O'REILLY®

Compliments of IBM

Data Science and Engineering at Enterprise Scale

Notebook-Driven Results and Analysis

Jerome Nilmeier, PhD

Data Science and Engineering at Enterprise Scale

Notebook-Driven Results and Analysis

Jerome Nilmeier, PhD

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

Data Science and Engineering at Enterprise Scale
by Jerome Nilmeier, PhD

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://oreilly.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Editors: Rachel Roumeliotis and Michele Cronin
Production Editor: Kristen Brown
Copyeditor: Rachel Monaghan

Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Rebecca Demarest

February 2019: First Edition

Revision History for the First Edition
2019-02-05: First Release

978-1-492-03931-0

[LSI]

For Sofia, David, and Charlie.
Thank you for giving me purpose, joy, and love.

Table of Contents

Foreword

Recently I helped create an upskilling curriculum for data science. It was aimed at people already in the industry with some tech background, though not in big data—those who didn't have months to spend, but needed hands-on experience to get started. Our team debated which technologies would be most important for people to learn, given time constraints in the course. Jupyter, Docker, Spark, and TensorFlow each came up as key technologies. We started building examples to tie the tools together into practical workflows, plus a sampler of machine learning, data visualization, and associated topics.

At about the same time, I saw a preview of this book. "Hey, that's it!" we recognized. "That's just the right mix of tools, techniques, and real-world examples."

I met Jerome years ago while guest lecturing for a data engineering fellowship. We were focused on Apache Spark in that course. Although other components described here—Jupyter, Docker, Anaconda, deep learning, vector embedding, etc.—existed at the time, it wasn't clear how they'd evolve and become important together. Later Jerome adapted IBM training materials for Spark to use in a training program at O'Reilly where we were both teaching. It's been a pleasure to see him grow in this field.

Jerome has a passion for education and developer advocacy that shows throughout the pages here. Beyond demonstrating these open source technologies, he enjoys showcasing them in context, giving people tools they need to succeed in their work.

This book is an easy read—for example, you can take it along on a flight. I especially like how notebooks get used. They serve as containers for most of the technical details, organized as scaffolding: run them at first to get familiar and see the big picture, then dig into details on later iterations through the code. Moreover, these notebooks provide examples of how you'll be collaborating on data science teams, delivering insights in enterprise.

That's the point about learning data science: you'll continue to learn and grow in your practice, as these popular tools continue to evolve and as your team continually adapts to business needs. But get started now, and head in the right direction with *Data Science and Engineering at Enterprise Scale*.

— *Paco Nathan*
Derwen, Inc.

Preface

What This Book Will Cover and How It Will Help You with Your Daily Work

While the notebook is generally the domain of the data scientist, this book is intended for anyone who will be interacting at varying levels with the code, from developers and engineers who will be working with codebases and deployments, to management teams who wish to understand some of the issues facing the team.

We will provide examples in open source Jupyter notebooks, along with installation instructions for desktop-scale studies.

We will also provide examples in IBM Watson Studio, which is a cloud-hosted notebook environment that is also scalable. This service is free for smaller-scale studies, and it has a rich collection of examples that can be run with no installation requirements. This environment also offers enterprise-grade collaboration tools for sharing code and data conveniently and securely.

The book will cover:

- Notebooks
  - — Notebooks and the Jupyter ecosystem
  - — Language examples (Python, Scala)
  - — IBM Watson Studio
- Using machine learning frameworks in a notebook
  - — Installation, architectures
  - — Scalable machine learning (Apache Spark)

— Deep learning frameworks

- Analytics in the production environment
  - — Implementation issues
  - — Collaboration across the enterprise

All examples will be made available in the IBM Watson Studio as well as on GitHub with an associated Docker image.

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic
: Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`
: Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

`Constant width bold`
: Shows commands or other text that should be typed literally by the user.

`Constant width italic`
: Shows text that should be replaced with user-supplied values or by values determined by context.

This element signifies a general note.

Using Code Examples

Supplemental material (code examples, exercises, etc.) is available for download at *https://github.com/nilmeier/DSatEnterpriseScale*.

This book is here to help you get your job done. In general, if example code is offered with this book, you may use it in your programs

and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Data Science and Engineering at Enterprise Scale* by Jerome Nilmeier, PhD (O'Reilly). Copyright 2019 O'Reilly Media, Inc., 978-1-492-03931-0."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at *permissions@oreilly.com*.

O'Reilly

For almost 40 years, *O'Reilly Media* has provided technology and business training, knowledge, and insight to help companies succeed.

Our unique network of experts and innovators share their knowledge and expertise through books, articles, conferences, and our online learning platform. O'Reilly's online learning platform gives you on-demand access to live training courses, in-depth learning paths, interactive coding environments, and a vast collection of text and video from O'Reilly and 200+ other publishers. For more information, please visit *http://oreilly.com*.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North

Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

To comment or ask technical questions about this book, send email to *bookquestions@oreilly.com*.

For more information about our books, courses, conferences, and news, see our website at *http://www.oreilly.com*.

Find us on Facebook: *http://facebook.com/oreilly*

Follow us on Twitter: *http://twitter.com/oreillymedia*

Watch us on YouTube: *http://www.youtube.com/oreillymedia*

Acknowledgments

In no particular order, for their various contributions, assistance, and/or inspirations:

Justin McCoy, Brendan Dwyer, Vijay Bommireddipalli, Carmen Rupach, Susan Malaika, Sourav Mazumder, Stacey Ronaghan, Dean Wampler, Paco Nathan, William Roberts, Michele Cronin, Steve Beier, Colleen Lobner, Nicole Tache, Anita Chung, Lori Vella, Katherine Krantz, Edward Leardi, Rachel Monaghan, and Kristen Brown.

CHAPTER 1

Sharing Information Across Disciplines in the Enterprise

> *Programs are meant to be read by humans and only incidentally for computers to execute.*
>
> —Donald Knuth

This chapter will introduce you to the challenges of communicating ideas across multidisciplinary teams. While teams often have much in common in terms of skills and objectives, they may be composed of people from vastly different educational and cultural backgrounds, who bring different perspectives to bear on the same problem. In these environments, it is important to share information in a clear and consistent way. Notebooks provide an excellent way to do this, as they combine live code with formatted text so that programmers, data scientists, and even nontechnical members of the team can understand what is happening with various elements of the code being used.

The Overlap Between Data Scientist and Data Engineer

The modern data scientist on an enterprise team often has an intellectual ancestry in the academic world. The standard workflow in academic research is to measure something, compare the result to the predicted one, and report the findings in a peer-reviewed environment. The assumption in this environment is that "if you didn't

publish it, it didn't happen," which places a very heavy emphasis on careful documentation of work as the measure of success. It is not enough, however, to document and present your findings. As a data scientist, you must also be prepared to defend your position and persuade skeptics. This process requires diligence and determination if your idea is to be embraced.

On the other hand, for modern enterprise developers and engineers working in a fast-paced environment, the emphasis is on delivering code that provides the functionality required for the company's success. The process of reporting findings is not typically as highly valued, and documentation is often considered a necessary evil that only the more diligent developer is committed to maintaining. Tracking progress is tied more to performance measures for time management than to explaining your reasoning and design choices. Furthermore, an aesthetic of compactness and brevity is more highly valued in a mature codebase. This more terse style, however, may be more difficult to read without additional documentation or explanation.

How, then, do we reconcile the two approaches in a coherent way? The data scientist may have a question about an algorithm that could affect performance, and will want to run tests. How do these tests translate into useful code? How does the data scientist persuade the development team that these tests open a path to a useful solution?

Conversely, how can an engineer or developer explain some of the more elegant but difficult-to-read pieces of code to a data scientist without creating unnecessarily verbose descriptions in the codebase?

Finally, how can management figure out what on earth their team is up to, beyond using a ticketing system (such as JIRA or GitHub Issues)?

Enter the notebook.

How Notebooks Bridge the Gap

The notebook is simply a collection of cells that run small snippets of code interactively. It also has features that allow you to display images, plot graphs, and display formatted text. This simple format allows you to provide a narrative structure around your code that

enables you to describe the thinking behind it, while also providing all the necessary machinery to run it and explore the output. A simple notebook is shown in Figure 1-1.

