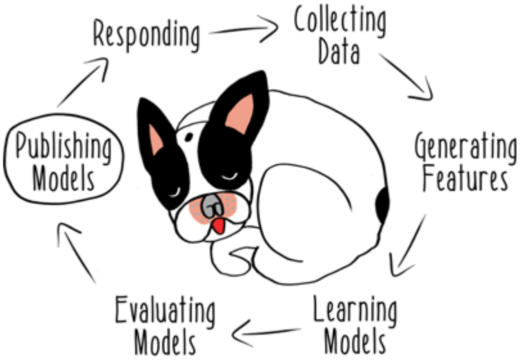


Figure 7.1 Phases of machine learning

available for use in the complex environment of a real-world machine learning system. We'll work through an approach to packaging models into services and then making those services into independently deployable units.

7.1 The uncertainty of farming

Machine learning is used in all sorts of industries, not just ones that you think of as having a lot of technology involved. The world of farming, for example, requires a great deal of technological sophistication. Consider Hareloom Farms, an organic farm, run entirely by rabbits. The rabbits grow fruits and vegetables, including celery, tomatoes, and, of course, carrots.

The business of farming is fraught with uncertainty. At Hareloom, an early freeze could destroy their tomatoes. Producing less kale than there is demand for could mean missing out on potential revenue. A sudden drop in the price of turnips could leave them with a crop that's barely worth harvesting.

For all these reasons, Hareloom Farms needs predictive capabilities to run their mostly nontechnology business. They have a data team consisting of data scientists and engineers (including you) who are smarter than the average hare. You build machine-learned models for all these problems using tools that you've seen before: Scala, Spark, and MLlib. Let's see what you cooked up down on the farm.

7.2 Persisting models

The farmers at Hareloom are very concerned with their crop *yields*—the amount of crops produced per unit of farmland. Lately, they've been trying to model the problem of how much carrot seed to use during planting. Carrot seed itself is a pretty low-cost ingredient for their operation, so they historically used it pretty freely. When they used too little carrot seed, they didn't produce as many carrots as they'd like. But when they used too much, they saw issues with crowding, leading to unsatisfactorily small carrots.

Your team recorded all of this historical data in the form of simple instances for training. For your single feature, you chose the number of seeds per inch of soil planted. For your concept label, you chose to use a single Boolean value to indicate whether that particular harvest was considered a success. This success label is manually produced by aggregating subjective judgments around the amount of carrots harvested, the size of the carrots, and the difficulty of harvesting them.

With the modeling task defined, you can start to build up your Spark pipeline.

Listing 7.1 Loading data

```
val session = SparkSession.builder
  ➥ .appName("ModelPersistence").getOrCreate()

val data = Seq(
  (0, 18.0, 0),
  (1, 20.0, 0),
  (2, 8.0, 1),
  (3, 5.0, 1),
  (4, 2.0, 0),
  (5, 21.0, 0),
  (6, 7.0, 1),
  (7, 18.0, 0),
  (8, 3.0, 1),
  (9, 22.0, 0),
  (10, 8.0, 1),
  (11, 2.0, 0),
  (12, 5.0, 1),
  (13, 4.0, 1),
  (14, 1.0, 0),
  (15, 11.0, 0),
  (16, 7.0, 1),
  (17, 15.0, 0),
  (18, 3.0, 1),
  (19, 20.0, 0))

val instances = session.createDataFrame(data)
  .toDF("id", "seeds", "label")
```

Sets up a SparkSession

Sequence of instances for training and evaluation

Instances consist of an identifier, seeds used per inch, and a binary label for the success of the harvest.

Creates a DataFrame from instances

Names columns in the DataFrame

The rabbits at Hareloom have more historical data than that, but this sample should be enough to get you started implementing your model. Note that you're going to use the DataFrame-centric Spark ML API again.

Once you have data loaded, you need to produce features from that data. In this case, the feature transform you'll apply is the binning technique from chapter 4, this time using some MLlib library functionality, to reduce your feature values to three bins, as shown in the next listing. For more on the technique of binning features, see section 4.3.

Listing 7.2 Preparing the features

```
val discretizer = new QuantileDiscretizer()
  .setInputCol("seeds")
  .setOutputCol("discretized")
  .setNumBuckets(3)

val assembler = new VectorAssembler()
  .setInputCols(Array("discretized"))
  .setOutputCol("features")
```

Sets up a QuantileDiscretizer for use in feature engineering

Takes as input the seed-density data

Sets an output column for the discretized data

Tells the discretizer to use three buckets

Sets up a VectorAssembler to format data for use as features

Sets input columns to be formatted

Defines an output column

The QuantileDiscretizer performs the binning (or discretization) operation for you without requiring predefined boundaries between buckets. Instead, you specify the number of buckets, and the discretizer infers reasonable buckets by sampling the data.

Following the discretizer, you also called another bit of helper functionality from Spark, the VectorAssembler. Its purpose is to add a new column to your DataFrame containing the feature values wrapped in the necessary Vector type expected by other parts of the ML pipeline functionality.

Following that, you can now compose the remainder of your learning pipeline. In this example, you'll be using a technique called *cross validation* to explore multiple models to determine which performs best. Cross validation is a technique based on dividing the data into random subsamples so that the results of the model-learning process can be evaluated on different portions of the data. This process can then be repeated with different hyperparameters.

You'll need to set up an object to hold the parameters that will be used in different executions of the model-learning process. Then you can choose which parameters produce the best model, based on the performance of that model. The problem of finding the most effective parameters for the model-learning process is generally referred to as *hyperparameter optimization*. The technique you use in the example code is known as *grid search*, for the parameter grid being iterated through. Unlike some other applications of search that you may have seen, the form of search used here is a simple *exhaustive search*, meaning your pipeline will try all the parameters you provide to it. Although there are more sophisticated approaches to hyperparameter optimization that you could implement, the simple exhaustive search of the parameter grid is conveniently provided by MLlib and is effective for parameter grids of small size.

The following listing shows how these concepts come together in the remainder of your pipeline implementation.

Listing 7.3 Composing the pipeline

```
val classifier = new LogisticRegression()
  .setMaxIter(5)

val pipeline = new Pipeline()
  .setStages(Array(discretizer, assembler, classifier))

val paramMaps = new ParamGridBuilder()
  .addGrid(classifier.regParam, Array(0.0, 0.1))
  .build()

val evaluator = new BinaryClassificationEvaluator()

val crossValidator = new CrossValidator()
  .setEstimator(pipeline)
```

Instantiates a classifier to learn a logistic regression model

Limits the number of iterations it should use during model learning

Instantiates a pipeline

Sets stages of the pipeline

Adds regularization parameters

Instantiates a ParamGridBuilder to set some parameters

Builds a map of parameters

Sets up a BinaryClassificationEvaluator for evaluating learned models

Sets up a CrossValidator to evaluate different models

Sets an Estimator to use: previously instantiated pipeline

```
    .setEvaluator(evaluator)
    .setNumFolds(2)
    .setEstimatorParamMaps(paramMaps)
```

Sets the number of folds to use in cross validation

Sets an Evaluator to use; previously instantiated BinaryClassificationEvaluator

Uses previously defined regularization parameters

The preceding example performs cross validation over different models using different regularization parameters. *Regularization* is a technique designed to produce simpler models that are more generally applicable, as shown in the next listing.

Listing 7.4 Learning and saving the model

```
val model = crossValidator.fit(instances)

model.save("my-model")
```

Learns the model, executing all steps in the pipeline

Persists the learned model to the directory named my-model

The final save operation in this pipeline relies on new functionality added in Spark 2.0. It makes saving and reusing models very easy. In previous chapters, you had to define case classes for the output of your work. These were often structured as pure data and were often intended to be used as immutable messages. The model-persistence functionality of Spark does that for you in a single save call.

To see exactly what it's doing, poke around the my-model directory you created. What you'll find are two types of files. The first is a metadata file in JSON format. You first saw the JSON format in chapter 3, where you used it as a structure to translate your Scala case classes into, so that they could be persisted in your Couchbase document database. Here, JSON is used to record the metadata about various aspects of your pipeline. For example, you can open the my-model/bestModel/metadata directory and find a file like part-00000. It should contain something like the model in the following listing.

Listing 7.5 Pipeline metadata

When the model was produced

```
{
    "class": "org.apache.spark.ml.PipelineModel",
    "timestamp": 1467653845388,
    "sparkVersion": "2.0.0-preview",
    "uid": "pipeline_6b4fb08e8fb0",
    "paramMap": {
      "stageUids": ["quantileDiscretizer_d3996173db25",
          "vecAssembler_2bdcd79fe1cf",
          "logreg_9d559ca7e208"]
    }
}
```

Type of model being persisted: a PipelineModel

Unique identifier for this model

Version of Spark it was produced with

Parameters about the pipeline

Unique identifiers for each phase in the pipeline

After looking at this metadata, you can examine the my-model/bestModel/stages directory, where you should find directories corresponding to the identifier for each stage, as listed in your metadata file. If you look at the one for the logistic-regression phase, you should see something like the next listing.

Listing 7.6 Logistic-regression model metadata

```
{
    "class": "org.apache.spark.ml.classification
    ➥ .LogisticRegressionModel",
    "timestamp": 1467653845650,
    "sparkVersion": "2.0.0-preview",
    "uid": "logreg_9d559ca7e208",
    "paramMap": {
        "threshold": 0.5,
        "elasticNetParam": 0.0,
        "fitIntercept": true,
        "tol": 1.0E-6,
        "regParam": 0.0,
        "maxIter": 5,
        "standardization": true,
        "featuresCol": "features",
        "rawPredictionCol": "rawPrediction",
        "predictionCol": "prediction",
        "probabilityCol": "probability",
        "labelCol": "label"
    }
}
```

Type of predictive model being persisted: a LogisticRegressionModel

Parameters about the model

Threshold of the model

Various columns from the DataFrame used to produce the model

Similar files are produced for the other stages of the pipeline.

The use of a human-readable format like JSON makes exploring this data easier, but that's not the primary purpose of these files, which is to allow you to load previously learned models that have been saved. In fact, these metadata files aren't the models. That's where the other type of files produced comes in.

The other files use the Parquet format. Apache Parquet started as a joint project between engineers at Cloudera and Twitter. It's commonly used in big data projects that use Hadoop or Spark because it's an efficient method of serializing data, with wide support. Data stored in the Parquet format can very easily be used across different sorts of data-processing systems. In this case, you use it to store any of the data around the phases in your modeling pipeline. For a simple logistic-regression model without many features, like the one you built, the amount of data in a model is pretty modest, but the efficient compression capabilities of Parquet could be more useful for larger models requiring the persistence of more parameters.

The point of all of this was to allow you to load a model after you had persisted it, as shown in the following listing.

Listing 7.7 Loading a persisted model

```
val persistedModel = CrossValidatorModel.load("./my-model")
```

In this example, that's all it takes to restore the previously learned model from the files you inspected. In a full system implementation, this will allow the rabbits at Hareloom

Farms to learn a model using a distributed model-learning pipeline running on a cluster of nodes and then later use that model in a predictive microservice, such as the kind discussed in the next chapter.

7.3 Serving models

Now that you've seen how to persist and load models, let's look at how you can use models to make predictions. The component of a machine learning system that you need to build now is called by various names, such as *model server* or *predictive service*, depending on the design of the component. Typically, model servers are applications that can make predictions using a *model library*, a collection of previously learned models. In contrast, a predictive service only serves predictions for a single model. However it's designed, the model-serving component is crucial to having a useful machine learning system. Without this component, the rabbits would have no way of using their models to make predictions on new problems.

The Hareloom data-science stack has evolved using different technologies and designs. In the original model server, all the models that had ever been learned were stored in and served from a single server-side application. That approach had a number of problems: it was inflexible, changes in the model structure often meant changes to the model server, and it required that a lot of infrastructure be built around the management of the library of models.

7.3.1 Microservices

Eventually, the team decides to abandon their old model-serving implementation when their load becomes too high and predictions can't be quickly retrieved at peak scale. In the redesign, they decide to try a different approach using some new tools and techniques. Instead of having a monolithic server containing all their models, they choose to create a microservice per model. A *microservice* is an application that has very limited responsibilities. At the most extreme end, a microservice might do only one thing, holding to the *single responsibility principle*. When it was originally developed, the single responsibility principle was meant to apply to a small component like a class or a function. The principle was that a given system component should do only one thing. When you extend this principle to system design, you can create a service that does only one thing.

In the context of machine learning, the one thing you want your microservice to do is expose the model function. The team likes the simplicity of this "model function as a service" design because each microservice can focus on the specific needs of a given type of model, instead of having one application handle all types of models.

Choosing to break up their library of models into separate microservices has solved some of their performance problems. If a given predictive service isn't keeping up with its load, the team can deploy another instance. The pure functions of the team's models are stateless, so you can have many of them running at once without any need to coordinate among instances. Later, I'll discuss some of the infrastructural pieces that may help when working with arrays of services.

7.3.2 Akka HTTP

Even given those advantages, the team wants to see if they can come up with a highly performant implementation of a model microservice. They turn to our old friend Akka to help construct the wrapping infrastructure around their models. Akka HTTP is the module in Akka that's focused on building web services. It's not a web application framework like Play. Instead, it's more useful for building things like services that expose APIs over HTTP. It evolved from a previous framework built on top of Akka with a lot of the same goals, called Spray. Note that some parts of the functionality you'll use in this chapter come from the older Spray project. The advantage of using Akka to build the predictive service is that you can take advantage of the powerful concurrency capabilities of Akka. Akka's actor model provides substantial capability for efficiently using hardware to build performant services. For example, you can use many more actors to model concurrency in an application than you can use threads. That means you can serve many prediction requests at the same time, with the Akka framework doing most of the hard work of handling all the concurrent requests.