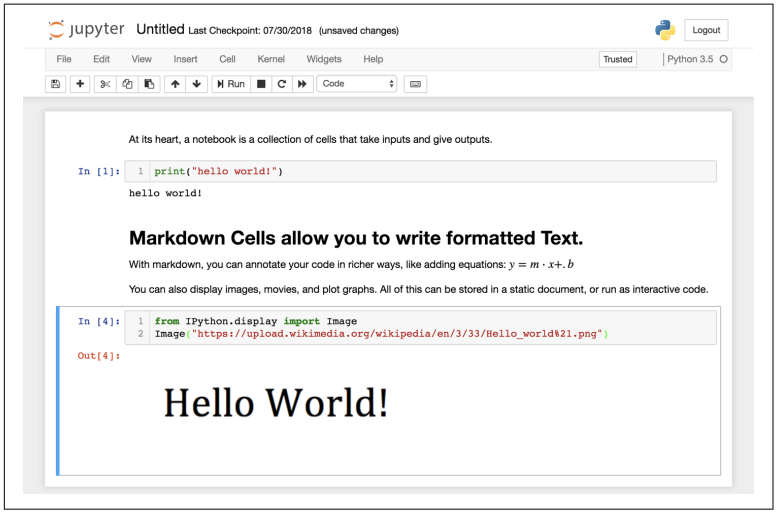

Figure 1-1. A notebook is a living document

The notebook environment has its heritage in more academic codes, including R, MATLAB, and Mathematica. Jupyter Notebook, for example, most closely resembles the Mathematica environment. In these environments, the intent is to provide a fully functional numerical engine that is exploratory in nature. Graphical capabilities and variable exploration are fundamental to the design. In the early stages of a notebook, the environment provides a low barrier to entry and the ability for users to quickly understand functions, numerical properties, and other questions that can be cumbersome to explore in debugger environments.

In a later phase of the notebook, the functionality shifts more decidedly from exploratory to expository. The notebook is no longer a scratch pad, but a means of communicating your idea, discovery, or algorithm to your coworkers. The graphs you were using to understand the data can now be used to make your case to the reader. The notebook can be presented as a static document or as a living piece of code that can be run and explored, even when connected to a cluster or multiple APIs.

Notebooks as a Medium of Communication

A well-written notebook or two in a large code repository provides a quick overview of what the code can do. While a *README.md* file often contains the details of installation and running a "Hello World" can verify the installation, a notebook can give deeper understanding of the code's capabilities by providing formatted text and figures. A person who wants to quickly run examples can review a notebook tutorial, run the code, and be up to speed in minutes. This has important implications for code adoption. It is often said that adoption of code depends on whether the user can run examples within the first few minutes of interacting with it. If the code cannot be run within this time, the user may lose interest and never return.

The notebook is designed to engage not only the newcomer who wishes to learn to use the code, but also someone who is nontechnical but is nonetheless involved in evaluating the codebase. For these audiences, the persuasive element of the notebook is more critical, as the important ideas are illustrated with examples, descriptions, and figures.

Finally, the notebook may be used simply to educate members of your team on a particular algorithm that was developed and implemented. As shown in Figure 1-2, notebooks can be both generated and consumed by various members of the team in order to share interesting aspects of their work with one another.

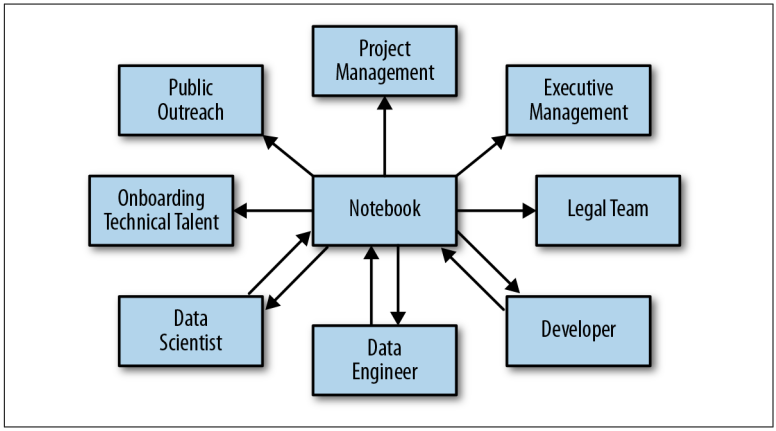

Figure 1-2. A notebook can be used to communicate to many different people within the enterprise ecosystem

Example: Validating Statistical Functions and Developing Unit Tests

For this example, we will be writing and running Python code. The code can be run either at the command line or in a notebook environment. The notebook for this example is provided at *https://github.com/nilmeier/DSatEnterpriseScale/blob/master/multinomialSampler.ipynb*.

Imagine that you are a data scientist who works in a data center, and you have been asked to help write code that will simulate process failures on a large cluster of computers. Each machine in your cluster will have a number of processes running, and each process will have an estimated failure probability. To create a sampler that will sample a number of failures over a time period, you determine that you will need to sample from a multinomial distribution. You want to also write this sampling procedure to run at a very large scale (both in number of processes and in time duration) and compute lots of statistics.

The multinomial distribution is given as:

$$f(x_1, \ldots, x_N) = \frac{n!}{x_1! \cdots x_k!} p_1^{x_1} \cdots p_k^{x_k}.$$

After looking through the Apache Spark documentation, you learn that it is not implemented as one of the standard random number generators. Fortunately, it is not too difficult to implement (which may explain why it is not in the core library), and you can contribute to your team project by offering a multinomial sampler that can be used at scale.

While our example is somewhat simplified, it is very typical of a scale-up process. It is often the case that an algorithm that runs well at desktop scale will not be trivial to run at large scale. Fortunately, with a framework like Apache Spark, writing scalable code is not as difficult as you might expect.

Evaluating a Validated Desktop-Scale Function

Our example may have a real-world application, but we are going to use a more accessible example for the multinomial distribution: the

six-sided die. The built-in function for generating samples from a multinomial distribution is `np.random.multinomial`. The method is easy to call as shown here:

```
import numpy as np
nTrials = 500 # of rolls per round
nRounds = 100 # of rounds
np.random.seed(10)
numFaces = 6
p = [1/6.]*numFaces
s = np.random.multinomial(nTrials, p1, size=nRounds)
```

If you were running this at your terminal, you could verify that the first few samples are given as:

```
>> print(s[1:5])
[[70 86 92 91 65 96]
 [81 75 78 81 91 94]
 [78 91 91 89 73 78]
 [93 77 80 81 80 89]]
```

This is a pseudorandom sequence. It can be repeated if the same random seed is used. We can only truly understand the properties of this sampler, however, by taking many samples and observing the convergent behavior. How many rolls of the die does it take to converge to the expected roll frequency? What are the error bars? Sometimes these questions can be answered with theories, but they are often best answered by generating many samples.

We can generate these samples in a notebook and produce a plot like Figure 1-3. This plot shows the number of counts for each die value rolled, normalized by the total number of counts. The error bars are the standard deviation of the number of counts accumulated at each round, for a total of `nRounds`. Each round consists of a number of trials, `numTrials`, where an individual sample is drawn from the multinomial distribution.

The great thing about the notebook is that the plots and output data will stay cached in it for future reference; this means that we can refer to the static notebook without having to regenerate the data. For statistical sampling, this is particularly helpful, because we always rely on a large number of samples to improve our estimates.

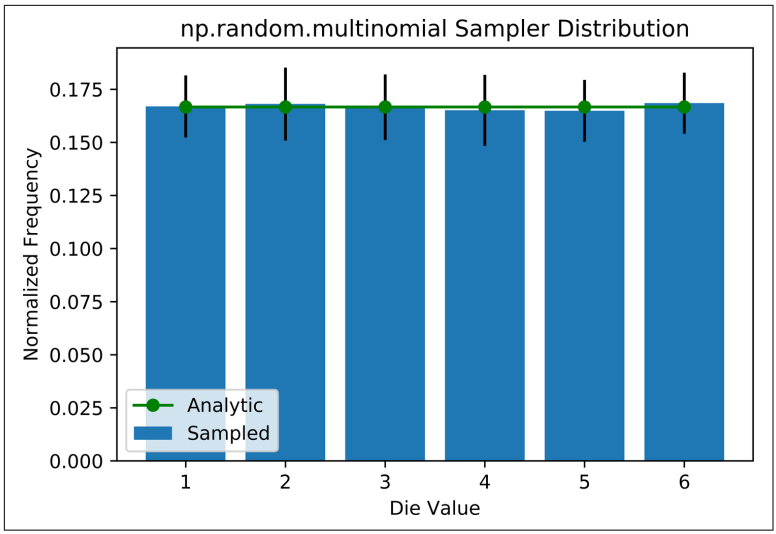

Figure 1-3. Histogram of averages and error bars for the multinomial distribution (a fair six-sided die)

Understanding the Logic to Be Used at Scale

We want to write our scalable function to improve our estimates. Let's not be too hasty, however. We have a good understanding of the outputs of the NumPy built-in function, so let's write our own function that returns values in the same way that the built-in function does. We can compare the outputs from the two functions to verify that our thinking is correct about how the sampler works. We don't know exactly what this will look like in our scalable code (PySpark, Apache Spark's Python API), but we do know that we will have a uniform random number generator at our disposal. So, let's think about how we would write a multinomial sampler that leverages a uniform random number generator.

Referring to Figure 1-4, we can briefly describe the procedure. A random number ξ is generated from a uniform random number generator. Each die face has some probability of being selected, so we compare our random number to the accumulated probability value computed for each die value to see which die face to select. The code for accomplishing this is as follows:

```
def multinomialLocal(nTrials, p, size):

    nRounds = size # using a more descriptive variable
    xi = np.random.uniform(size=(nTrials, nRounds))

    # computing the cumulative probabilities (cdf)
    pcdf = np.zeros(numFaces)
    pcdf[0] = p[0]

    for i in range(1, numFaces):
        pcdf[i] = p[i-1] + pcdf[i-1]

    s = np.zeros((nRounds, numFaces))

    for iTrial in range(nTrials):
        for jRound in range(nRounds):
            index = np.where(pcdf >= xi[iTrial, jRound])[0][0]
            s[jRound, index] += 1

    return s
```

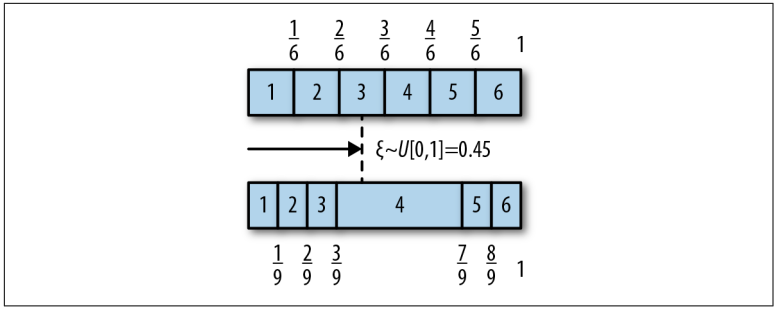

Figure 1-4. Method for selecting a die value for a six-sided die using a uniform random number generator; the top bar shows a fair die and the bottom die shows a biased die

While this is not too difficult to read, it is much easier to understand the intent of the function by looking at the figure, which explains with much more clarity the essential sampling algorithm. The important line that contains the codified version of the die face selection process is:

```
index = np.where(pcdf >= xi[iTrial, jRound])[0][0]
```

The remainder of the function adds up these trials and stores them in a row of values. The output for a single round is an array that contains the count of the number of times each value (die face) is selected. This array should sum to the number of trials. As was the

case with `np.random.multinomial`, there will be an array of length `nRounds`, with each round containing an array of `numFaces`. We can now generate histogram plots and compare them to our NumPy function. We won't be able to compare specific numbers, but we will be able to verify that the statistics are similar to within a sampling error (which will, of course, decrease for large sample sizes). In our notebook, we simply generate another histogram and verify visually that they are similar. In practice, there are many stricter methods for making this comparison more quantitative, which we will omit for the sake of brevity.

Generating a Unit Test with a Smaller Sample

Now that we know that our function has the right statistical properties, we can generate a much smaller pseudorandom example. A pseudorandom sequence will mimic a random sequence, but we can make it reproduceable by setting the random seed to the same value each time the code is run. This sequence will be more easily generated at compile time and a comparison can be made. We have made a detailed study of the algorithm's properties in our notebook. We can now simply generate a pseudorandom sequence of numbers and compare them to a known sequence to validate the function. The larger, more time-intensive study will reside in the notebook, with all of the supporting data and explanations (much in the way that scientific code will refer to journal articles to explain the code).

```
>> np.random.seed(10)
>> nTrialsUT = 2
>> nRoundsUT = 5
>> sLocal = multinomialLocal(nTrialsUT, p1, size=nRoundsUT)
>> print(sLocal)
[[ 0.  0.  0.  1.  1.  0.]
 [ 0.  0.  0.  0.  1.  1.]
 [ 1.  1.  0.  0.  0.  0.]
 [ 0.  0.  0.  1.  0.  1.]
 [ 0.  0.  0.  0.  2.  0.]]
```

Writing the Scalable Code

Now we've got our multinomial sampler working with some of the logic more exposed to us. We can now refer to it when writing our function in PySpark. We will describe Apache Spark in detail in later chapters, but for now you can think of it as a framework for writing code at scale, which simply means that it can be run on a cluster

with many machines. If we can correctly write this in PySpark, it can be scaled to arbitrarily large sample sizes with no change in the way the code is written. We can verify that it works correctly at desktop scale by comparing it to our other validated functions. Additional work may be needed to validate it at scale, but we will have made the most difficult leap already, which is from the desktop to the cluster.

```
# Using Spark random number generator and
# accumulating statistics for a single round.

def countsForSingleRound(numFaces,nTrials,seed, pcdf):
    s = np.zeros(numFaces)
    xi = pyspark.mllib.random.RandomRDDs. \
          uniformRDD(sc,nTrials,seed = seed )
    index = xi.map(lambda x: np.where(pcdf >= x)[0][0]).
              map(lambda x: (x,1))
    indexCounts =  \
       np.array(index.reduceByKey(lambda a,b: a + b).collect())
    # assigning counts to each location,
    # accounting for the possiblity of zero counts in
    # any particular value
    for i in indexCounts:
        s[i[0]]=i[1]
    return s
```

We start by writing a function to do the counts for a single round, which leverages the uniform random number generator in Spark, `pyspark.mllib.random.RandomRDDs.uniformRDD`. From there the essential sampling piece has a slightly different syntax than in NumPy, as shown here:

```
index = xi.map(lambda x: np.where(pcdf >= x)[0][0]).
              map(lambda x: (x,1))
```

This code will generate an RDD (resilient distributed dataset) of key/value tuples. The first element is the index number (die face), and the second is a number that will be accumulated in a downstream counting step (this value is used a lot in word counting algorithms on distributed systems). Now we can wrap another function around this that samples for multiple rounds:

```
def multinomialSpark(nTrials, p, size):
    # setting Spark seed with numpy seed state:
    sparkSeed = int(np.random.get_state()[1][0] )
    nRounds = size
    numFaces = len(p)
    pcdf = np.zeros(numFaces)

    # computing the cumulative probability function
```

```
    pcdf[0] = p[0]
    for i in range(1, numFaces):
            pcdf[i] = p[i-1] + pcdf[i-1]

    s = np.zeros((nRounds, numFaces))
    # assume that nRounds is of reasonable size
    # (nTrials can be very large).
    # This means that Spark data types won't be needed.
    for iRound in range(nRounds):
        # each round is assigned a deterministic, unique seed
        s[iRound, :] =
             countsForSingleRound(
             numFaces, nTrials, sparkSeed + iRound, pcdf)
    return s
```

Our histograms here should also look pretty much the same for sufficiently large sample sizes. Once we have verified this, we can generate a pseudorandom sequence for unit testing:

```
>> np.random.seed(10)
>> sSpark = multinomialSpark(nTrialsUT, p1, size = nRoundsUT)
>> print(sSpark)
>> print(sSpark[0:5])
[[ 1.  0.  1.  0.  0.  0.]
 [ 0.  0.  0.  1.  0.  1.]
 [ 0.  2.  0.  0.  0.  0.]
 [ 0.  0.  0.  1.  1.  0.]
 [ 0.  1.  1.  0.  0.  0.]]
```

These outputs can now be used to define a unit test. Unit tests are used to verify that a function (or method, depending on how it is written) is producing the correct outputs. For large codebases, they are a fundamental component that allows the developer to make sure that newly added functionality does not break other pieces of code.

These tests should be added as new functions are incorporated. In many cases, you can even write the test beforehand and use it as a recommendation for writing the function by enforcing input and output types as well as the expected content. This approach to coding is referred to as *test-driven development* (TDD), and can be a very efficient way to assign coding tasks to a team.

At the very least, TDD can be a nice way to concretely express your idea to those who are considering it for production code. The unit tests for the two functions discussed are given as follows, with the outputs extracted. Notice that the random seed assignment is critical to the reproduceablity of these functions.

```
import unittest

class TestMultinomialMethods(unittest.TestCase):
    # See
    # http://localhost:8888/notebooks/multinomialScratch.ipynb
    # for a detailed description

    nTrials = 2
    nRounds = 5

    def testMultinomialLocal(self):
        np.random.seed(10)
        p = [1/6.]*6
        nTrials = 2
        nRounds = 5
        # reference data generated in notebook
        # (preferably a GitHub link)
        # http://localhost:8888/notebooks/multinomialScratch.ipynb
        # Numpy-Unit-Test-Data
        sLocalReference = np.array([[ 0., 1., 0., 0., 1., 0.],
                                    [ 1., 1., 0., 0., 0., 0.],
                                    [ 0., 0., 0., 1., 1., 0.],
                                    [ 0., 1., 0., 0., 1., 0.],
                                    [ 1., 0., 1., 0., 0., 0.]])

        sTest = multinomialLocal(nTrials, p, size=nRounds)
        np.testing.assert_array_equal(sTest[0:5],sLocalReference)

    def testMultinomialSpark(self):
        np.random.seed(10)
        p = [1/6.]*6
        nTrials = 2
        nRounds = 5
        # reference data generated in notebook:
        # http://localhost:8888/notebooks/multinomialScratch.ipynb
        #Spark-Unit-Test-Data
        sSparkReference = np.array([[ 0., 0., 1., 0.,  1.,  0.],
                                    [ 0., 0., 1., 0.,  1.,  0.],
                                    [ 0., 0., 1., 0.,  1.,  0.],
                                    [ 0., 0., 1., 0.,  1.,  0.],
                                    [ 0., 0., 1., 0.,  1.,  0.]])

        sTest = multinomialSpark(nTrials, p, size = nRounds)
        np.testing.assert_array_equal(sTest[0:5],sSparkReference)
```

We can run the unit test as follows to see how it looks:

```
>> suite = unittest.TestLoader().\
... loadTestsFromTestCase(TestMultinomialMethods)

>> unittest.TextTestRunner(verbosity = 2).run(suite)
testMultinomialLocal (__main__.TestMultinomialMethods) ... ok
```

```
testMultinomialSpark (__main__.TestMultinomialMethods) ... ok

Ran 2 tests in 0.846s

OK
```

Summary

The tests were a success, and so we can provide not only the function, but also a means for testing it. We now have a well-defined function that can be implemented and studied at scale. The development team can now review the notebook before it gets implemented in the codebase. From this point forward, we can focus on implementation issues rather than wonder if our algorithm is performing as expected.