Let's begin by setting up the schema of what your predictions will look like, in a case class, as usual. This will also be used to define how your predictions can be serialized as JSON and deserialized back into case classes.

Listing 7.8 Predictions data

Uses JSON formatting functionality from Spray JSON

Case class for predictions

```
case class Prediction(id: Long, timestamp: Long, value: Double)

trait Protocols extends DefaultJsonProtocol {
  implicit val predictionFormat = jsonFormat3(Prediction)
}
```

Defines an implicit formatter for the prediction case class using the Spray JSON helper function

Then you can set up the predictive functionality itself. In this example, you'll use a dummy model and a simplified representation of features. The string representation of the features is parsed from the API call and turned into a feature vector implemented as a `Map`.

Listing 7.9 Models and features

Defines a dummy model that operates on features structured as maps of characters to doubles

Creates a map of feature identifiers to "coefficients"

Generates a prediction value by operating on all feature values

```
def model(features: Map[Char, Double]) = {
  val coefficients = ('a' to 'z').zip(1 to 26).toMap
  val predictionValue = features.map {
    case (identifier, value) =>
      coefficients.getOrElse(identifier, 0) * value
  }.sum / features.size
```

Retrieves a model coefficient for a feature, multiplying it by the feature value

Uses a case statement to bind names to each feature identifier and feature value

Sums feature values and divides by the number of features to produce an average prediction value

```
    Prediction(Random.nextLong(), System.currentTimeMillis(),
    ➥ predictionValue)
  }

  def parseFeatures(features: String): Map[String, Double] = {
    features.parseJson.convertTo[Map[Char, Double]]
  }

  def predict(features: String): Prediction = {
    model(parseFeatures(features))
  }
```

Instantiates, returning a prediction instance

Defines a function to convert a feature passed as strings into maps of strings to doubles

Converts the feature strings, using a parser and a converter from Spray JSON

Defines a prediction function

Composes a model and feature parser to produce predictions

Now you can wrap these functions in a service definition. Using Akka HTTP, the following listing defines an API route named `predict` that will take serialized string representations of features and return the predictions by your model.

Listing 7.10 A service with routes

```
  trait Service extends Protocols {
    implicit val system: ActorSystem
    implicit def executor: ExecutionContextExecutor
    implicit val materializer: Materializer

    val logger: LoggingAdapter

    val routes = {
      logRequestResult("model-service") {
        pathPrefix("predict") {
          (get & path(Segment)) {
            features: String =>
              complete {
                ToResponseMarshallable(predict(features))
              }
          }
        }
      }
    }
  }
```

Creates a Service trait that contains your formatting functionality

Requires an implicit actor system for Akka

Requires an implicit ExecutionContextExecutor for Akka

Requires an implicit Materializer for Akka

Requires a logger

Defines routes of service

Logs each request and result to a service

Defines the prefix of your route to be predict

Defines that this path will receive gets only

Accepts features as a String of JSON

Defines how to complete a request

Calls a prediction function and turns it into a response

Finally, the next listing instantiates an instance of that service, starts up logging, and binds your service with its `predict` route to a given port on your local machine.

Listing 7.11 A model service

```
  object ModelService extends App with Service {
    override implicit val system = ActorSystem()
    override implicit val executor = system.dispatcher
```

Defines a model service as a runnable App using the Service trait you defined

Instantiates an actor system for Akka

Instantiates an executor for Akka

```
    override implicit val materializer = ActorMaterializer()

    override val logger = Logging(system, getClass)

    Http().bindAndHandle(routes, "0.0.0.0", 9000)
}
```

Instantiates a materializer for Akka

Instantiates a logger

Starts a new HTTP server with defined routes at a given IP and port

This service can now accept requests for predictions on features and return the model's predictions, from any program on a network that can produce a well-formed GET containing the necessary features (for example, a web browser, mobile app, and so on). That will make it a lot easier for your team at Hareloom to use their predictive models in other applications, even beyond the ones maintained by the data team. All they have to do is get this microservice running on a server somewhere.

7.4 Containerizing applications

The JVM ecosystem has an approach to building and distributing runnable applications. Applications can be built as JARs—archives of code—and then executed anywhere that has a Java runtime. Scala inherited all these capabilities from Java, and they work just fine.

But there are other options. Lately, many teams have been packaging and distributing applications in a way that maximizes portability and works for all types of runtimes, called containers. *Containers* are a way of virtualizing an entire server such that the resulting container can be run on top of another OS while still appearing internally as if it were directly operating on a server, without an intervening host. A container can guarantee a complete static snapshot of the state of the system, because all the necessary resources to run the application being contained are inside the container. Containers are an alternative to directly installing an application on a server within an OS running other programs.

When using traditional methods of installing applications on servers, all sorts of aspects of the server state could affect the runtime of the application, such as environment variables, the installed libraries, networking configurations, and even system time. When using containers, developers get complete control over what's inside the application's view of the world and prevent the base OS or other running applications from interfering with proper operation of the application. Similarly, applications are very tightly limited in the resources they can consume inside a container, so there's less chance of a containerized application interfering with another application.

The Hareloom Farms team chooses to use Docker, a popular implementation of containers, as their standard method of packaging applications. One reason you choose Docker specifically is that a wealth of tooling and infrastructure has been built around Docker, some of which you'll see in this chapter and some in the next.

For the latest instructions on how to install Docker, visit the Docker website: www.docker.com. This technology is evolving fast, and the setup details vary a great deal by OS. Regardless of how you choose to set up Docker, once you can run `docker run` hello-world and see a successful result, your installation is complete, and you can begin containerizing applications!

Next, you'll need to set up some functionality within your build to help you containerize your model service. Sbt is a sophisticated build tool that you can use to do a lot of tasks related to building and deploying your code. For instructions on how to get started with sbt, see the Download section (www.scala-sbt.org/download.html) of the sbt website (www.scala-sbt.org). In this case, you'll use a plugin for sbt called sbt-docker that will help you work with Docker. To install this plugin, add it to your /project/plugins.sbt file.

Listing 7.12 Adding an sbt plugin

```
addSbtPlugin("se.marcuslonnberg" % "sbt-docker" % "1.4.0")
```

Then you'll enable the plugin in your build by editing your build definition in build.sbt.

Listing 7.13 Enabling an sbt plugin

```
enablePlugins(DockerPlugin)
```

Now that you have the tools in place, you need to define a build for the project. This is stuff you need to do regardless of whether you use Docker, if you want to define how to build your code into a runnable artifact. For this build, you'll use the standard package build task defined by default in sbt. But thanks to sbt-docker, you can take this further and define how you'd like the built JAR to be packaged into a Docker container.

Listing 7.14 Building a Docker image in sbt

```
dockerfile in docker := {
  val jarFile: File = sbt.Keys.`package`
  ➥ .in(Compile, packageBin).value
  val classpath = (managedClasspath in Compile).value
  val mainClass = "com.reactivemachinelearning.PredictiveService"
  val jarTarget = s"/app/${jarFile.getName}"
  val classpathString = classpath.files.map("/app/" + _.getName)
    .mkString(":") + ":" + jarTarget

  new Dockerfile {
    from("java")
    add(classpath.files, "/app/")
    add(jarFile, jarTarget)
    entryPoint("java", "-cp", classpathString, mainClass)
  }
}
```

Defines how to build a Dockerfile to be produced by a docker build task

The location to put the JAR in the output

Looks up the classpath

Defines the main class, the runnable entry point of the build, to be PredictiveService

Looks up the location of the JAR file produced

Builds up a classpath string with the JAR on it

Defines instructions for constructing a Dockerfile

Base Docker image to build on top of

Adds all the dependencies files on the classpath and other resources in the app directory

Adds the built JAR

Defines an entry point of the application to be running Java with the classpath to execute the main class

Now, with this build defined, run `sbt docker` to build the app as a JAR inside a Docker container that will start up the predictive service on initialization.

Using `sbt docker`, you skipped over one of the parts of building a Docker image that you might otherwise have done manually: defining a Dockerfile. The Dockerfile is essentially the instructions about how to build your Docker image from other Docker images and the unique resources used in your image. You can find the Dockerfile you produced in the target/docker directory of your project. When you open it, it should look something like the following listing (except much longer).

Listing 7.15 Dockerfile

```
FROM java
ADD 0/spark-core_2.11-2.0.0-preview.jar 1/avro-mapred-1.7.7
➥ -hadoop2.jar ...
ADD 166/chapter-7_2.11-1.0.jar /app/chapter-7_2.11-1.0.jar
ENTRYPOINT ["java", "-cp", "\/app\/spark-core_2.11-2.0.0
➥ -preview.jar ...
 "com.reactivemachinelearning.PredictiveService"]
```

Base image being built on

Adds the dependency JARs

Adds an application JAR

Defines the entry point of the application to be running Java

Indicates that Java should be run on the predictive service to start the application

Your Dockerfile will be much longer, but it shouldn't be any more complex than listing 7.15. The Dockerfile lists out all the dependencies to be included in the image and then instructs Java where they are using the `-cp` (classpath) argument.

Using this Dockerfile, Docker will build an image containing everything needed to run your predictive service inside a container and then place that image in your local Docker repository. In the production pipeline at Hareloom, your team uses their continuous-delivery infrastructure to push their Docker images to a remote repository as part of the build process, which works very similarly to the local build you just did.

You can run the service by calling `docker run default/chapter-7` or whatever Docker told you it had tagged the built image with as a name. Docker will then retrieve your built image from your local Docker repository and run it.

There's still a lot more you could do with this predictive microservice now that you have it packaged inside a Docker container. It can be deployed into all sorts of environments, at massive scale, taking advantage of sophisticated container-management systems. But we'll leave the discussion of how to use machine-learned systems in your live system for chapter 8.

If you want to go deeper into how to use Docker, this book definitely shouldn't be your last stop. *Docker in Action* by Jeff Nickoloff (Manning, 2016, www.manning.com/books/docker-in-action) provides a deeper introduction to the larger process of working with Docker models, and *Docker in Practice* by Ian Miell and Aidan Hobson Sayers (Manning, 2016, www.manning.com/books/docker-in-practice, new edition forthcoming) can show you more-advanced aspects of deploying applications using Docker.

Beyond those books, there's an ever-increasing amount of resources for working with containers online.

7.5 Reactivities

- *Build a microservice.* This book contains a number of examples of building services. As you've done before, you can implement your own microservice. You can either take some real functionality you'd like to deploy, like wrapping a machine-learned model, or you can take a dummy function just to focus on the service infrastructure. Often when I'm building systems, I take the latter approach and do something like a service that randomly returns either true or false. This "random model" allows me to focus on the properties of the service implementation, independent of the service functionality.

  Tools like Akka HTTP make it pretty easy to expose a given function to the web as a service, so this reactivity can be as simple or as complex as you choose. With this microservice, you can then begin to think through questions like how it can be deployed for high load, how failure would be detected and managed, how it might work in concert with other services, and so on.
- *Containerize an application.* Take either the microservice from the previous reactivity or some other application and build it into a container. If you really want to dig into this process, begin with figuring out the requirements of your base image. What OS will your image contain? If you have a Scala service, how is the JVM installed and configured? Where do you pick up your dependencies from? When you make changes to your service, what parts of the image come from previously built layers versus having to be rebuilt? Can you tell if your application was built properly in the container build? How do you start your application inside your container?

Summary

- Models, and even entire training pipelines, can be persisted for later use.
- Microservices are simple services that have very narrow responsibilities.
- Models, as pure functions, can be encapsulated into microservices.
- You can contain failure of a predictive service by only communicating via message passing.
- You can use an actor hierarchy to ensure resilience within a service.
- Applications can be containerized using tools like Docker.

The full process of model publishing looks a lot different than reusing a model that you just learned in a REPL like the Spark shell. The tools and techniques you learned in this chapter will allow you to use machine-learned models in sophisticated real-time systems of all kinds, which is the topic of chapter 8.

8 Responding

This chapter covers

- Using models to respond to user requests
- Managing containerized services
- Designing for failure

Now we come to the final component of a machine learning system—the part responsible for using models to respond to user requests and act on the real world (see figure 8.1). In the last chapter, we began using models in a more real-world way than just playing with them on a laptop. The approach we took involved building predictive microservices that wrapped models and then putting those microservices into containers. This chapter continues that approach by using containerized predictive services in systems that are exposed to real requests for predictions.

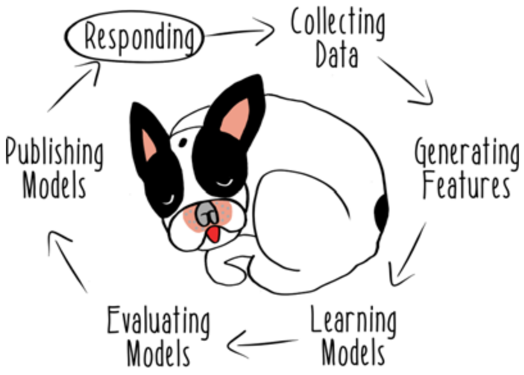


Figure 8.1 Phases of machine learning

Using models in the real world is tough. To learn about all the complexity of using models in the real world, we need to move from the quiet of the farm to the bustle of the big city. We'll consider the fastest-moving animals in the city: turtles.