In the next chapter, we will discuss the details of setting up your notebook environment. Once this is complete, you will be able to run all of the examples in the text and in the GitHub repository.

CHAPTER 2
Setting Up Your Notebook Environment

Each approach we discuss is beneficial to go through, as you will be exposed to various technologies and services that are useful for you to understand. The examples in this book are designed to be run in a laptop-scale environment, which means that they can run on a single node. No GPU or multinode (cluster) resources will be needed. Figure 2-1 shows the technologies that you will be working with in this book.

Once you become familiar with these technologies at this scale, you can consider investing resources in the advanced hardware that is needed to run enterprise-scale workloads. The great benefit of working with the code at this smaller scale is that the syntax is essentially unchanged on the more sophisticated hardware configurations.

The entire repository of notebooks used for this book can be found at *https://github.com/nilmeier/DSatEnterpriseScale*.

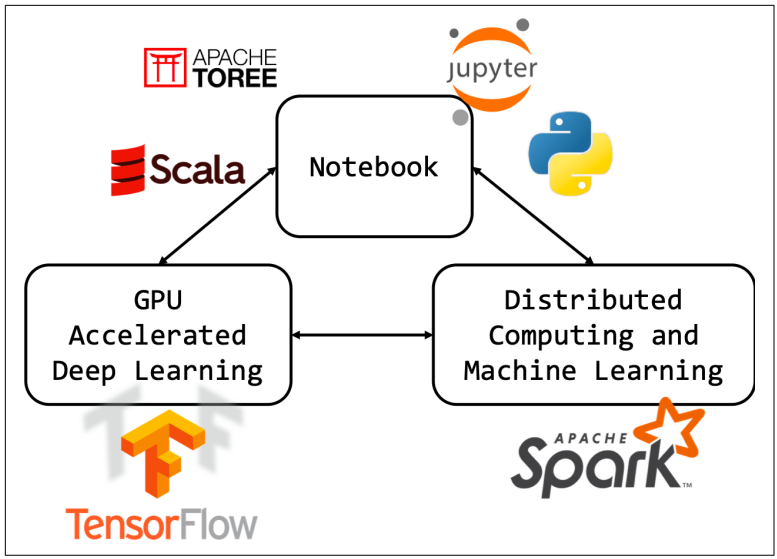

Figure 2-1. The enterprise-scale data science ecosystem covered in this book

Quick Start with Watson Studio

Watson Studio is IBM's hosted notebook service, and you can create a free account at *https://www.ibm.com/cloud/watson-studio*. Other hosted notebook services can be used to run the noteooks as well, but Watson Studio offers all of the frameworks and languages that are used for this book's examples. Once you have created an account and logged in, you can begin by creating a project and notebook.

Creating a Project and Importing a Notebook

Once you have logged in, you will see IBM Watson at the upper-left corner. To create a new project, click the Projects tab and select the "New project" button. The page for creating a new project should look like Figure 2-2.

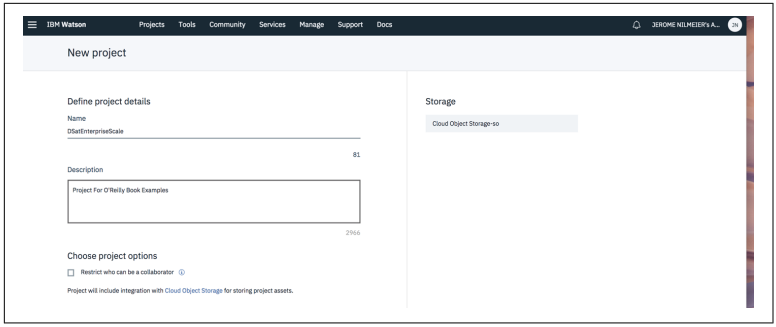

Figure 2-2. Page for creating a new Watson Studio project

You can name the project whatever you like. Here we've named it DSAtEnterpriseScale because the GitHub repository has the same name. A project is a place for storing (among other services) notebooks and data. Once the project is created, go to the Assets tab and see what is available to you, as shown in Figure 2-3. For the purposes of this text, only data assets and notebooks will be used.

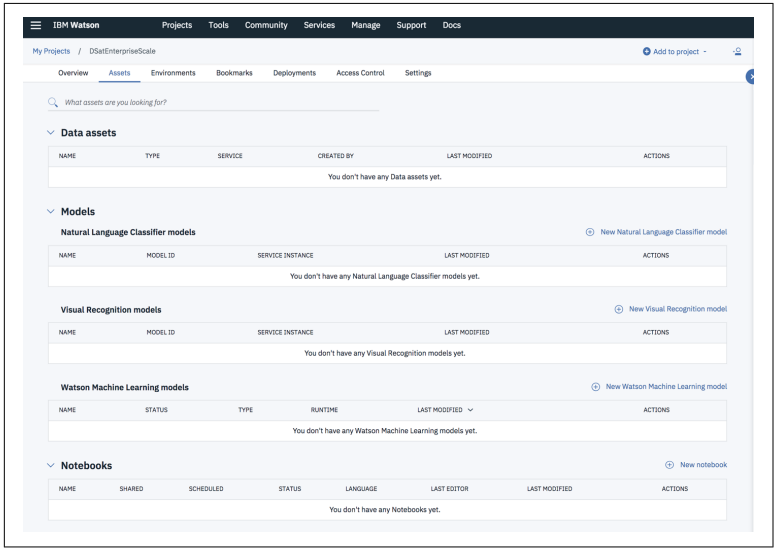

Figure 2-3. Project page for creating a new Watson Studio notebook

Creating Spark environments

By default, the Apache Spark environments that we want to use are not available, so we will need to create them. Some examples will be

in Scala and some will be in Python, so those are the two environments we will create.

To create an environment definition:

1. From the Environments tab in your project, click "New environment definition."
2. Enter a name and a description.
3. Select the environment type: either Python 3.5 or Scala. You will need to create both environments.
4. Select at least three executors for each environment.

Creating your first notebook

Click the "New notebook" tab from the project page. The example from Chapter 1 is available on GitHub. You can import this notebook directly from the GitHub repo, or download the notebook and import it as a file. You can also create a new notebook from scratch. If you create it from scratch, you will need to specify the environment (Python 3.5 and Spark 2.3). If you import the notebook, you may need to manually change the kernel. If the notebook is imported, a security feature will prevent you from running the notebook until you mark it as a "trusted" notebook.

Once the notebook environment is created, you should be able to run code within the cells. The first two cells are shown in Figure 2-4. The second cell will not run correctly if the Spark environment is not set up properly. It is important to verify that your Spark Context exists in order to run the examples effectively. A correctly initialized Spark Context will return the output shown in Figure 2-4.

You can run each cell (including markdown cells) with Ctrl-Enter or Shift-Enter to see the results of each calculation.

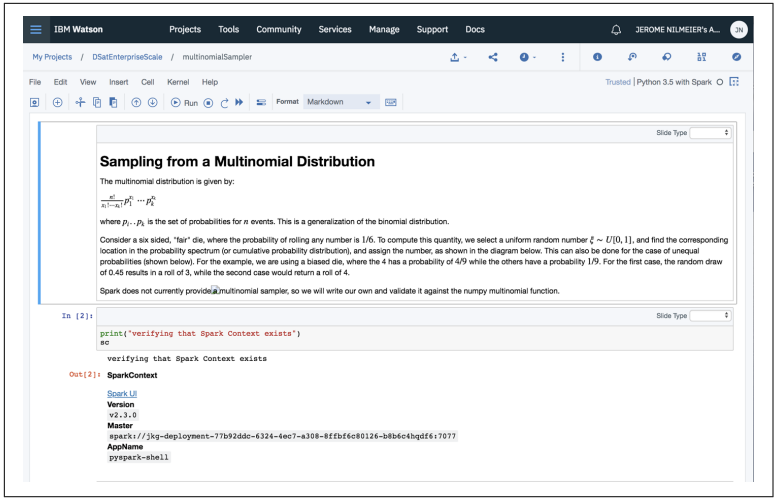

Figure 2-4. The first two cells of the multinomial sampler notebook

Setting Up Your Own Environment

Using Docker Images

Docker is an open source container framework that allows you to quickly build environments that do not affect the overall configuration of your system. It can be a very quick and easy way to get started, and is a very popular option. Installation of Docker is relatively straightforward, and instructions can be found at *https://www.docker.com/get-started*. Once you have installed Docker and downloaded the entire repository from GitHub, simply go to the directory and build the Docker image by typing:

```
$ docker build -t dses .
```

You will only need to build the image one time. It will take several minutes to download all of the required packages. After the image is built, type:

```
docker run -it -p 8888:8888 dses
```

This will launch a Jupyter instance hosted on *localhost:8888*. The terminal will instruct you to browse to *http://localhost:8888/?token=<someVeryLongString>*. Once you have browsed to this address, you can follow the instructions in "Running commands in Jupyter" on page 27.