8.1 Moving at the speed of turtles

One of the most successful startups in the entire animal kingdom is Turtle Taxi, a technologically sophisticated take on the business model of taxis. In many major cities, they've largely displaced legacy transportation businesses like Caribou Cabs.

Part of their success is due to their user-friendly mobile app, which allows riders to hail a taxi from anywhere at any time. But a less obvious part of their success is machine learning. Turtle Taxi employs a large team of semiaquatic data scientists and engineers (including you) who perform sophisticated online optimization of their transportation infrastructure. In comparison to something like a city bus or rail system, the Turtle Taxi fleet is a much harder system to manage. Because drivers have no fixed schedules and can drive whenever they choose to, the fleet of vehicles available to serve customers is always changing. Similarly, customers choose to hail a ride whenever they need one, so there are no static schedules like in a traditional public-transit system.

This highly dynamic environment creates huge challenges for the data team at Turtle Taxi. They need to answer important business questions, including the following:

- Are enough drivers on the road to serve demand?
- Which driver should serve which request?
- Should the price of a ride move up or down based on demand?
- Are customers getting good or bad service?
- Are drivers in the right part of town to serve the demand?

The Turtle Taxi team has spent a lot of time and effort ensuring that their machine learning system holds to reactive properties. They learn a lot of models to help their systems make autonomous decisions about all those complex business problems, and their infrastructure helps them *use* those models in real time, at scale. We'll begin with a simplified view of that infrastructure, leaving discussion of some of the more complex real-world issues for later in the chapter.

8.2 Building services with tasks

In previous chapters, you've seen various techniques and tools for building services of different kinds. In chapter 7 specifically, you used Akka HTTP, a component of the Akka toolkit, to build your services. Though Akka HTTP is an excellent choice for a lot of applications, this chapter introduces an alternative library for building services, called http4s.

Whereas Akka HTTP developed from work using the actor model of concurrency, http4s provides a programming model influenced more by functional programming. The main difference between the two design philosophies manifests in the user API. When you use http4s, you don't use actors or set up an execution context for an actor system. Http4s is part of the Typelevel family of projects (http://typelevel.org) and uses many other libraries from that group in its implementation. You can build very complex services using http4s, but you'll use it here primarily for its simplicity.

One new concept we should look into before exploring how to build services with http4s is tasks. *Tasks* are related to futures but are a more sophisticated construct that allows you to reason about things like failure and timeouts. Implemented and used properly, tasks can also be more performant than standard Scala futures due to how they interact with the underlying concurrency facilities provided by the JVM. In particular, with tasks you can express computation you might not ever execute. This section will show you how to use this capability in your programs.

The implementation of tasks in this chapter comes from the popular scalaz project. For those unfamiliar with scalaz, it's a project focused on providing advanced functional programming features in Scala, similar to the goals of the Typelevel family of projects. Unfortunately, scalaz's implementation of tasks is famously poorly documented, so I'll provide you with the basic information necessary to use it here.

> **NOTE** Tasks, like futures, are a powerful concurrency abstraction that can be implemented in various different ways. Monix (https://monix.io), also from the Typelevel family of projects, is an alternative implementation of the concept of tasks.

Like futures, tasks allow you to execute asynchronous computation, but futures are eager, as opposed to lazy, by default. In this context, *eager* means execution begins immediately. Assuming that futures are lazy is a common and logical mistake, but though they're asynchronous, they're in fact eager. They start executing immediately, even if you might like to delay the start of execution. Listing 8.1 demonstrates this sometimes undesirable property. In this listing and listing 8.2, which demonstrates tasks, assume that `doStuff` is an expensive, long-running computation that you only want to trigger when you're ready to.

Listing 8.1 Eager futures

```
import scala.concurrent.ExecutionContext.Implicits.global
import scala.concurrent.Future

def doStuff(source: String) = println(s"Doing $source stuff")

val futureVersion = Future(doStuff("Future"))
```

Imports the execution context to be used for a Future

Defines the function to represent your expensive work

Instantiates a Future (and starts work)

```
Thread.sleep(1000)

println("After Future instantiation")
```

Waits 1 second to make apparent that the previous line has begun execution

Shows that the next line of code will execute only after work from the future has been submitted for execution

If you execute this code on your console, you should see output like the following:

```
Doing Future stuff
After Future instantiation
```

Sometimes this property isn't an issue, but sometimes it is. For example, if you wanted to define that long-running computation for all the requests that the service gets, but only run that computation 1% of the time, a `Future` would have your service doing 100 times the work you'd want to do. In a system where you have many models available to make predictions, you might only want to perform the prediction on a small subset of qualifying requests for any given model. It would be nice to have another option that doesn't do work that you don't want the system to do.

The next listing shows how tasks behave differently than futures.

Listing 8.2 Lazy tasks

```
import scalaz.concurrent.Task

val taskVersion = Task(doStuff("Task"))

Thread.sleep(1000)

println("After Task instantiation")

taskVersion.run
```

Waits 1 second to make apparent that the previous line hasn't yet begun execution

Instantiates a Task but doesn't start work

Shows that an entire second has passed

Executes the Task

In contrast to listing 8.1, this code, if executed, should produce output that looks like the following:

```
After Task instantiation
Doing Task stuff
```

Now you have control over when and *if* you do work like long-running computations. This is clearly a powerful feature of tasks, and it's the basis for many of the rest of the more advanced features of tasks, such as cancelability. When you use http4s to build services, you don't need to know too much more than this about tasks, but understanding the basis of the performance properties of the library is helpful.

Tasks are just one aspect of the functionality that http4s provides for building performant services. The library also uses scalaz streams to process arbitrary amounts of data.

That's just a taste of what you can do with these libraries, but it should be enough for you to start building predictive services.

8.3 Predicting traffic

Now that I've introduced the tools, let's get back to solving the problem. In particular, let's consider the problem of matching taxi drivers to riders. The Turtle Taxi team uses machine learning to predict successful driver-rider matches. For a given rider, their system will attempt to predict from a set of available drivers which one the rider will most likely enjoy riding with (as recorded by the driver rating on the mobile app). This section will walk through how your team can create a service to make predictions on this driver-rider match-success problem.

To begin, you'll need to create some models to work with. Because the past chapters have already covered several different ways to produce machine-learned models, I won't repeat all that material here. To build the infrastructure you need for this chapter, you can use simple *stub* (fake) models.

The Turtle Taxi team uses a lot of models, so you'll start with building support for multiple models from the beginning. As discussed in chapter 5, using multiple models in combination is called an *ensemble*. In this section, you'll do something a bit different than an ensemble. Instead of using your models in combination, you'll use one or the other on any given prediction request. As you've seen in previous chapters, real-world machine learning systems use models in lots of different ways. Big data teams like Turtle Taxi's often produce many models for different purposes. These models may have different strengths and weaknesses that the team figures out by using the models. Experimentation on different modeling techniques is an important part of the machine learning process. Your Turtle Taxi team has built the system to allow them to test different learned models in production, so you'll approximate that implementation here. In particular, you'll build a simple model experimentation system that will send some traffic to one model and some to another to evaluate model performance. Figure 8.2 shows a simple form of what you're going to build.

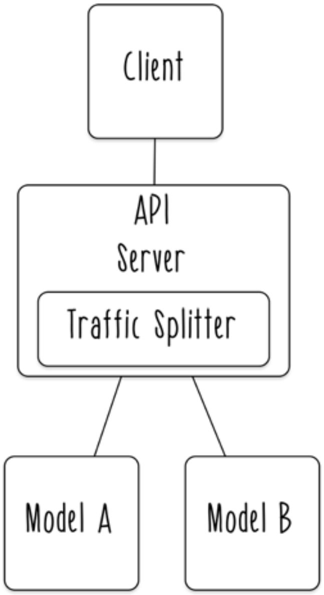


Figure 8.2 Model experimentation architecture

Listing 8.3 shows how to create two simple stub models that predict true or false, structured as services. They represent two different models of driver-rider match success.

Listing 8.3 Stub models

```
import org.http4s._
import org.http4s.dsl._

object Models {
  val modelA = HttpService {
    case GET -> Root / "a" / inputData =>
      val response = true
      Ok(s"Model A predicted $response.")
  }

  val modelB = HttpService {
    case GET -> Root / "b" / inputData =>
      val response = false
      Ok(s"Model B predicted $response.")
  }

}
```

Defines a model as an HTTP service

Uses pattern matching to determine that the request to model A has been received

Always responds true for model A

Returns an OK status code with a model's prediction

Defines a similar stub model service for model B

Always returns false for model B

Note that these models are constructed as HTTP services. Once you finish building all the necessary infrastructure, they'll be independently accessible via any client capable of sending HTTP to them on the network (for example, your local computer).

Though you haven't done everything necessary to expose these models as services, let's start to flesh out how you'd like to call these services. For development purposes, let's assume you're serving all of your predictive functionality from your local computer (localhost) on port 8080. Let's also namespace all your models using the name of a given model under a path named *models*.

Using those assumptions, you can create some client helper functions to call your models from other parts of the system. It's important to note that you define this client functionality in Scala in the same project purely as a convenience. Because you're constructing these services as network-accessible HTTP services, other clients could easily be mobile apps implemented in Swift or Java, or a web frontend implemented in JavaScript. The client functionality in the next listing is an example of what a consumer of the success match-predictions functionality might look like.

Listing 8.4 Predictive clients

```
import org.http4s.Uri
import org.http4s.client.blaze.PooledHttp1Client

object Client {

  val client = PooledHttp1Client()

  private def call(model: String, input: String) = {
    val target = Uri.fromString(s"http://localhost:8080/
    ➥ models/$model/$input").toOption.get
```

Creates an object to contain client helpers

Instantiates an HTTP client to call modeling services

Factors out the common steps of calling models to a helper function

Dangerous technique: creates a URI to call a model from dynamic input and forces immediate optimistic parsing

```
        client.expect[String](target)
      }
      def callA(input: String) = call("a", input)

      def callB(input: String) = call("b", input)

    }
```

Creates a Task to define a request

Creates a function to call model A

Creates a function to call model B

The use of `.toOption.get` in listing 8.6 isn't good style—I'm using it as a development convenience. The implementation of the URI-building functionality in http4s is trying to be a bit safer about dynamically generated values like the name of the model and the input data. A future refactor of this code could focus on more-sophisticated error handling or use a statically defined route, but for now you'll accept that you could receive unprocessable input that would throw errors.

You want to expose a public API that abstracts over how many models you might have published to the server at any given time. Right now, the turtles want to have model A receiving 40% of the requests for predictions and model B receiving the remaining 60%. This is an arbitrary choice they've made for preferring model B until model A demonstrates superior performance. You'll encode that split using a simple splitting function to divide traffic based on the hash code of the input data, similar to how you divided data in chapter 6. The next listing shows the implementation of this hashing function.

Listing 8.5 Splitting prediction requests

```
    def splitTraffic(data: String) = {
      data.hashCode % 10 match {
        case x if x < 4 => Client.callA(data)
        case _ => Client.callB(data)
      }
    }
```

Function to split traffic based on input

Hashes the input and takes the modulus to determine which model to use

Uses pattern matching to select model A 40% of the time

Uses model B in the remainder of cases

If you had more models deployed, you could extend this approach to something more dynamic based on the total number of models deployed and the amount of traffic they should receive.

Now that you have these pieces in place, you can bring all this together into a unified model server. In this case, you'll define your public API as located at a path named *api* and the prediction functionality as specifically located under the predict path of api.

Listing 8.6 A model service

```
    import org.http4s.server.{Server, ServerApp}
    import org.http4s.server.blaze._
    import org.http4s._
    import org.http4s.dsl._

    import scalaz.concurrent.Task

    object ModelServer extends ServerApp {
```

Defines the model-serving service as a ServerApp for a graceful shutdown

```
  val apiService = HttpService {
    case GET -> Root / "predict" / inputData =>
      val response = splitTraffic(inputData).run
      Ok(response)
  }

  override def server(args: List[String]): Task[Server] = {
    BlazeBuilder
      .bindLocal(8080)
      .mountService(apiService, "/api")
      .mountService(Models.modelA, "/models")
      .mountService(Models.modelB, "/models")
      .start
  }

}
```

Defines another HttpService to be the primary API endpoint for external use

Uses pattern matching to define when a prediction request has been received

Passes input data to a traffic-splitting function, immediately invoking it

Passes through a response with an OK status

Defines the behavior of the server

Uses the built-in backend from Blaze to build the server

Binds to port 8080 on a local machine

Mounts an API service to the path at /api

Attaches a service for model A to the server at /models

Attaches a service for model B to the server at /models

Starts the server

Now you can see your model server in action. If you've defined a way to build the application, you can build and run it.

For an example of how to set up a build for this application, see the online resources for this book (www.manning.com/books/reactive-machine-learning-systems or https://github.com/jeffreyksmithjr/reactive-machine-learning-systems). Once your application can be built, you can issue `sbt run`, and your service should start up and bind to port 8080 on your local machine.

You can test your service using a standard web browser and hitting the API endpoint with various endpoints. For example, if the string `abc` represents a valid feature vector for this service, then hitting http://localhost:8080/api/predict/abc produces a prediction of false (no match) from a prediction from model B.

Looking back on what you just built, you see some useful functionality. It has a simple way of handling multiple models. Moreover, it should be pretty obvious how you could get at least some elasticity by starting up more instances of your model services and maybe putting them behind a load balancer.

You can see a sketch of such an architecture in figure 8.3. It's not a bad approach, but it still lacks some realism. Turtles are tough creatures who know how to prepare for the worst that life can throw at them. Let's look at how they've hardened their machine learning systems.

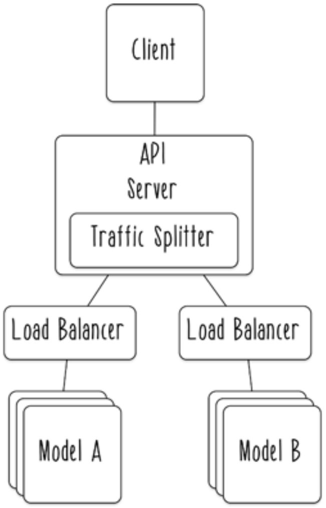


Figure 8.3 Load-balanced model services

8.4 Handling failure

As you've seen multiple times in this book, things fail. Whether those things are pangolins, poison dart frogs, or plain old databases, nothing runs without errors.

Model services are no different. You get some nice properties from pulling apart the different components of your system, like containment and the ability to supervise components. But your current implementation is still vulnerable to the consequences of failure.

Let's examine how you can deal with failure by building a model that fails half the time. That should give you plenty of opportunity to deal with failures. Listing 8.9 is another simplified stub model like the ones you built earlier, but with one important difference: it treats half of all requests as bad requests and fails to return a prediction.

Listing 8.7 An unreliable model

```
import scala.util.Random

val modelC = HttpService {
  case GET -> Root / "c" / inputData => {

    val workingOk = Random.nextBoolean()

    val response = true

    if (workingOk) {
      Ok(s"Model C predicted $response.")
    } else {
      BadRequest("Model C failed to predict.")
    }
  }
}
```

Creates an HttpService for model C

Defines the same GET endpoint as the other models

Always predicts true when the service works

Simulates an occasional failure with a random Boolean value

Determines whether the service is in a working or nonworking state

Returns a normal successful prediction

Fails to predict, returns a BadRequest status code

This irritatingly unreliable model is a good stand-in for the real possibility of failure in your system, so how could you handle the possibility of this failure? You can build the possibility of failure into your system using supervisory hierarchy, as you saw in chapters 2 and 3.

The next listing begins this refactor by tweaking how you call the model services.

Listing 8.8 Refactored calling services

```
private def call(model: String, input: String): Task[Response] = {
  val target = Uri.fromString(
    s"http://localhost:8080/models/$model/$input"
    ).toOption.get
  client(target)
}

def callC(input: String) = call("c", input)
```

Redefined call helper function

Calls a target with a client and returns Task[Response]

Defines a helper function for calling model C

After this refactor, the call to the service returns a `Task[Response]`. I think this approach is a bit more straightforward about what work you're doing. Specifically, this new type signature encodes two bits of knowledge: that this call will take time to do, and that this call will return a `Response`, which might not be a successful one.

Next, let's see how you can handle the possibility of failure at the top level. Before, you had a mere `ModelServer` whose job was just to handle passing around data from requests to models and back. With these changes, you're beginning to build a `ModelSupervisor`, something with a hierarchical responsibility to decide what to do in the event of undesirable outcomes. In this context, you want to recognize when models fail and pass any messages about that failure back to the user. That's a design choice. In other situations, you might want to do something different, such as return a default response. The point is that you now explicitly handle failure and make a decision *that you can see in source code* about what to do about it.

Listing 8.9 A model supervisor aware of failure modes

```
import org.http4s.server.{Server, ServerApp}
import org.http4s.server.blaze._
import org.http4s._
import org.http4s.dsl._

import scalaz.concurrent.Task

object ModelSupervisor extends ServerApp {

  def splitTraffic(data: String) = {
    data.hashCode % 10 match {
      case x if x < 4 => Client.callA(data)
      case x if x < 6 => Client.callB(data)
      case _ => Client.callC(data)
    }
  }

  val apiService = HttpService {
    case GET -> Root / "predict" / inputData =>
      val response = splitTraffic(inputData).run

      response match {
        case r: Response if r.status == Ok =>
        ➥ Response(Ok).withBody(r.bodyAsText)
        case r => Response(BadRequest).withBody(r.bodyAsText)
      }
  }

  override def server(args: List[String]): Task[Server] = {
    BlazeBuilder
      .bindLocal(8080)
      .mountService(apiService, "/api")
      .mountService(Models.modelA, "/models")
      .mountService(Models.modelB, "/models")
```

Redefined traffic-splitting function

Defines the last 40% of traffic as being allocated to model C

Pattern matches on the result of calling a service

Returns successful responses with the model's prediction as the body of the responses

Returns failed responses with a failure message as the body of the responses

```
            .mountService(Models.modelC, "/models")
            .start
        }

    }
```

Adds model C to the services being served

Again, you can do whatever you choose in the `case` clause that defines how you handle failure, because now you have explicit control via the supervisory structure.

Now let's see how this structure works. To do the next phase of testing, I suggest we stop writing Scala code and instead use command-line utilities. Specifically, let's use cURL, a useful, open source tool you may already have installed on your system (if you're using macOS or Linux). If you're using Windows, you may need to download the latest version from the cURL website (https://curl.haxx.se).

With cURL, you can send data to your API server and model services in the same way you were doing with the web browser before. The advantage of using cURL is that you can set more options around how you interact with your server-side applications and how you inspect their results. In the following examples, you'll use the `-i` option to inspect the HTTP headers being returned from your services.

The nest listing introduces how to use cURL by calling to an API endpoint that maps to a model that it's behaving normally.

Listing 8.10 A successful response

```
$ curl -i http://localhost:8080/api/predict/abc
HTTP/1.1 200 OK
Content-Type: text/plain; charset=UTF-8
Date: Sun, 02 Oct 2016 21:07:31 GMT
Content-Length: 114

Model B predicted false.
```

OK status code from a header

Calls for a prediction and shows headers

Information about the response sent back

Prediction of model B

That all works fine, but what if you use the periodically unreliable model C? In some cases, it will behave like this.

Listing 8.11 A possible successful response

```
$ curl -i http://localhost:8080/api/predict/abe
HTTP/1.1 200 OK
Content-Type: text/plain; charset=UTF-8
Date: Sun, 02 Oct 2016 21:12:32 GMT
Transfer-Encoding: chunked

Model C predicted true.
```

Successful response

Calls for a different prediction, maps to a different model

Prediction of model C

Everything is fine, and you see exactly what you expect to see. But the other half of the time, you should see a failed request that looks something like this.

Listing 8.12 A possible failed response

```
$ curl -i http://localhost:8080/api/predict/abe
HTTP/1.1 400 Bad Request
Content-Type: text/plain; charset=UTF-8
Date: Sun, 02 Oct 2016 21:27:33 GMT
Transfer-Encoding: chunked

Model C failed to predict.
```

A failed response from a prediction request

Failure message

As expected, sometimes model C completely drops the ball. That's bad news for taxi-driving turtles. This failure to predict could have very negative consequences for the rest of the system, depending on how the caller was implemented. At its worst, this failure could bring down the whole process of matching riders to drivers, and that could mean lost fares.

Yet this failure, like lots of real-world failures, is ephemeral; the model doesn't always fail to predict. When it does return a prediction, you have no reason to not use it. The model should be a pure, stateless function. You can still build a solution that accommodates the possibility of failure—it will just take a bit more effort. One possible solution could look like the following listing, which sets up retry logic.

Listing 8.13 Adding a retry to calls to models

```
private def call(model: String, input: String) = {
  val target = Uri.fromString(s"http://localhost:8080/
  ➥ models/$model/$input").toOption.get
  client(target).retry(Seq(1 second), {_ => true})
}
```

Retries after 1 second for all failures

With this approach, you'll immediately halve the failure rate by retrying once. For the dummy unreliable model, this retry strategy could asymptotically approach full reliability.

8.5 Architecting response systems

This chapter introduced some strategies for using models to respond to requests from users outside your machine learning system. I've tried to keep things realistic enough to be useful by including complications like multiple models and the possibility of failure. But believe it or not, the real Turtle Taxi machine learning system is more complicated than this. It has millions of users, with tens to hundreds of thousands of users active at a given time. Riders need to be connected to available drivers in their area within seconds and picked up within minutes—otherwise, the Turtle Taxi business will grind to a halt and fail. With all these demanding real-world needs, the team's real infrastructure looks a bit more like figure 8.4.

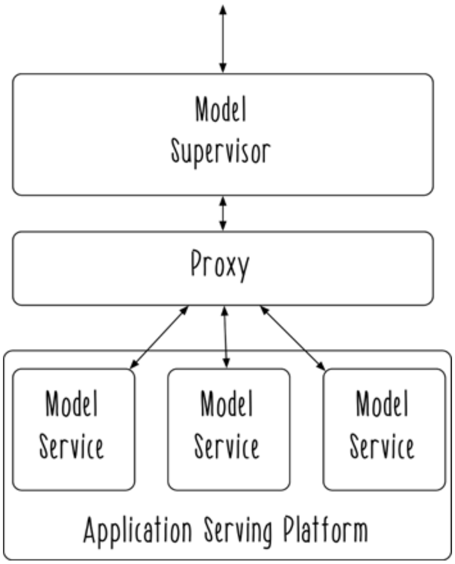


Figure 8.4 Model-serving architecture

Just as you implemented in this chapter, all models are individual services, but each of those services is independently packaged in some form or another. Docker is a common choice, using techniques like you explored in the last chapter. But the JVM also contains an approach to packaging—building JARs—that can be used on some application-serving platforms. You worked with JARs a little in chapter 7, and in chapter 9 you'll work with them in more detail.

Containment

Each of these model services is hosted on what I'm calling an *application-serving platform*, but is sometimes called a *container orchestration platform* (usually, when it's focused on using Docker or other forms of containers). Examples of platforms like this are Marathon (https://mesosphere.github.io/marathon) and Kubernetes (http://kubernetes.io), both open source software; Amazon's EC2 Container Service (https://aws.amazon.com/ecs); Microsoft's Azure Container Service (https://azure.microsoft.com/en-us/services/container-service); and Kubernetes Engine (https://cloud.google.com/kubernetes-engine), which are all cloud-hosted services. A key aspect of these solutions is that they allow you to host packaged applications and manage them using some interface. They all offer a level of intrinsic containment by isolating the resources used by any one application from all other applications, thus limiting the possibility of problems like error propagation.

Supervision

As you've seen in this chapter, once you have models available for use, there's still a lot to figure out. Figure 8.4 represents a lot of the functionality you worked on in this chapter as the model supervisor. It also makes clear that many of the networking components you implemented in Scala in this chapter are often handled by another component, which the diagram calls a *proxy*. The role of this component is purely to route requests for predictions to the specific service that should serve them. Examples of applications that can serve this role are NGINX (www.nginx.com) and HAProxy (www.haproxy.org). In practice, many container-oriented platforms can also handle some amount of this networking complexity. Note that both the model supervisor and the proxy could in principle be hosted on the same application-serving platform as the model services. However it's implemented, the job of this component is the same: to give the model supervisor the ability to manage models and the traffic sent to them.

Architectures like this are nontrivial to implement. Typically, you'd need a whole team of smart turtles to stand up all these components and get them working effectively in concert.

But you can certainly build incrementally from the simplified designs you implemented in this chapter toward the more sophisticated approaches. The underlying principles of reactive design remain the same.

8.6 Reactivities

- *Build a pipeline of data transformations using tasks.* Now that you've seen them in action, you may be interested in doing a bit more with tasks. One logical use case for tasks is compute-intensive data-transformation pipelines. Such pipelines are common in machine learning, especially (but not exclusively) in feature-generation pipelines. One of the nice things about tasks is that they can be composed in various ways and then run concurrently. You can try implementing things like y-shaped operation graphs in your pipeline, where two dependent steps must execute concurrently before a third can begin.

  If you want to dig deeper into the behavior of your pipeline, try introducing faults into one of the steps via bad data or some other technique:

  – How does your pipeline react when it encounters bad input?
  – Is that behavior what you want?
  – If not, how could you change it?

- *Deploy a containerized service to an application-serving platform.* If you've been following along in this book, you should have a service that builds inside a container. The great thing about containers is that they're portable, so deploy that service somewhere. Quite a few different cloud vendors supply container-hosting services, and usually your initial usage is free (see section 8.5 for a few options). You can also choose to deploy your containers to an application-serving platform that you yourself are hosting. That's a bit more involved, but if you happen to already have something like Marathon on a Mesos cluster running at your office, you can use one of those options as well.

  Once you've deployed the service, you can think through what it might mean to operate it on an ongoing basis:

  – What would happen if requests to the service increased dramatically?
  – How can you tell that the deployed service is doing what it's supposed to do?
  – How can you roll back to a previous version of your service?

– What would happen to your service if one of the underlying servers for the application-serving platform went away? (If you don't know the answer, you could always send a shutdown command to a server under your control and see what happens!)

Summary

- Tasks are useful lazy primitives for structuring expensive computations.
- Structuring models as services makes elastic architectures easier to build.
- Failing model services can be handled by a model supervisor.
- The principles of containment and supervision can be applied at several levels of systems design to ensure reactive properties.

This concludes part 2 of the book. In part 3, we'll explore some of the more advanced issues involved with keeping a machine learning system running, changing, and scaling.

Part 3

Operating a machine learning system

Whereas part of this book focused on getting to the point of a having an entire machine learning system, part 3 is about what comes next. All sorts of work must be done over the lifetime of operating a machine learning system, making it possible for you to change and improve the system.