Installing Apache Spark, TensorFlow, and Notebooks

If you want to have a better understanding of what is required for the exercises in this book, you can manually install all of the components yourself. While it is a time-consuming process, it is also very instructive, and will give you a strong foundation that will be very helpful for you and your work teams when setting up specialized environments.

We provide instructions for installing the frameworks used in this book for macOS and Linux systems. For Windows systems, we recommend using a virtual machine environment that emulates a Linux OS. For Linux systems, Ubuntu seems to be the preferred version for TensorFlow installations. The main components to be installed are:

- Apache Spark
- Jupyter Notebooks
- TensorFlow

Installing Spark

Apache Spark is a framework for scalable computing. You can, however, download and run it on a single compute node (your laptop works too). The code can be written in `local` mode, and can then run on a cluster with larger-scale data and computations without modifications. The project is written entirely in Scala, which compiles to Java byte code. This means you will need the Java Development Kit (JDK).

Installing Java

Java may already be installed on your machine. To check, simply type:

```
$ java -version
```

For this book, we recommend Java 8. *Java 9 will not work for this version of Spark!* Older versions of Java may work, but we have not tested them for the examples presented.

```
java version "1.8.0_161"
Java(TM) SE Runtime Environment (build 1.8.0_161-b12)
Java HotSpot(TM) 64-Bit Server VM (build 25.161-b12, mixed mode)
```

If you have a different version of Java, you will want to install Java 8, which is available at *http://www.oracle.com/technetwork/java/javase/downloads/index.html*. Be sure to install the correct version, and the development kit (JDK), not just the runtime environment (JRE). If you wish to maintain your original Java version, you can install Java 8 in a different directory and point to it by setting the environment variable `$JAVA_HOME` like so:

```
export JAVA_HOME=/path/to/java8
```

Apache Spark will use this environment variable to access the Java Virtual Machine (JVM).

Installing Spark Binary

Once the Java environment is working, installing Spark is very straightforward. We will not attempt to set up the development environment here, and will download only the compiled code. The binaries are available at *https://spark.apache.org/downloads.html*. Be sure to pick version 2.3.0 and select the package type labeled "Pre-Built for Apache Hadoop 2.7 and later." You will be redirected to a list of mirror sites, and we recommend picking the site closest to you.

Then, download the binary *spark-2.3.0-bin-hadoop27.tgz*, and unzip it to a convenient location in your directory structure. You will need to set the `$SPARK_HOME` environment variable for the notebook applications to work correctly:

```
$ export SPARK_HOME="/path/to/spark-2.3.0-bin-hadoop2.7"
```

If you wish to set this variable every time your terminal opens up, you can update your *.bash_profile* file (macOS) or your *.bashrc* file (Ubuntu and other Linux) with this shell command.

To test the installation, launch Spark shell with this command:

```
$ $SPARK_HOME/bin/spark-shell
```

You will see a splash screen that tells you what version you are using, and other settings that may be useful to know later on.

```
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel).
Spark context Web UI available at http://rally1.fyre.ibm.com:4040
Spark context available as 'sc'
(master = local[*], app id = local-1531752157593).
Spark session available as 'spark'.
```

```
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\   version 2.3.0
      /_/

Using Scala version 2.11.8
(OpenJDK 64-Bit Server VM, Java 1.8.0_162)
Type in expressions to have them evaluated.
Type :help for more information.
```

Since the native language of Spark is Scala, you will see a `scala>` prompt. We will go into more detail about Scala syntax later, but for now we only want to verify that an object called the Spark Context has been created. This object, `sc`, is initialized by the `spark-shell` command. The Spark Context is the portal to all things Spark, and contains all of the information required to communicate commands to a cluster of computers. Of course, we are only using a local machine here for our examples, but it is possible to connect the Spark Context to a cluster of arbitrary size and run the same code at much larger scale.

To verify that this object has been initialized properly, type:

```
scala> sc
```

You should see:

```
res0: org.apache.spark.SparkContext =
org.apache.spark.SparkContext@3f322610
```

If you do not see something like this, something has gone wrong in creating the Spark Context, and you will not be able to run Spark commands.

You can also verify that the Python API to Spark is working like so:

```
$SPARK_HOME/bin/pyspark
```

You should see a similar splash screen upon initialization, followed by a Python shell prompt. Verify that the Spark Context is correctly initialized by typing **`sc`** at the Python REPL:

```
>>> sc
```

You should see a similar result:

```
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /__ / .__/\_,_/_/ /_/\_\   version 2.3.0
      /_/

Using Python version 2.7.13 (default, Dec 20 2016 23:09:15)
SparkSession available as 'spark'.
```

The native version of Python in many systems is 2.7.X, but this will vary depending on your settings. Note that PySpark will initialize based on the version of Python associated with the `python` command, and is not tied to the Spark installation. We will see that this is the case once we update our version of Python to 3.5.

We can once again verify that the Spark Context exists by typing **`sc`** at the Python shell prompt:

```
<SparkContext master=local[*] appName=PySparkShell>
```

More details on how to use Apache Spark are in Chapter 3. Most examples are in notebook form, however, so we will need to install a notebook environment first. The next section will discuss how to create a Python environment and how to install all of the necessary libraries to run the code we will be using throughout the rest of the book.

Creating the Python Environment

The first order of business is to create a Python environment that has the necessary libraries and dependencies in a place separate from your native Python installation. This environment will include things like NumPy and Jupyter, which have complex dependencies and can present challenges if we were to install them completely from scratch. The Anaconda project will come to our rescue and provide the necessary libraries, as well as convenient ways of managing which version of Python we choose to use.

For Ubuntu systems, the installer is `apt`, and requires `sudo` access to use. It is a good practice to use `sudo apt update` and `sudo apt upgrade`. To obtain the `curl` command, which is used for installation, you must install with the installer. The commands are as follows:

Ubuntu Anaconda installation

```
$ sudo apt update
$ sudo apt upgrade
$ sudo apt install curl
```

Download the Anaconda install script. For Linux, the command is:

```
$ curl -O
    https://repo.continuum.io/archive/
    Anaconda3-5.0.1-Linux-x86_64.sh
```

macOS Anaconda installation

Most macOS systems will have `curl` installed. For macOS, the command is:

```
$ curl https://repo.continuum.io/archive/
       Anaconda2-5.1.0-MacOSX-x86_64.sh -o anaconda2.sh
```

At the end of the installation, you will be asked if you wish to prepend the Anaconda installation to your `$PATH` variable. We recommend that you select yes here.

Following the Anaconda documentation, we will create a separate environment with the recommended Python version of 3.5. From within this environment, we will add all of the libraries that will be used for this book. This environment can have libraries installed separately from your native Python environment, which can prevent dependency and versioning issues in the future. We recommend Python 3.5 because the most recent version of Python (3.6 as of the writing of this text) does not seem to have full GPU support. While this is not likely to be an issue for our needs, it is something to consider when you are creating environments in the future. You will use the `conda` command utility to create this environment:

```
$ conda create -n py35 python=3.5 anaconda
```

Once your environment is created, you can activate it and launch the shell as follows:

```
$ source activate py35
(py35) $
```

NOTE

If you want to return to the native Python environment, type **`source deactivate`**, and the prompt will return to normal, indicating that your native Python installation is available.

Notice that the prompt now shows the py35 environent. Launch the Python shell.

```
(py35) $ python
```

A splash text will appear telling you which version is installed:

```
Python 3.5.5 |Anaconda, Inc.|
(default, Mar 9 2018, 12:23:37)
[GCC 4.2.1 Compatible Clang 4.0.1
(tags/RELEASE_401/final)] on darwin
Type "help", "copyright", "credits" or
"license" for more information.
>>>
```

For the remainder of the book, we will work in this Python environment.

Installing Jupyter

Following the Anaconda documentation, add this to your *.bashrc* or *.bash_profile* file:

```
export SPARK_HOME=/path/to/spark-2.3.0-bin-hadoop2.7
```

To launch Jupyter Notebook, simply type:

```
(py35) $ jupyter notebook
```

This command will launch Jupyter in your default browser. You should see a browser tab open that looks like Figure 2-5.