Chapter 9 goes deeper into how to build and deploy machine learning systems. It covers best practices developed for the operations of other sorts of software applications and applies them to the unique responsibilities of a reactive machine learning system.

Chapter 10 works through how you can incrementally improve the intelligence capabilities of a system. It introduces the far-reaching ambitions of artificial intelligence platforms. This final chapter provides the book's most expansive perspective on how to design systems. I encourage you to adopt this broad perspective as a way of thinking about the skills you've developed as an architect of reactive machine learning systems.

9 Delivering

This chapter covers

- Building Scala code using sbt
- Evaluating applications for deployment
- Strategies for deployments

Now that you've seen how all the components of a machine learning system work together, it's time to think about some system-level challenges. In this chapter, we'll explore how to *deliver* a machine learning system for use by the ultimate customers of the system. The approach we'll use for this challenge is called *continuous delivery*. The ideas behind continuous delivery were developed outside of a machine learning context, but as you'll see, they're entirely applicable to the challenge of making machine learning reactive.

Continuous delivery practitioners seek to rapidly deliver new units of functionality through regular cycles that build and deploy new code. Teams that take on this approach are often trying to move fast while keeping users happy. The techniques of continuous delivery provide tactics that allow teams to fulfill these competing aims. Given all the ways you've seen for machine learning systems to fail, I hope it's clear that maintaining consistent behavior in these systems is tough stuff. Uncertainty is pervasive and intrinsic in machine learning systems.

9.1 Shipping fruit

The team you'll join in this chapter is composed of some of the most customer-focused, empathetic animals in the entire jungle: gorillas. Jungle Juice Box (JJB) is a primate startup focused on animals who like to make fruit smoothies in their own home (figure 9.1). Each month, subscribers to Jungle Juice Box receive a box of fresh fruit that has been individually selected for them, based on what fruit is in season and the customer's perceived preferences.

Figure 9.1 Jungle Juice Box

As with most startups today, the gorillas of Jungle Juice Box use sophisticated data techniques to satisfy their customers. In particular, they regularly gather feedback ratings from all their subscribers so they can use machine learning to make unique fruit recommendations for each subscriber. The architecture of the portion of the system concerned with fruit recommendations looks something like figure 9.2.

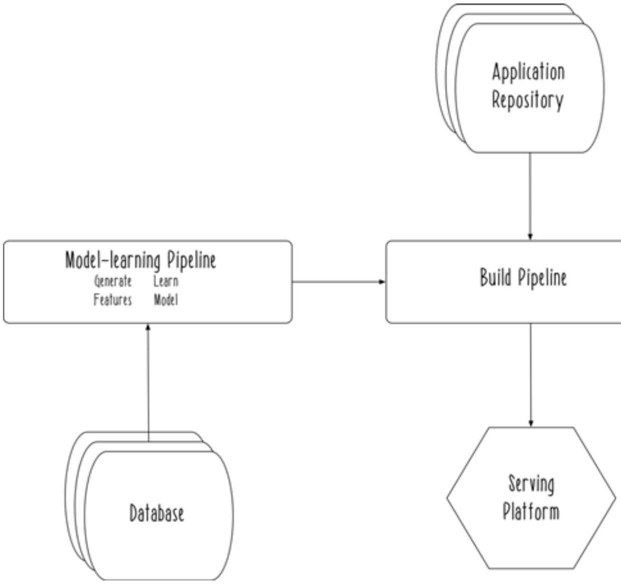


Figure 9.2 Fruit-selection architecture

Past subscriber ratings of fruit selections, stored in a database, are used to produce features and concept labels for training instances. Models are then learned from those instances in the model-learning pipeline. Those models are included in the build of the larger application in the build pipeline. The build pipeline produces an artifact that's then deployed to production systems on an application-serving platform for use in real-time fruit-selection decisions. All this should look relatively familiar from part 2 of the book. This chapter focuses on the key part of the system where new models are evaluated and deployment decisions are made. Within the JJB system, this point is the build-and-deploy pipeline.

9.2 Building and packaging

At this point, it's worth taking a quick step back and examining what I mean by *building code*. In this book, you've primarily built applications in Scala, which runs on top of the Java Virtual Machine (JVM). At a minimum, your source-code files need to be compiled into bytecode for execution by the JVM, but you often need to do more than compile your code. Sometimes, there are resource files, like the Parquet and JSON model files you produced in chapter 7. These other types of artifacts are part of what you need to run your code, even if they're directly output by the `scalac` compiler. Usually, all you need to do with such files is ensure that they can be passed around with the rest of your code. This process is often called *packaging*.

Chapter 7 demonstrated one powerful approach to packaging using Docker containers. But there are certainly other approaches you can take to package your applications. Anything that gives you a coherent way of grouping all the executable code and necessary resources for your application is a potentially viable option.

Because the gorillas at JJB build their machine learning systems in Scala, they use a packaging method from the JVM ecosystem: building JARs. As discussed in chapter 7, JARs are archives of compiled JVM code and associated resources. When working with Scala, you have several ways that you can produce JARs. The Jungle Juicers use a method that's executed from sbt and relies on a plugin to extend sbt's ability to produce deployable artifacts.

The way the Juicers distribute the executable version of their application is in a JAR containing all the necessary dependencies (libraries) that the application needs. The artifact produced by this style of packaging is sometimes called a *fat JAR*, meaning it contains all the dependencies that could otherwise potentially be provided by the execution environment. They choose this approach because it simplifies some aspects of distributing and executing their application.

To get started, you'll need to add the `sbt-assembly` plugin to your project. Within your project, create a directory called *project* and a file named *assembly.sbt* in that directory. That file should contain the instructions to add the `sbt-assembly` plugin.

Listing 9.1 Adding `sbt-assembly`

```
addSbtPlugin("com.eed3si9n" % "sbt-assembly" % "0.14.3")
```

Then you need to define a build, as in the next listing, which shows how your JJB team sets up a build.

Listing 9.2 An SBT build

```
lazy val http4sVersion = "0.14.6"

libraryDependencies ++= Seq(
  "org.http4s" %% "http4s-dsl" % http4sVersion,
  "org.http4s" %% "http4s-blaze-server" % http4sVersion,
  "org.http4s" %% "http4s-blaze-client" % http4sVersion,
  "org.http4s" %% "http4s-argonaut" % http4sVersion
)

mainClass in Compile := Some
➥ ("com.reactivemachinelearning.ModelServer")
```

Defines dependencies

Sets the main class, to be executed when the archive is run

If you've done all that correctly, you can now build your project by issuing the command `sbt assembly`. The command will produce a JAR containing all your code and resources, along with all the dependencies your code requires. The assembly task will tell you where this JAR is with a message that says something like, "Packaging /yourapp/target/scala-2.11/your-app-assembly-1.0.jar...." This single archive can now be passed around to any execution environment that can run JVM code.

9.3 Build pipelines

This build-and-packaging step is just one of multiple possible steps executed by real-world build pipelines like the one at Jungle Juice Box. In their pipeline, they need to get their code, build it, test it, package it, and publish the resulting artifact. This pipeline must also be executed on a build server of some sort that already has the necessary software installed, such as Git and sbt. The pipeline also expects that certain environment variables are present in the execution environment. For more on the use of environment variables, see the discussion in chapter 4. Finally, the pipeline assumes that it's being executed on a Unix-like environment (for example, Ubuntu Linux). The next listing shows an approximation in shell script of how all of this comes together into a pipeline.

Listing 9.3 Build pipeline

```
#!/bin/sh

cd $PROJECT_HOME

git pull

sbt compile

sbt test
```

Navigates to the root project directory using an environment variable

Pulls down the latest version of the code

Compiles the project

Tests the project

```
sbt assembly

rsync -a $PROJECT_HOME/target/scala-2.11/fruit-picker-assembly-1.0.jar \
    username@artifact_server:~/jars/fruit-picker/$VERSION_NUMBER
```

Packages the project into a fat JAR

Publishes the JAR to a remote server using rsync under a version-numbered directory

Just to be clear, this is a simplified version of a build pipeline. Note that calling `sbt test` would itself invoke compilation, removing the need for the `sbt compile` step. Both steps are shown just for clarity about the steps in the process. The use of `rsync`, a Unix utility for copying data between locations such as remote servers, is a simple, if crude, approach. This publishing step uses the directory structure and a name to organize the different artifacts for different versions of the application, instead of a more sophisticated technique, like publishing to a Maven repository. If you know much about build pipelines, this one may seem crude, but even a simple build pipeline like this can be illustrative.

First, you have a step that's reserved for evaluation of your application, the `test` step. This is a great location to put any tests that evaluate models, similar to the techniques you saw in chapter 6. If any of those tests fail, the build pipeline will halt and not publish the application. Second, you're merely publishing at the end of the pipeline. As you've done several times before, you make the new artifact available for use, but you don't immediately change the state of the running application. Instead, you leave that decision for another system component.

9.4 Evaluating models

Now that you've built a minimal version of a build pipeline, let's consider how to make some decisions in the pipeline. In chapter 6, you worked through how to evaluate models and determine whether they should be used. Now you can bring those skills into play in the larger mission of building and deploying components of a machine learning system.

The Jungle Juicers, like the members of most machine learning teams, don't want to deploy broken functionality. In their case, the consequence would be bad—no predictions of which fruits their subscribers would want—so they've developed a range of safety mechanisms designed to keep their machine learning system stable. One of those is the model-evaluation step in their deploy process.

In this step, models are verified as being better than some standard before being deployed. If the models pass the test, then the application can be used in production. If not, this version of the application shouldn't be used, and production systems shouldn't be updated.

In chapter 6, I showed how to assess a model's performance relative to a random model. That's not the only approach you can use to determine whether a model is usable in your production system. Table 9.1 shows advantages and disadvantages of some of the alternatives.

Table 9.1 Model deployment criteria

Criterion	Advantages	Disadvantages
Better than random	Very unlikely to reject a useful model	A low bar for performance
Better than some fixed value	Can be designed to match business requirements	Requires an arbitrary parameter
Better than the previous model	Guarantees monotonically increasing performance	Requires accurate knowledge of the previous model's performance post-deployment

In the case of Jungle Juice Box, they plan to use a fixed value, 90% precision, to determine if a learned model should be deployed. The choice of 90% is arbitrary, but it matches well with their intuition about how models map to subscriber satisfaction with fruit recommendations. On the downside, manually performing this assessment and checking these values for every single learned model would be a laborious process, so let's look at how this step can be integrated into a more automated process.

9.5 Deploying

At this point, you may feel confident about your models based on the guarantees that the validations in the build pipeline give you. You could potentially move forward with deployment. In this context, *deployment* means to publish the component of your machine learning system and start acting on real user requests, as we explored in chapter 8. For your JJB team, the component system in this step looks something like figure 9.3.

After tests have passed, the application JAR is pushed to a remote artifact repository. Then the build pipeline calls the application-serving platform's API to start the

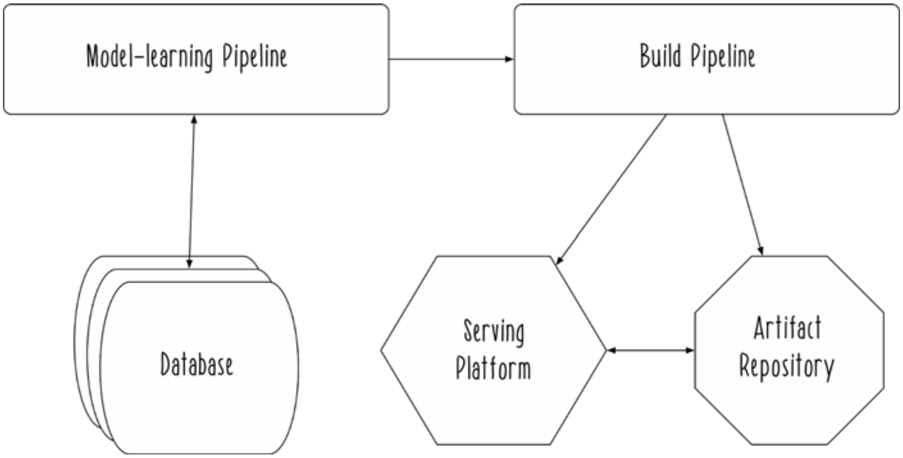


Figure 9.3 Model deployment

deployment of the application. The serving platform will need to provision the necessary resources, download the JAR to a sandbox, and then start the application.

How often should you deploy the system? What determines that you should do a deploy? This is actually a very complicated topic. Roughly speaking, there are four approaches to when you deploy and why, as shown in table 9.2.

Table 9.2 Approaches to deployments

Style	Criteria	Frequency	Advantages	Disadvantages
Ad hoc	None	Variable, but often infrequent	Simple and flexible	Deploys can be difficult due to low deployment skills and/or automation
At milestones	Achieving some meaningful development milestone	Weeks to months	Clarifies planning around deploys	Unplanned deploys can be hard
Periodically	Reaching a determined amount of time	Days to weeks	Regularity builds skills and speed	Can be labor intensive
Continuously	Committing to the master branch	Multiple times per day	Fast responses to change	Requires investment in pre-deploy enabling capabilities

As you can see, there are some complex trade-offs to consider when deciding how to structure your deployment process. The Jungle Juice Box team chooses a continuous deployment process, after using other processes earlier in the company's history. In their experimentations with variations on the approaches in table 9.2, they found that less-frequent deploys led them to underinvest in their build-and-deploy infrastructure. They didn't deploy that often, and when they did, it was so painful that they wanted to work on anything else, once the deploy was done, instead of improving the deploy process. When they decided that they needed to ship updates to the fruit-recommendation system more quickly, they realized they needed to implement the capabilities that would allow them to continuously deploy their system.

At a base level, they need reliable tests that will tell them whether their application can be deployed. Those tests need to be used by an automated build pipeline capable of determining whether the application should be deployed and then proceeding with that deployment.

They wind up with a flow that looks like figure 9.4, where a series of automated decisions are made to ensure that a given deployment is safe. Note that the predictive system's capability is tested at two levels. First, unit tests verify properties of the system that can be assessed without using much data. Then, after a deployable version of the application has been built, that release candidate is evaluated on a larger set of data. In this step, metrics about the system's performance are assessed to ensure that the system as a whole can do a sufficiently good job of its core mission: predicting

Supervision

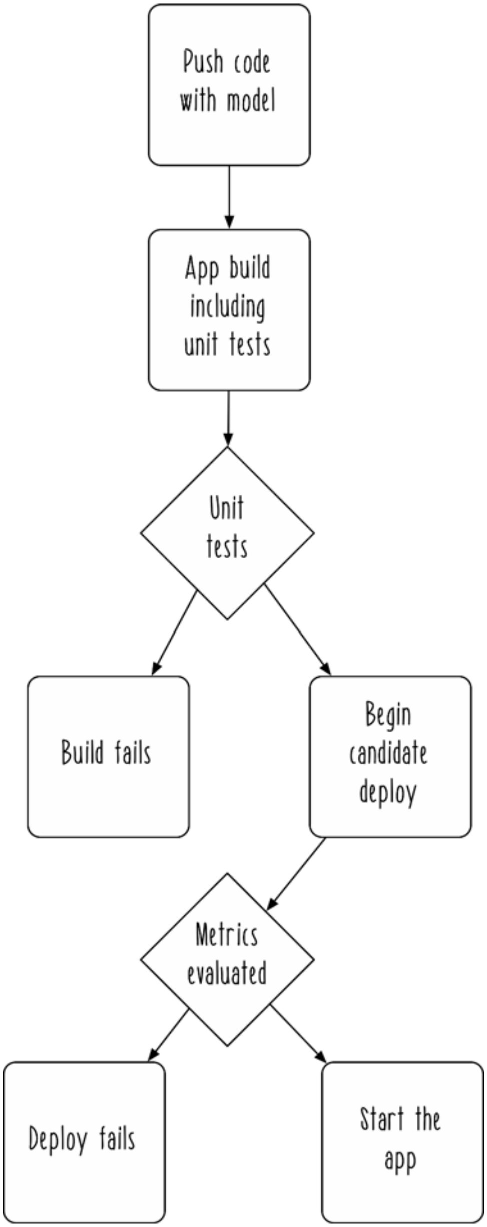


Figure 9.4 Automatic deployment of the fruit-prediction system

subscribers' fruit preferences. This particular technique is sometimes called a *metrics-based deploy*. Only when both levels of testing have been passed do you call the command to start the application.

With this approach to ensuring application safety in place, the Jungle Juice Box team gets to work in a powerful way. All commits to the master branch of the application's repository invoke a deploy, so they're happening all the time, usually many times a day. That means the fruit-prediction system is always reflecting pretty much the latest and greatest out of the JJB data team. When something goes wrong, the team is able to respond quickly and get the system fixed in a relatively short amount of time. Usually, nothing goes wrong—the tests are doing all the work of ensuring that things are fine, so the team does *other stuff* instead of worrying about whether the deploy went well. Mostly, they don't think about deploys at all and stay focused on the hard work of building better machine learning systems.

9.6 Reactivities

Here are a few reactivities to take you deeper into the jungle of building and deploying machine learning systems:

- *Try building your application with a different build tool.* Quite a few build tools target the JVM, like Ant, Maven, and Gradle. How does another build tool approach the task of building a deployable JAR?
- *Add validations.* Using either the sbt build in this chapter or your alternative build from the previous reactivity, add additional validations to your build. These validations could be any arbitrary "business logic" you want. For example:
  - Only build and deploy on weekdays.
  - Test-set performance must be higher than an arbitrary threshold.
  - No new dependencies have been added.
- *Write a deployment script and execute it regularly.* In this reactivity, you're not necessarily focused on the application's functionality, so your deploy can be something simple, like sending yourself an email that says that the deploy happened or using your computer's built-in voice to announce, "Deploy complete!" Once you implement your deployment script, ask yourself questions about its behavior:
  - What will happen if the deployment fails?
  - How could I "roll back" a deployment?
  - How long will deployments take? What functionality guarantees the responsiveness expectation?
  - What would happen if many different processes were running this deployment at the same time?

Summary

- Scala applications can be packaged into archives called JARs using sbt.
- Build pipelines can be used to execute evaluations of machine learning functionality, like models.
- The decision to deploy a model can be made based on comparisons with meaningful values, like the performance of a random model, previous models' performance, or some known parameter.

- Deploying applications continuously can allow a team to deliver new functionality quickly.
- Using metrics to determine whether new applications are deployable can make a deployment system fully autonomous.

Many of the techniques discussed in this chapter are defensive in nature—they protect you from the possibility of deploying a broken application. In the next chapter, we'll consider what happens when failure of your live system *does* happen.

10 Evolving intelligence

This chapter covers

- Understanding artificial intelligence
- Working with agents
- Evolving the complexity of agents

We've covered a lot of territory in our exploration of machine learning, but before we wrap up, we're going to expand our perspective just a bit more to consider the exciting world of artificial intelligence. To do that, we'll have to shrink down and explore one of the most complex societies in all of tech: bees who use instant messaging.

10.1 Chatting

The bees at Buzz Me have built one of the hottest apps in the insect world. They have literally trillions of users, all of whom spend a large portion of their working day coordinating with their hive mates via instant messaging. Although instant messaging has been around for millennia (in bee years), Buzz Me has recently taken over the market with a slick design that makes chatting about work as fun as setting up a night out with friends (figure 10.1).

As part of their expansion plans, the team at Buzz Me is developing a platform for bots (or agents). These bots will interact with their users in real time, taking on all sorts of simple, tedious, repetitive tasks (figure 10.2). Their plan for how to build these bots involves a sophisticated system that goes a bit beyond machine learning into the wider territory of artificial intelligence.

Figure 10.1 Buzz Me

10.2 Artificial intelligence

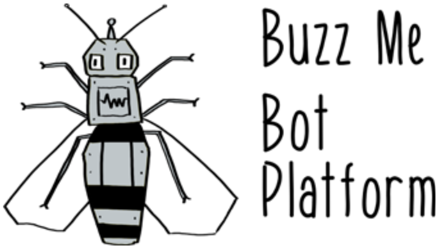


Figure 10.2 Buzz Me bot platform

Artificial intelligence (AI) is the overarching field of which machine learning is a part. In this book, we've already covered topics outside the range of what is usually thought of as machine learning. Typically, machine learning is concerned only with the process of a model learning from data and demonstrating that it's capable of performing better after the learning process, as discussed in chapters 4 through 6 of this book. Areas like data collection and responding to user input aren't well represented in traditional machine learning literature, but they are found in the broader discussions of AI. You could view *reactive AI* as the topic of this book.

The main reason I'll start to discuss things in AI terms is to talk about an AI concept called an *agent*. At its simplest, an agent is a software application that can act on its own. This chapter doesn't talk about the simplest forms of agents. Instead, it talks about AI agents that can sense data in their environment, make decisions about that data, and act on those decisions, within the environment—all while holding to the principles of reactive design.

Being capable of autonomous response to user input is the minimum requirement of the initial implementation of the bots on the Buzz Me platform. But as you join the bee team in this chapter, you'll see that the team and bot developers in general benefit from shooting for higher levels of functionality. Ideally, our agents will be capable of all sorts of useful things, like answering common questions, setting up meetings, and ordering office supplies. We may not be able to build bots as smart as bee bots, but we should easily be able to develop bots that are useful.

10.3 Reflex agents

To get your agent to do anything, you'll have to figure out how it decides what to do. The simplest way an agent can make decisions is through deterministic rules. An agent that operates in this manner can be called a *reflex agent*. Strictly speaking, lots of applications fulfill the nominal requirements for being a reflex agent, but here I'm only talking about those applications designed to operate as agents.

You'll begin building your agent as a simple reflex agent. This agent will eventually be a sort of insect buddy, with various likes and dislikes that users get to know through regular interactions. Eventually, your team wants the agent to be able to take on useful work, but they're beginning with small talk to help the user get accustomed to the idea of chatting with an agent. By introducing agents with some sort of "personality," the team believes users will become acclimated to agents as toys first and then later use them as tools.

To begin your implementation, the following listing gives the agent the capability to answer questions about its likes and dislikes as evidence of its "personality."

Listing 10.1 A reflex agent

```
object ReflexAgent {

  def doYouLike(thing: String): Boolean = {
    thing == "honey"
  }

}
```

Defines an agent as a singleton object for expedience

Expresses things to like as strings, and likes as Booleans

Defines an agent to like only honey

This reflex agent can now receive questions in the form of strings and respond with whether it likes them or not. In this case, the agent has extremely simple tastes and likes only honey. Even in this simple implementation, you can employ functional programming principles. Specifically, the input string is immutable data that's never changed, and the `doYouLike` function is a pure function, having no side effects. The next listing shows how this agent can be asked what it does and doesn't like.

Listing 10.2 Talking to a reflex agent

```
scala> import com.reactivemachinelearning.ReflexAgent
import com.reactivemachinelearning.ReflexAgent

scala> ReflexAgent.doYouLike("honey")
res0: Boolean = true

scala> ReflexAgent.doYouLike("flowers")
res1: Boolean = false
```

Imports the agent for use in this console session

true indicates the agent likes honey.

Asks whether the agent likes honey

Asks whether the agent likes flowers

false indicates the agent doesn't like flowers.

Your reflex agent certainly seems to work, although it's clearly very limited. But even simplistic reflex agents can be useful. For example, some agents developed for the Buzz Me bot platform are using text as a way of invoking commands. They don't need to contain any intelligence; they just need to send the user data back to some backend service and respond.

People have built agents that handle simple tasks like setting reminders and displaying random GIFs. Whether or not the developer thinks of their software as an agent, we can take that perspective even for such simple applications. But we're not interested in such simple agents. There's more interesting stuff we could do with more-intelligent agents.

10.4 Intelligent agents

The next level of complexity we'll consider is an agent that doesn't just do things, but actually *knows* things as well. If an agent is designed in such a way that it has a store of knowledge, then it can be called an *intelligent agent*. Thinking in terms of software, if a reflex agent is essentially a function, then an intelligent agent must be a function over some data other than the input arguments.

Throughout this book, we've been looking at machine learning systems that contain databases of facts. The knowledge component of an intelligent agent is more of the same. Ideally, the knowledge in an intelligent agent is implemented as an immutable log of facts. It would also be helpful if the facts in that database were expressed in an uncertainty-aware manner that acknowledged that various possible scenarios could be true at a given time.

But none of that is strictly required to fulfill your definition. For that matter, this database of knowledge doesn't even have to be a database. It can be a simple in-memory data structure, which is what you'll start with in the following listing.

Listing 10.3 An intelligent agent

```
class IntelligentAgent(likes: Set[String]) {

  def doYouLike(thing: String): Boolean = {
    likes.contains(thing)
  }

}
```

Constructs an intelligent agent with a set of likes

Still takes in things to like as a string and returns a Boolean result

Defines likability as a presence within a set of known things

This agent is only a bit more complex to implement than the previous one, but it meaningfully expands the possible range of functionality of the agent. Instead of having its likes fixed in the implementation of the agent itself, the agent's likes are now factored out to data that can be passed in at construction. This means you can instantiate an arbitrary number of such agents, with whatever likes you choose, rather than being constrained to a singleton, as you were with the reflex agent.

The next listing shows how to instantiate and interact with this intelligent agent.

Listing 10.4 Talking to an intelligent agent

```
scala> import com.reactivemachinelearning.IntelligentAgent
import com.reactivemachinelearning.IntelligentAgent
```

Imports an intelligent agent for use within a console session

```
scala> val aBeesLikes = Set("honey", "flowers")
aBeesLikes: scala.collection.immutable.Set[String] = Set(honey, flowers)
```

Defines a small set of things a particular bee might like

```
scala> val agent = new IntelligentAgent(aBeesLikes)
agent: com.reactivemachinelearning.IntelligentAgent =
➥ com.reactivemachinelearning.IntelligentAgent@2bdf3b2
```

Instantiates a new intelligent agent with those likes

```
scala> agent.doYouLike("honey")
res0: Boolean = true
```

The intelligent agent still likes honey.

```
scala> agent.doYouLike("flowers")
res1: Boolean = true
```

It also likes flowers.

```
scala> agent.doYouLike("birds")
res2: Boolean = false
```

It doesn't know about birds, so it doesn't like them.

This is just one possible configuration of the agent that you've created. You can continue to create more agents, each with likes appropriate for the given agent. At this level of complexity, you've already implemented as much intelligence in your agent as exists in many real-world products called *chatbots*. The agent has a fixed amount of knowledge that won't change, but in principle, you can fill it up with as much data as you have. Many chatbots don't have more capability than this simple intelligent agent. They may have much more knowledge in their implementation, in the form of canned responses to provide, but those responses don't change over time.