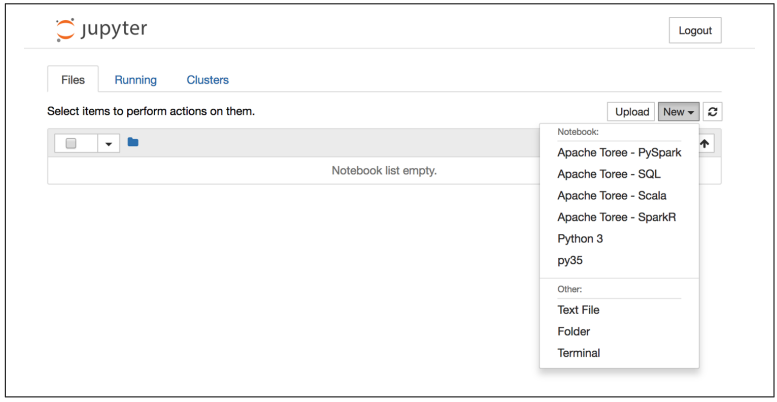

Figure 2-5. The Jupyter landing page

Under the New tab, select the `py35` kernel from the drop-down list (see Figure 2-6). This will open a new notebook with the correct

kernel. We encourage you to explore the notebook interface by running example code.

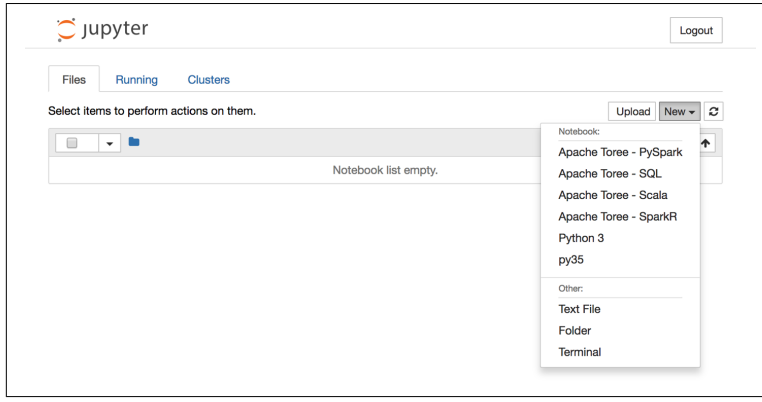

Figure 2-6. The drop-down menu will allow you to choose which kernel to run when launching a notebook

For purposes of verifying the installation, you can run the code in the cell as shown in Figure 2-7.

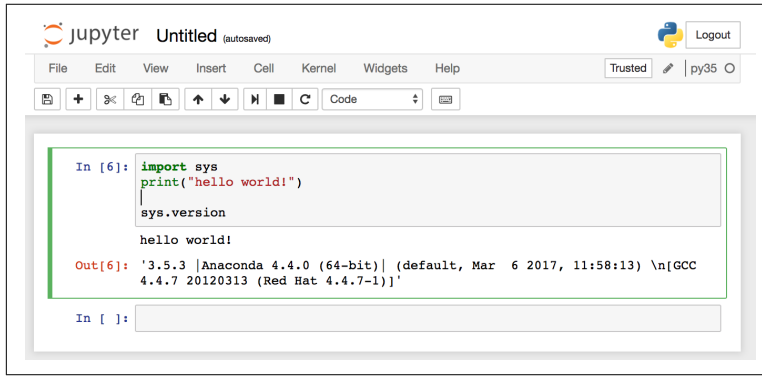

Figure 2-7. Hello World code, along with a version check; the name of the kernel (py35) is shown in the upper-right corner

To launch a Jupyter session on a remote terminal, you need to tell Jupyter what IP address to use when processing remote requests. Since the session is running remotely, you will not want the remote server to try to launch a browser, as you will be connecting to it with your own local browser.

```
(py35) $ jupyter notebook --ip=999.99.99.9 --no-browser
```

For remote sessions, you use the IP address of the remote server. To obtain this IP address, type the shell command **`ifconfig`** and look for the address corresponding to `inet`.

You will see the following instruction for launching Jupyter remotely:

```
Copy/paste this URL into your browser when you connect for the
    first time, to log in with a token:
        http://999.99.99.9:8888/?token=someVeryLongString
```

When you point your browser to this address, you should see the same landing page you would see if you had run it from your local machine (refer back to Figure 2-5).

Running commands in Jupyter

The Jupyter Project documentation is extensive, with great examples to get you accustomed to using the notebook interface.

The basic idea is that there are input cells for you to enter code, and output cells that are returned by the kernel.

Running Spark in Jupyter: Apache Toree

IBM has released an open source project called Apache Toree, which allows for Apache Spark to be run in Scala from within the Jupyter environment, as well as PySpark. To install Toree, use the `pip` installer:

```
(py35) $ pip install toree
```

Once Toree is installed, you will need to set the environment variable `$SPARK_OPTS` (you should have set `$SPARK_HOME` previously). In your *.bashrc* or *.bash_profile* file, include:

```
Export SPARK_OPTS='--master=local[*]' jupyter notebook
```

The `[*]` option specifies the number of threads, and `*` indicates that you wish to have as many threads as are available (you can specify a number as well if you know how many you want to use).

We will only use the Scala and PySpark terminals for this book, but it may be of interest to you to pursue the other interpreters. Once Toree is installed, you can launch Jupyter Notebook as usual, but you will now have several kernels to choose from:

```
jupyter toree install --interpreters=Scala,PySpark,SparkR,SQL
```

The Scala kernel is very useful because you can run Scala interactively. Since it is Apache Spark's native language, there can be advantages to writing code in Scala, and an interactive interface can often be faster for people still learning the syntax.

You will have two options for running the Python API to PySpark. The first is to launch the Toree PySpark kernel from the drop-down list, and the second option is to launch the py35 kernel and initialize the Spark kernel manually. We will choose the second option for most of the examples in the book.

Installing Deep Learning Frameworks

If you are on a system without GPU support, you can install the basic version of TensorFlow with the Python index package manager. The examples in this book are intended to be run using only standard CPU resources, so that you can defer the effort required to set up or acquire accelerated hardware.

```
(py35)$ pip install tensorflow
```

If you have installed CUDA and verified that it is working, you can install the GPU-enabled version of TensorFlow:

```
(py35) $ pip install tensorflow-gpu
```

In either case, you can verify that TensorFlow has been installed by importing it at the interactive Python shell like so:

```
>>> import tensorflow as tf
```

You may get some output depending on the state of your GPU connectivity. If there are no errors, you should be able to access the newly created module as follows:

```
>>> tf
<module 'tensorflow' from
'/path/to/my/anaconda2/envs/py35/lib/
python3.5/site-packages/tensorflow/
__init__ __.py'>
```

Summary

That's it for setting up your environment! You should be able to run all of the examples in the remainder of the text through any of the three options discussed in this chapter. We encourage you to run the notebooks for most of the examples, but you can also copy the code

into text files and run it from either the Python or Scala REPL. The remaining chapters will mostly cover examples.

CHAPTER 3

Data Science Technologies

Data science tooling has entered a golden age. At the laptop scale, the most common tools are R, MATLAB, and Python Scikit-learn, but there are many others. Oftentimes, an expert data scientist will have her "go-to" language, where she feels most confident developing prototypes. A data engineer may also have her preferences when writing scalable code.

There are so many tools to choose from that it can sometimes be a challenge to know which ones to start with. Our aim here is to narrow the field by providing technical foundations for a few simple frameworks. These may not be optimal for all workloads, but they are certainly among the most popular choices for many use cases. We will discuss Apache Spark for most scalable applications. For the deep learning examples, we will use TensorFlow, which we will not discuss in detail in this chapter. The examples provided in later sections are relatively straightforward to follow along with.

Apache Spark

As you have learned, Apache Spark is a framework for writing distributed code. It can be developed at the desktop scale on moderately sized datasets. Once the code is ready, it can be migrated to a cluster or cloud computing resource. The scale-up process is straightforward, and often requires only trivial modifications to the code to run at scale.

Spark is often considered to be the second generation of distributed computing in the enterprise. The first generation was Hadoop, which consists of the Hadoop Distributed File System (HDFS), a resource manager (YARN), and an execution framework (MapReduce). Apache Spark can use a variety of resource managers and filesystems, but the basic design has its roots in the Hadoop ecosystem.

Spark Core: Executors, Cluster Configurations, and More

When a Spark instance is running, it connects to the cluster through an object known as the Spark Context, or `sc`. For the examples you will be running, the notebooks are configured to run in `local` mode, which simply means that the Spark Context communicates only with the local machine that is running the notebook.

In general, however, the Spark Context contains all of the information needed to coordinate jobs across the entire cluster. The Spark Context resides in the same location as the driver program. The driver program is responsible for creating and monitoring the progress of jobs that result from the code being evaluated. These jobs are sent from the driver program to a cluster manager (most often YARN or Kubernetes), which is external to the Spark instance and manages all jobs being sent to the cluster. Based on the nature of the code, the driver program will know how to parse it into parallel jobs when possible.