Developers on the Buzz Me platform have discussed using such capabilities for cases like simple customer service, where the intelligent agent retrieves the best answer from a set of FAQs. For such simple applications, machine learning isn't always required—user questions could be matched to answers from the FAQs using string-matching techniques alone. It's also possible to build entertaining chatbots for things like games and toys using this technique. In either case, the implementers of the agent ultimately make all the decisions about exactly what the agent will do in response in all cases. Intelligent agents are definitely more useful than reflex agents, but you can build even more powerful agents if you want.

10.5 Learning agents

Once you've built an intelligent agent, it's worth asking what its limitations are. Looking critically at your agent, it should be clear that it will always remain somewhat bound by the core agent function that you've implemented. Sure, it can continue to ingest new facts, but it will never really alter its behavior that much in response to new input. To see new capabilities emerge, you're going to have to put your machine learning skills to use and teach this agent how to learn.

Similar to the definition of machine learning itself, the definition of a *learning agent* is an agent that can improve its performance given exposure to more data. That sounds similar to what a intelligent agent is supposed to be, but there's an important difference. An intelligent agent that lacks learning capabilities is unable to alter the mapping of inputs (features) to outputs (concept labels). What we aspire to build in a learning agent is an agent that can get better merely through acquiring more data, just as we humans do.

To begin building your learning agent, let's go beyond raw strings and create some meaningful types. Your agents have been replying whether or not they like various things, so, in the following listing, let's call those replies *sentiments* and create case objects sharing a common sentiment type.

Listing 10.5 Sentiment types

```
object LearningAgent {

  sealed trait Sentiment {
    def asBoolean: Boolean
  }

  case object LIKE extends Sentiment {
    val asBoolean = true
  }

  case object DISLIKE extends Sentiment {
    val asBoolean = false
  }

}
```

Companion object to hold some learning-agent functionality

Defines a sealed trait as the basis for a sentiment type

Objects of this type have a Boolean representation.

Like sentiment represented as a true in Boolean

Dislike sentiment represented as a false in Boolean

If you're familiar with C++, Java, or C#, this implementation may remind you of the concept of *enumerations*. This `sealed` trait with its two implementations serves a similar function. Only the two implementations in this source file will be able implement the sentiment trait, making them the entirety of all members of the set of sentiment types. The `sealed` functionality will ensure that, even if you want to add future implementations, you'll be required to do it in this same source file.

With these types, you can now begin to build out a learning agent.

Listing 10.6 A simplistic learning agent

```
class LearningAgent {

  import com.reactivemachinelearning.LearningAgent._
```

Defines a learning-agent class

Imports types from a companion object

```
  val knowledge = new mutable.HashMap[String, Sentiment]()

  def observe(thing: String, sentiment: Sentiment): Unit = {
    knowledge.put(thing, sentiment)
  }

  def doYouLike(thing: String): Boolean = {
    knowledge.getOrElse(thing, DISLIKE).asBoolean
  }

}
```

Creates a modifiable collection of observations about things and their likability

Observes a thing and whether to like it

Records that observation in a data structure of knowledge

Takes things to like as a string and returns a Boolean

Checks for the presence of a thing in known likes and returns false if it's not known

This simple learning-agent implementation now has new capabilities that the previous agents didn't have: a changeable set of knowledge and an interface to take in new knowledge (the `observe` function). As modeled here, learning is the process of recording expressed sentiments that have been passed into the agent. Unlike the previous agents, this learning agent starts out with no likes or dislikes, but it can accumulate sentiments through the process of ingesting observations. The next listing shows an example interaction with the learning agent.

Listing 10.7 Talking to a simplistic learning agent

```
scala> import com.reactivemachinelearning.LearningAgent
import com.reactivemachinelearning.LearningAgent

scala> import com.reactivemachinelearning.LearningAgent._
import com.reactivemachinelearning.LearningAgent._

scala> val agent = new LearningAgent()
agent: com.reactivemachinelearning.LearningAgent =
➥ com.reactivemachinelearning.LearningAgent@f7247de

scala> agent.observe("honey", LIKE)

scala> agent.observe("flowers", LIKE)

scala> agent.doYouLike("birds")
res1: Boolean = false

scala> agent.observe("birds", LIKE)

scala> agent.doYouLike("birds")
res1: Boolean = true
```

Imports an agent

Imports a functionality in a companion object

Instantiates a new learning agent

Observes some common likes

The agent doesn't know about birds, so it doesn't like them.

Observes that the agent should, in fact, like birds

Now the agent likes birds.

As this session shows, the agent is capable of changing its sentiments over time, based on the data that it's seen. The agent knows exactly what the user has instructed it—

things the user likes or dislikes—and presumes that any new piece of data will be disliked.

An agent like this can be used for cases like accumulating user preferences or surveys. Changing its behavior is simple: expose it to more data. This agent doesn't *infer* much about the data that it's observed, though. The learning algorithms you've seen in this book attempt to generalize from a given set of data what the mapping is from features to concept labels/values. The simplistic learning agent in listing 10.7 doesn't do that, so let's whip up something that does. Given that we spent all of chapter 5 working through how to implement real learning algorithms, I'm going to show you an utterly silly one for a change of pace. In this case, you're going to build a classifier with an underlying algorithm based on inferences made according to the vowels within a word in a given observation. This isn't useful in the real world, but it should help make the actions of the machine learning model through the training process more explicitly comprehensible than is possible in a more complex model. You can see an example of how this works in listing 10.9, but you need to start by implementing the agent in the next listing.

Listing 10.8 A more complex learning agent

```
class LearningAgent {

  val learnedDislikes = new mutable.HashSet[Char]()

  def learn() = {

    val Vowels = Set[Char]('a', 'e', 'i', 'o', 'u', 'y')

    knowledge.foreach({
      case (thing: String, sentiment: Sentiment) => {
        val thingVowels = thing.toSet.filter(Vowels.contains)
        if (sentiment == DISLIKE) {
          thingVowels.foreach(learnedDislikes.add)
        }
      }
    }
    )
  }

  def doYouReallyLike(thing: String): Boolean = {
    thing.toSet.forall(!learnedDislikes.contains(_))
  }

}
```

Adds new functionality to an existing agent class

Dislikes are stored as vowel characters to dislike.

Function to invoke the learning process

Set of vowels to reference

Pattern matches on things and known sentiments about them

Iterates through all entries in the knowledge base

Finds vowels in a given thing to like

Determines whether the thing is disliked

If the item is disliked, add its vowels to a set of disliked vowels.

New function to access an alternative form of likability

Likes only things with no disliked vowels

Again, the knowledge in the agent is dynamic and can be changed via observation. But the API of this agent is a bit closer to some of the machine learning libraries you used in previous chapters. Specifically, it treats the learning of a model from observed data as a distinct step that must be invoked (via the `learn` method). The next listing shows how you interact with this agent.

Listing 10.9 Talking to a more complex learning agent

```
scala> val agent = new LearningAgent()
agent: com.reactivemachinelearning.LearningAgent =
➥ com.reactivemachinelearning.LearningAgent@61cc707b

scala> agent.observe("ants", DISLIKE)

scala> agent.observe("bats", DISLIKE)

scala> agent.doYouReallyLike("dogs")
res7: Boolean = true

scala> agent.doYouReallyLike("cats")
res8: Boolean = false
```

Creates a new agent

Sets up some observed dislikes

The agent generalizes from past observations that it would like dogs.

The agent generalizes from past observations that it wouldn't like cats.

This agent, even though it's never heard anything about dogs or cats, presumes it will like dogs and dislike cats. At this point, you have something that's truly using machine learning (even if the learning algorithm is silly). This is about where the traditional machine learning literature stops discussing the work of agent design. But in the real world, your agent might encounter more problems. Let's see how you might use reactive techniques to enhance the design of your agent.

10.6 Reactive learning agents

As you've done many times in this book, you're now going to take a basic design of a machine learning system and attempt to improve it through the application of design principles from reactive systems. Proceeding from those principles, let's ask questions about your current design.

10.6.1 Reactive principles

Is the agent responsive? Does it return sentiments to users within consistent time bounds? I don't see any functionality that guarantees much of anything in that respect, so let's answer that with a no.

Is the agent resilient? Will it continue to return responses to users, even in the face of faults in the system? Again, I see no functionality to support this property, so let's call that a no, as well.

How about elasticity? Will the agent remain responsive in the face of changes in load? It's not entirely clear that it will. So, again, we've got a no.

Finally, does the agent rely on message passing to communicate? This doesn't really seem to be the case either, so, no.

It looks like the agent pretty much fails our assessment. The agent isn't necessarily a *bad* design, but it doesn't attempt to provide the sorts of guarantees that we've been focused on in this book, so let's work on that.

10.6.2 Reactive strategies

Drawing from your toolbox of reactive strategies, let's try to use what you know to identify opportunities to improve the agent's design.

Looking at replication, are there ways to use multiple copies of the data to improve the reactivity of the agent? The store of knowledge is the primary bit of data, so that could be offloaded to an external distributed database. You could also replicate the agent itself, having more than one copy of your entire learning agent.

How about containment? Are there ways of containing any possible errors the agent might make? It seems likely that the agent could get some form of bad data, so if you introduced message passing, you could probably get greater containment of errors within the agent.

Lastly, how could supervision help out? Typically, supervision is most useful in terms of error handling or managing load. If the agent were replicable, it could be supervised, and then new agents could be spawned in the event of the failure of any given agent. Similarly, a supervisor could spawn new agents if the existing agents were insufficient for the load being experienced at the time.

10.6.3 Reactive machine learning

You haven't learned only general reactive principles and strategies in this book. Looking at the world through the lens of reactive machine learning, you've learned to appreciate the properties of data in a machine learning system.

Data in a machine learning system is effectively infinite in size and definitionally uncertain.

If you wanted to use laziness in your design, you could probably improve the responsiveness and elasticity of your system.

You're already using pure functions where appropriate, but you might look for more places to use them. The great thing about pure functions is that they work well with replication to handle arbitrary amounts of data.

Immutable facts are always a great approach for a store of machine-learned knowledge, and you're largely already using that approach. Observations made by the agent are never discarded or changed in any way.

And if you wanted to, you could add more sophistication to your design by considering the various possible worlds that might be true of the concepts that your machine learning system is attempting to model.

10.7 Reactivities

After a whole book of building reactive machine learning systems, you should now know more than enough to build something really great for these bees and their bot platform. I won't show you a particular solution. I'll leave that up to you as a final reactivity. The next couple of sections go into more detail about the dimensions that you can consider when you implement your bot platform. This reactivity is worthwhile to walk through, even if you only design but don't implement your solution, because many of the questions speak to high-level architectural issues in your design.

10.7.1 Libraries

You've used various libraries/frameworks/tools in this book. Often, those libraries have given your applications properties that would be laborious to implement otherwise. In the case of this bot platform, are there libraries that might help you make this system more reactive?

Let's start with Spark. In this book, you've mostly used Spark as a way of building elastic, distributed, data-processing pipelines, but that's not all it can do. Spark is generally a great tool for building distributed systems, not just batch-mode jobs. You could certainly hold the agents in your system inside Spark data structures. That would allow you to use the strategy of replication.

Keeping your agent data distributed throughout a cluster should help with elasticity, because requests to agents can be served from multiple nodes in the cluster. Similarly, Spark's built-in supervision capabilities can help with resilience.

If a node in the cluster goes down, the Spark master won't send it work and may potentially bring up new nodes, depending on how your implementation works with the underlying cluster manager.

Useful as Spark is, it's not the only tool in your toolbox. Akka has many of the same strengths—as you might expect, because Spark used Akka internally in earlier versions of the library. An Akka implementation of a bot platform might be more natural in some ways. You could model agents as actors, which are somewhat similar concepts; an *actor* is like an agent that uses only message passing as its form of actuation. But as you've seen, message-driven applications can have really great properties.

Thanks to a message-driven design, an Akka implementation could easily contain the errors of agents on the platform. There's no reason why errors in a given agent should contaminate another agent if both are modeled as distinct actors. In this way, Akka actors aren't too different from the model microservices you built in chapter 7.

All actor systems are organized around supervisory hierarchies. The benefit of this is that the supervisory actors can take actions to improve the elasticity and resilience of the system by spawning new actors in cases of high load or killing actors that are misbehaving.

Of course, it's great to not have to design how all these actors compose by using libraries like Akka HTTP. Despite the power and flexibility of Akka, it abstracts all sorts of complexity in system design, allowing you to minimize the amount of focus that you spend on things like message-passing mechanics and how to manage supervision trees.

10.7.2 System data

Finally, let's look at the data in your system and see what design choices can be made. First, if you presume that your data is effectively infinite in scale, then how should that influence the design of the system?

Typically, that implies that you're building a distributed system. You've spent a fair bit of time on Spark and Akka in this book, and they can both be used to build highly reactive distributed systems. But this concern about data scale isn't just about data *processing*; it's relevant to data *storage* as well. As discussed in chapter 3, there are lots of reasons to ensure that the backing data store for your system is a highly replicated distributed database of some kind. Your options include self-hosted databases like Cassandra, MongoDB, and Couchbase as well as cloud-native databases provided as services like DynamoDB, Cosmos DB, and Bigtable. All the databases just mentioned (and too many more to enumerate) use techniques like replication and supervision to ensure elasticity, resilience, and responsiveness. There's not one good choice; there are many. But there's no need to begin your design with a traditional nondistributed relational database. Better ways of building systems are available via simple API calls to cloud vendors. That's not to say that you shouldn't consider using the relational model for your data, but if you do, definitely consider using a distributed relational database like Spanner or CockroachDB.

While you're thinking about the consequences of effectively infinitely sized datasets, let's think some more about how you can use other tools in your toolbox. For example, how are you going to design a development workflow that allows you to iterate on system design locally while still maintaining parity with a large-scale production deployment?

As you've seen before, one technique you can use is laziness. For example, if you compose your feature-generation and model-learning pipeline as a series of transformations over immutable datasets using Spark, then that pipeline will be composed in a lazy fashion and executed only once a Spark action has been invoked. You used this method of pipeline composition extensively in chapters 4 and 5.

Similarly, you've already seen lots of ways to use pure, higher-order functions as ways of implementing transformations on top of immutable datasets. As you've seen in several chapters, the use of pure functions enables various techniques for dealing with arbitrarily sized datasets. Where can you use pure functions in your system implementation? You've certainly seen how pure functions can be used in feature generation. In your bot platform implementation, does it make sense to have models themselves be functions? For example, could you refactor listing 10.6 to structure likes using pure functions?

Let's also think about the certainty of your data. Throughout this book, you've taken the approach that data in your machine learning system can't be treated as certain—that all data in a machine learning system is subject to uncertainty. Instead of treating the concept of sentiment as a Boolean, it could instead be modeled as a level of confidence in positive sentiment, along the lines of the following listing.

Listing 10.10 Uncertain data model for sentiments

Defines a sealed trait to structure different sentiment levels

```
object Uncertainty {

  sealed trait UncertainSentiment {
    def confidence: Double
  }

  case object STRONG_LIKE extends UncertainSentiment {
    val confidence = 0.90
  }

  case object INDIFFERENT extends UncertainSentiment {
    val confidence = 0.50
  }

  case object DISLIKE extends UncertainSentiment {
    val confidence = 0.30
  }

}
```

Requires all uncertain sentiments to have a confidence level

Instance of an uncertain sentiment, representing a strong like sentiment

Strong like sentiment modeled as 90% confidence of a positive sentiment

Indifferent sentiment modeled as 50% confidence of a positive sentiment

Dislike sentiment modeled as 30% confidence of a positive sentiment

This listing is a sketch of a simplistic way of encoding some uncertainty within the data model. A more sophisticated approach would likely involve calculating the level of confidence for any given sentiment prediction, as you've seen previously in the book.

By modeling your data as uncertain, you open the door to reasoning about the range of possible states that the concept being modeled could be in. How could your system design evolve to incorporate this style of reasoning? A given agent could return some of this uncertainty to the user by returning the top *N* results by confidence. Or, if multiple agents could be addressed to perform a given task for a user, then the Buzz Me bot platform could develop its own models of confidence in each and every agent. Then the supervisory component (which might itself be modeled as an agent) could dynamically choose which agent would be best suited to fulfill a given user task based on its confidence level in each agent, as in figure 10.3.

With all these questions and tools in mind, you could now build a pretty sophisticated solution for AI agents that converse with insects via instant messaging.

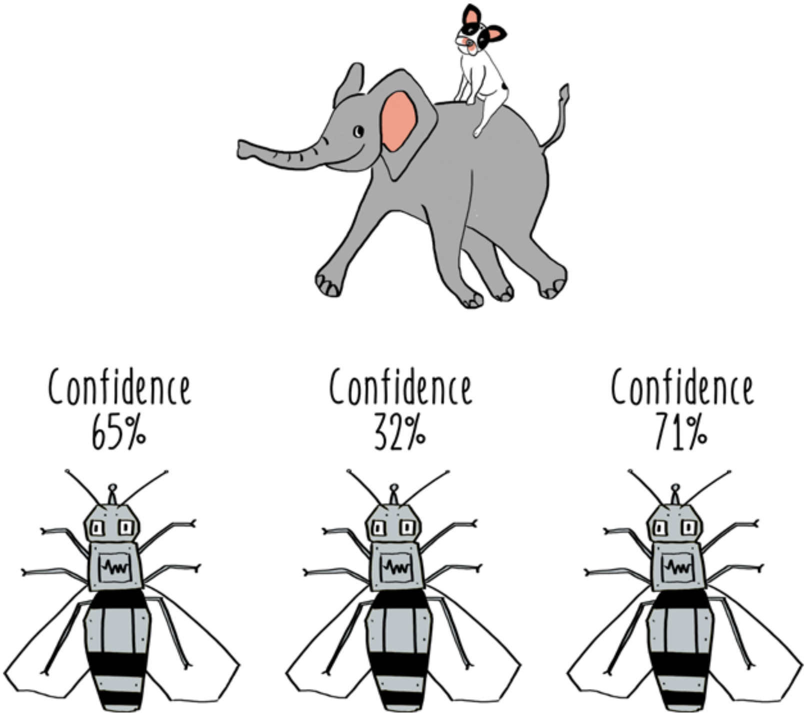


Figure 10.3 Agent supervision

10.8 Reactive explorations

At the end of each chapter, I've asked you to go out and apply the concepts of reactive machine learning to new challenges via the reactivities. This section explores how you can take others on a journey through the use of reactive techniques using a tool I like to call a *reactive exploration.*

In a reactive exploration, you ask questions about an existing system or component, examining it with its implementers/maintainers. You could start this exploration by just dropping a copy of this book on someone's desk and telling them to read it all before you talked—or you could try to ease into the topic by having a more general conversation.

10.8.1 Users

I like to begin by trying to figure out who the user is. That question can be trickier than it sounds. The user isn't always the literal customer of the company. For many machine learning components, the user is some other developer or team that relies on the machine learning system to perform useful functionality. One way to get at firm agreement on who this person is to ask, "Who would care if we all stopped coming in to work?" Once you've got that person characterized, you can place them on the board, using a cartoon animal or some other representation (figure 10.4).

Then you need to establish how this user interacts with your system. Specifically, you want to identify all the components of a request-response cycle. Examples of a request-response cycle could include the following:

- For an ad-targeting system, the user could send a request for an ad along with some browser data and get back the ID number of the ad to show.
- For a spam filter, the user could send an email and get back a classification as spam or not spam.
- For a music-recommendation system, the user could send a subscriber's listening history and get back a list of recommended songs.

Figure 10.4 **An unhappy user of a non-reactive machine learning system**

10.8.2 System dimensions

If you've properly defined it, this request-response cycle is the basis of the commitment your system has with its users. That allows you to ask questions motivated by reactive design principles without first having to introduce all the terminology used in this book and other discussions of reactive.

Here are four dimensions that you can ask questions about for a given system. First, you can ask questions about *response time* in the system:

- When will this system return responses to the user?
- How quickly does the user expect a response?
- How much will that response time vary?
- What functionality within the system is responsible for ensuring that the system responds within that time?
- What will happen on the user's end if the response isn't returned within the time expectation?
- Do you have any data around what real response times are?
- What would happen if the system returned responses instantaneously?

Next, you can ask questions about the behavior of the system under *varying levels of load*:

- What sort of load do you expect for this system?
- What data do you have about past historical load?
- What if the system was under 10 times the load you expect? 100 times? More?
- What would the system do under no load?
- What sort of load would cause the system to not return a response to a user within the expected time frame?

After that, you can move on to questions about *error handling*:

- What are some past bugs that this system has experienced?
- What behavior did the system exhibit in the presence of those errors?
- Have any past errors caused the system to violate the expectations of the user in the request-response cycle?
- What functionality exists within the system to ensure that errors don't violate user expectations?
- What external systems is the system connected to?
- What sorts of errors could occur in those external systems?
- How would this system behave in the presence of those errors from external systems?

Finally, you can ask about the *communication patterns* within the system:

- If one part of the system is under high load, how is that communicated?
- If an error occurs in one part of the system, how is that communicated to other components?
- Where do the component boundaries exist within the system?
- How do the components share data?

10.8.3 Applying reactive principles

For the attentive reader, the four dimensions of system behavior in the previous section should have sounded very familiar. They're restatements of the reactive principles that you've been using all through this book, as you can see in table 10.1.

Table 10.1 Mapping from system dimensions to reactive principles

System dimension	Reactive principle
Time	Responsive
Load	Elastic
Error	Resilient
Communication	Message-driven

Done with a legitimate curiosity about the behavior of the machine learning system, this exercise should leave you with a lot of interesting follow-up questions to try to answer. Very often, you won't really know how a system will behave under certain conditions, and you won't be able to point to any functionality responsible for ensuring that the system fulfills user expectations in a given scenario. That gives you the opportunity to figure out how to apply all the tools and techniques that you've learned in this book, guided by the user needs that you uncovered in the reactive exploration.

Summary

- An agent is a software application that can act on its own.
- A reflex agent acts according to statically defined behavior.
- An intelligent agent acts according to knowledge that it has.
- A learning agent is capable of learning—it can improve its performance on a task given exposure to more data.

That's the end of the book. I've shown you all that I can. Now it's your turn to show me just how amazing the machine learning systems you build will be. Happy hacking!

appendix *Getting set up*

Scala

Almost all of the code in this book is written in Scala. The best place to figure out how to get Scala set up for your platform is the Scala language website (www.scala-lang.org), especially the Download section (www.scala-lang.org/download). The version of Scala used in this book is Scala 2.11.7, but the latest version of the 2.11 series should work just as well, if there is a later version of 2.11 when you're reading this. If you're already using an IDE like IntelliJ IDEA, NetBeans, or Eclipse, it will likely be easiest to install the relevant Scala support for that IDE.

Note that all of the code provided for the book is structured into classes or objects, but not all of it needs to be executed that way. If you want to use a Scala REPL or a Scala worksheet to execute the more isolated code examples, that will generally work just as well.

Git code repository

All the code shown in this book can be downloaded from the book's website (www.manning.com/books/reactive-machine-learning-systems) and also from GitHub (https://github.com) in the form of a Git repo. The *Reactive Machine Learning Systems* repo (https://github.com/jeffreyksmithjr/reactive-machine-learning-systems) contains a project for each chapter. If you're unfamiliar with version control using Git and GitHub, you can review the bootcamp articles (https://help.github.com/categories/bootcamp) and/or beginning resources (https://help.github.com/articles/good-resources-for-learning-git-and-github) for learning these tools.

sbt

This book uses a wide range of libraries. In the code provided in the Git repo, those dependencies are specified in such a way that they can be resolved by sbt. Many Scala projects use sbt to manage their dependencies and build the code. Although you don't have to use sbt to build most of the code provided in this book, by installing it you'll be able to take advantage of the projects provided in the Git repo and some of the specific techniques around building code shown in chapter 7. For instructions on how to get started with sbt, see the Download section (www.scala-sbt.org/download.html) of the sbt website (www.scala-sbt.org). The version used in this book is sbt 13.9, but any later version in the 13 series should work the same.

Spark

Several chapters of this book use Spark to build components of a machine learning system. In the code provided in the GitHub repo, you can use Spark the way you would use any other library dependency. But having a full installation of Spark on your local environment can help you learn more. Spark comes with a REPL called the *Spark shell* that can be helpful for exploratory interaction with Spark code. The instructions for downloading and setting up Spark can be found in the Download section (http://spark.apache.org/downloads.html) of the Spark website (http://spark.apache.org). The version of Spark used in this book is 2.2.0, but Spark generally has a very stable API, so various versions should work nearly identically.

Couchbase

The database used in this book is Couchbase. It's open source with strong commercial support. The best place to start with getting Couchbase installed and set up is the Developer section (http://developer.couchbase.com/server) of the Couchbase site (www.couchbase.com). The free Community Edition of Couchbase Server is entirely sufficient for all the examples shown in this book. The version used in this book is 4.0, but any later version of the 4 series should work as well.

Docker

Chapter 7 introduces how to use Docker, a tool for working with containers. It can be installed on all common desktop operating systems, but the process works differently, depending on your OS of choice. Additionally, the tooling is rapidly evolving. For the best information on how to get Docker set up on your computer, visit the Docker website: www.docker.com.

index

MACHINE LEARNING/SOFTWARE ENGINEERING

Machine Learning Systems

Jeff Smith

If you're building machine learning models to be used on a small scale, you don't need this book. But if you're a developer building a production-grade ML application that needs quick response times, reliability, and good user experience, this is the book for you. It collects principles and practices of machine learning systems that are dramatically easier to run and maintain, and that are reliably better for users.

Machine Learning Systems: Designs that scale teaches you to design and implement production-ready ML systems. You'll learn the principles of reactive design as you build pipelines with Spark, create highly scalable services with Akka, and use powerful machine learning libraries like MLib on massive datasets. The examples use the Scala language, but the same ideas and tools work in Java, as well.

What's Inside

- Working with Spark, MLlib, and Akka
- Reactive design patterns
- Monitoring and maintaining a large-scale system
- Futures, actors, and supervision

Readers need intermediate skills in Java or Scala. No prior machine learning experience is assumed.

Jeff Smith builds large-scale machine learning systems using Scala, Akka, and Spark.

"This book doesn't just cover tools; it covers the whole job of building an entire machine learning system."

—From the Foreword by Sean Owen
Director of Data Science, Cloudera

"A helpful guide for data engineers building resilient machine learning systems."

—Jonathan Woodard, AT&T

"A fantastic entry to the world of robust machine learning systems that will scale with your business."

—Tommy O'Dell
Virtual Gaming Worlds

"You cannot afford to ignore this book!"

—José San Leandro, OSOCO

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit manning.com/books/machine-learning-systems

ISBN-13: 978-1-61729-333-7
ISBN-10: 1-61729-333-4
54499
9 781617 293337

MANNING

$44.99 / Can $59.99 [INCLUDING eBOOK]