The cluster manager will then distribute these jobs to executors, which persist on the worker nodes for the duration of the Spark instance (unless the executors are dynamically allocated). The executors will complete the jobs, which can be further divided into dependent tasks. If executors need to exchange data with each other, they can do so directly (most often as a shuffle operation). Once the jobs are completed, the requested results can be sent directly to the driver. Notice in Figure 3-1 that each node has its own data source. When possible, calculations on the data are performed at the same location as the data, a process referred to as "leveraging data locality."

Table 3-1 summarizes these elements of the Apache Spark cluster.

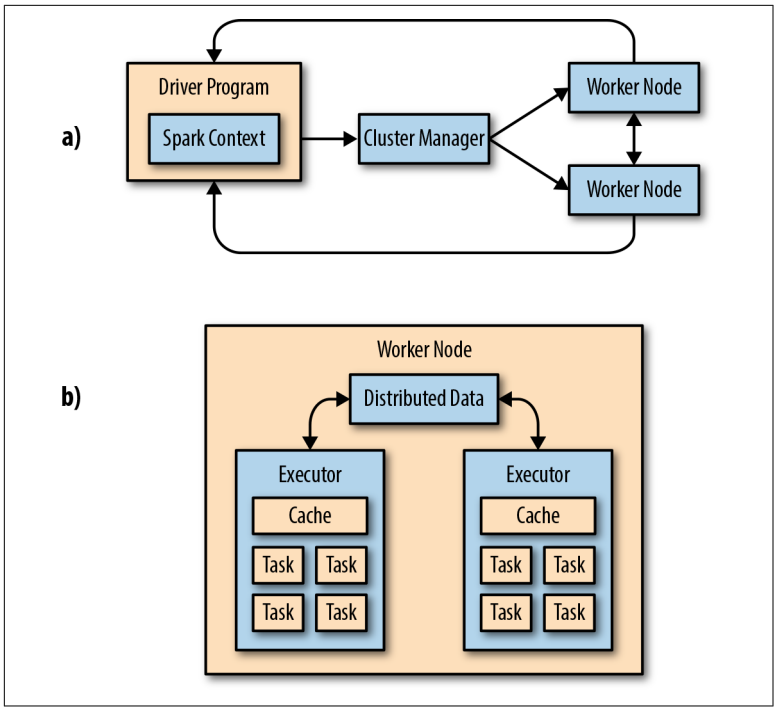

Figure 3-1. The Spark Context sits on a driver node and sends jobs (consisting of several tasks) to the cluster manager. The cluster manager (YARN, Standalone, Mesos, or Kubernetes) will then send the jobs to executors on the network. Each compute node can have multiple executors. A typical executor will use at least four CPUs for effective multithreading. When the job is complete, the results can be sent directly to the driver.

Table 3-1. Important elements of an Apache Spark cluster

Application	User program build on Spark.
Driver program	The process running the `main()` function of the application and creating the Spark Context.
Execution framework	The system that runs the distributed computation (Apache Spark or MapReduce, for example).
Cluster manager	An external service for acquiring resources on the cluster. Examples are Mesos, Standalone, Kubernetes, and YARN.
Distributed filesystem	A filesystem that stores large datasets across multiple disks. Examples are Hadoop Distributed File System (HDFS) and Object Storage.
Worker node	Any node that can run application code in the cluster.

Executor	A process that is launched for an application on a worker node, and runs tasks and keeps data in memory or disk storage across them. Each application has its own executors.
Job	A parallel computation that consists of multiple tasks and gets spawned in response to a Spark action.
Stage	A smaller set of tasks that make up a job and depend on each other.

Spark Core contains the basic parallel framework for running jobs on a cluster. All of the libraries can be accessed through this framework, which provides an impressive breadth of parallel functionality in a single unified platform (see Figure 3-2).

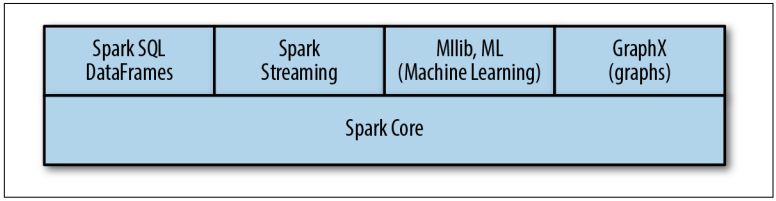

Figure 3-2. The main libraries that are accessible from within Spark

RDDs, Datasets, and DataFrames: How to Use Them

The basic parallel collection from which all scalable operations are carried out in Apache Spark is the *resilient distributed dataset* (RDD). The idea behind the RDD is that resilience not only is enforced at the disk storage level (through the filesystem), but is also guaranteed by the computation framework. To accomplish this, the RDD creates a record of instructions to be carried out on a dataset. This record of instructions in Spark is called a *directed acyclic graph* (DAG). If, at any time during the calculation, the computation node fails, the RDD may recover the state by going to another node where the source data resides and reconstructing the series of calculations (using the DAG) that led to the previous state.

RDDs can be created by:

- Parallelizing an existing collection,
- Loading from a parallel filesystem, or
- Transforming an existing RDD

Parallelizing an existing collection is often done only for demonstration purposes; an RDD is usually created from loading a large filesystem. Any transformation to that initial RDD will result in an

entirely new RDD, as RDDs are immutable objects. The RDD is only a list of operations to be carried out on data that resides throughout the cluster. These operations, or *transformations*, do not actually result in a calculation until an explicit instruction, called an *action*, is given, which will cause the result to be sent from the executors on the cluster to the driver program.

Since all operations concerning RDDs are controlled by the Spark Context, which resides on the driver, the Spark Context is also used to create an RDD.

Example: Creating and Calulating with an RDD

The next example assumes that you have installed the notebook environment as described in Chapter 2. You can create an RDD with an existing Spark Context, `sc`, as follows. Following the notebook given on GitHub, we can see how an RDD is created with the Scala API. Our first few examples will be in Scala, since that is Spark's native language. For a quick syntax reference, see the book's GitHub repo.

For those wishing to see equivalent Python examples, see the corresponding GitHub repo.

Most of the example code here is very straightforward to read and write, however, and requires only a basic understanding of the particulars of the language:

```
val data = 1 to 30

// most RDD operations have identical or nearly identical syntax:

val xrangeRDD = sc.parallelize(data, 4)
val subRDD = xrangeRDD.map(x => x-1)
val filteredRDD = subRDD.filter(x => x<10)
```

```
data = Range(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, ...
            11, 12, 13, 14, 15, 16, 17, 18, ...
            19, 20, 21, 22, 23, 24, 25, 26, ...
            27, 28, 29, 30)
xrangeRDD = ParallelCollectionRDD[18]
at parallelize at <console>:35
subRDD = MapPartitionsRDD[19] at map
at <console>:36
filteredRDD = MapPartitionsRDD[20]
at filter at <console>:37
MapPartitionsRDD[20] at filter
at <console>:37
```

Notice that the `sc.parallelize()` method from the Spark Context was invoked to create the RDD. In the result section, the name of the object is given as `ParallelCollectionRDD`. The next two commands are transformations, which result in new `ParallelCollectionRDDs`. Notice that no actual result is given, nor is any calculation carried out. Transformations are treated as *lazy evaluations*, which means that they are not computed until needed.

In another cell, we can carry out some actions on these RDDs:

```
println("Count: " + filteredRDD.count())
filteredRDD.collect()
```

```
Count: 10
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

These actions actually require a result to be computed and returned, which initiates a calculation. The results of an action are no longer RDDs, but are local Scala objects (or Python objects, or whatever API is used). If an action is carried out on an RDD that has undergone many transformations, the entire chain of calculations will be computed when the first action is called. It is important to remember this process when evaluating performance, as it can sometimes be confusing to determine which operations are consuming the most time.

Caching Results

Since an RDD is an object that persists for the lifetime of the Spark job, it is often desirable to access its contents repeatedly. The ability to cache results in memory was an important initial design element of the Spark framework, and paved the way for massive improvements in efficiency for machine learning and other algorithms that required iterative solvers to find parameters for complex models. An example of the `cache()` method is shown here in Scala:

```
val test = sc.parallelize(1 to 50000,50)
//cache this data
test.cache

val t1 = System.nanoTime()
// first count will trigger evaluation of count *and* cache
test.count
val dt1 = (System.nanoTime() - t1).toDouble/1.0e9

val t2 = System.nanoTime()
// second count operates on cached data only
```

```
test.count
val dt2 = (System.nanoTime() - t2).toDouble/1.0e9
```

And the output is:

```
test = ParallelCollectionRDD[3] at parallelize at <console>:27
t1 = 454365386432198
dt1 = 0.305775544
t2 = 454365692225735
dt2 = 0.122984662
```

The first calculation took 0.3 seconds and the second calculation took 0.122 seconds, but this is not the entire story. A Spark UI is available at *localhost:4040*. From the page shown in Figure 3-3, we can see the time required to compute the `count()` operation. Calculating the first operation on the executor takes 0.2s, which means that there is a 0.1s latency in sending the results to the driver. Now, the second operation takes only 79ms. While the latency of communication to the driver is significant here, it will be less and less noticeable relative to the executor compute time for larger datasets. This example illustrates the massive improvements that can be obtained by careful caching.

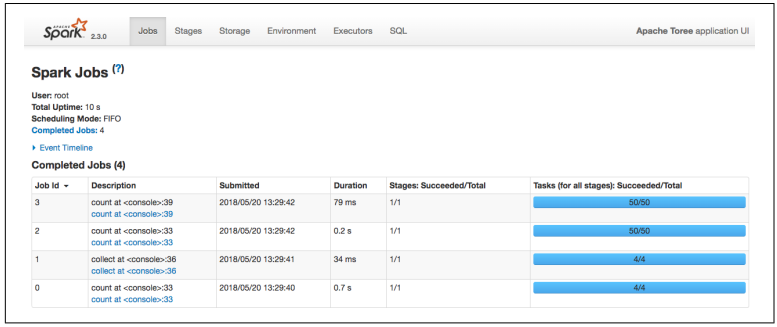

Figure 3-3. The Spark UI has many features for diagnostics. Here we are comparing the calculation time for the two count operations. Job Id 1 is the count after caching, and Job Id 2 is the count before caching.

There are many important features of the Spark UI to understand, but we will not attempt to cover them all here. With regard to caching and memory, however, it is useful to point out the Storage tab, shown in Figure 3-4. The most important thing to notice there is the Fraction Cached column, which tells the user if the entire RDD is in memory. If the entire RDD does not fit in memory for the cluster, the full advantage of caching may not be realized, which can significantly affect performance.

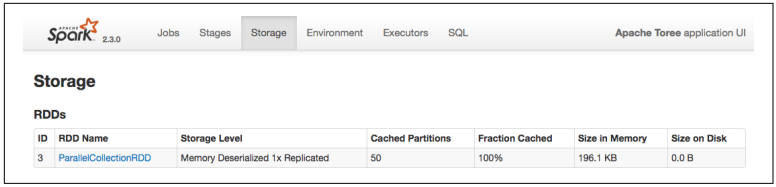

Figure 3-4. The Spark UI also provides information on the status of cached data.

Spark SQL and DataFrames

While Spark SQL is technically a library (refer back to Figure 3-2), it is of fundamental importance, and it should be leveraged as much as possible, particularly when you are loading and processing data for other downstream tasks. The idea behind Spark SQL is to make SQL commands available to the user. SQL, or Structured Query Language, is a powerful language for processing data, particularly when you are combining different data sources to develop new insights. Not only is the language powerful, but the underlying query engine (called Catalyst) is highly optimized, and any workloads that leverage this engine will benefit greatly.

The class that is used to support SQL operations is a DataFrame. DataFrames are simply RDDs with additional annotation, such as a table schema. This allows the RDD to be treated as a database table, which supports efficient SQL operations. DataFrames can be created from a variety of data sources, including ORC, Avro, Parquet, and others. For simple applications, a JSON file is used, as in the next example, which shows the process of creating and querying a DataFrame. Additional examples can be found in the GettingStartedWithSpark notebooks in both Python and Scala.

```
import org.apache.spark.sql.SparkSession

val spark = SparkSession
  .builder()
  .appName("Spark SQL basic example")
  .config("spark.some.config.option", "some-value")
  .getOrCreate()

val df = spark.read.json("people.json")
df.show
df.printSchema
```

```
// Register the DataFrame as a SQL temporary view
df.createTempView("people")
```

In order to create a DataFrame, you must first create a `SparkSession` (named `spark` here), and then use the `json` reader to ingest it into a DataFrame. Simple commands like `show` and `printSchema` can be used to explore the contents of the table. In order to issue SQL commands, you must first create a table view (here it is named `people`).

The output is shown here:

```
+----+-------+
| age|   name|
+----+-------+
|null|Michael|
|  30|   Andy|
|  19| Justin|
+----+-------+

root
 |-- age: long (nullable = true)
 |-- name: string (nullable = true)

df = [age: bigint, name: string]
[age: bigint, name: string]
```

In addition to SQL, other SQL-like syntaxes are available:

```
//need to register DataFrame to use (Hive-like) SQL
println("Query 1: select statements")
df.select("name").show
spark.sql("SELECT name FROM people").show
```

Only the last line shown requires the table registration, while the `select()` method can be called directly. Both give identical results.

```
Query 1: select statements
+-------+
|   name|
+-------+
|Michael|
|   Andy|
| Justin|
+-------+

+-------+
|   name|
+-------+
|Michael|
|   Andy|
```

```
| Justin|
+-------+
```

Summary

In this chapter we discussed the fundamentals of the primary framework for big data that is used in this book, Apache Spark. Though it is only the very beginning of our understanding of this powerful framework, this should give you an idea of Spark's basic syntax and usage.

Now that we have the tools installed and available to us, along with a basic understanding of syntax and usage, we are able to dig deeper into its functionality. The notebook examples that follow in the rest of the book will build on this basic understanding.

CHAPTER 4
Introduction to Machine Learning

Machine learning is a vast subject, and can be daunting at first. Many of its core concepts, however, can be understood with simple examples. These ideas form the basis of our thinking for even the most complex models. We will introduce the basic ideas here (see Table 4-1 for a summary), and will build on them in Chapter 5.

Table 4-1. Basic machine learning terms

Term	Description
Model	The functional form of the prediction. In this chapter, we study the linear model only.
Features	Input variables to the model (x).
Label	Output variable (y). In this chapter, labels are floating-point numbers.
Parameters	The set of numbers ($\mathbf{w}$) that relates the features to the label. Parameters must be discovered from many examples of features and their associated labels.
Linear model	A simple but very useful model where each feature is multiplied by a single parameter. The form of the model is $f(\mathbf{x}, \mathbf{w}) = w_0 + \sum_{i=1}^{N} w_i x_i$, and is used exclusively in this chapter.
Bias	The parameter in a linear model that does not have a feature associated with it (w_0).
Training data	A list of features and associated labels. The data is used to determine the parameters of the model.
Loss function	Also referred to as the objective function. The form of the function to be minimized when determining parameters. The loss function used in this chapter is the least squares function.
Normal equation	A direct solution of the linear model with the least squares objective function. Only a linear model can be solved using this equation.

Term	Description
Numerical optimization	A method for solving for parameters for a given loss function and training dataset. The loss function can be for linear or nonlinear models, but only the linear model is studied in this chapter. The only optimization technique used in this chapter is the gradient descent algorithm.
Gradient descent	The most basic numerical optimization technique. It is an iterative process, where the parameters are incrementally updated by evaluating the gradient of the objective at each step.

Linear Regression as a Machine Learning Model

We will start with the most basic machine learning model, which is linear regression:

$$f(x, w_1, w_0) = w_1 \cdot x + w_0$$

In mathematical terms, $f(x)$, a function of the independent variable x, and w_1 and w_0 (the slope and intercept) are the *parameters* of the model. These parameters are not specified here, but must be determined from a set of example data. The process of using example data to determine parameters was historically referred to as a *regression*. In machine learning parlance, we refer to the determination of parameters as *learning*. Let's see how this is accomplished.

In machine learning, a model like this is used as a *prediction* of a *label* based on an input *feature*. The prediction is $f(x)$, and the feature is x. A linear model with N features would have the form:

$$f(x_1, x_2, \ldots, x_N, w_0, \ldots, w_N) = w_0 + w_0 + \sum_{i=1}^{N} w_i x_i$$

where the variable with no feature associated with it is called the *bias* term. The prediction is a result of applying the model to an incoming feature set, and is based upon a *training set* of data. The training set contains a list of features and *labels*. Labels can, in general, take on discrete values (which leads us more naturally to think of them as labels). For the model studied here, labels are simply floating-point numbers.

The previous equation is often written in the more compact linear algebra notation as:

$$f(\mathbf{x}, \mathbf{w}) = \mathbf{x}\mathbf{w}$$

where:

$$\mathbf{x} = \{1 \;\; x_1 \;\; \ldots \;\; x_N\}$$

is written as a row vector and $\mathbf{w} = \begin{bmatrix} w_0 & w_1 & \ldots & w_N \end{bmatrix}^T$ is written as a column vector. The value 1 in the first row of the feature vector is there to account for the intercept.

Defining the Loss Function

The form of the linear model is defined by the preceding equation, but, as mentioned, the parameters are not specified. We need to determine these parameters by providing example data. We can create a list of M examples as a training set for determining the parameters. The training set will consist of labels, or values $\mathbf{y} = \begin{bmatrix} y_1 & \ldots & y_M \end{bmatrix}^T$ and their associated features $\mathbf{X} = \begin{bmatrix} \mathbf{x}_1 & \ldots & \mathbf{x}_M \end{bmatrix}$. Note that each label entry y_m is a single value, while each row of the matrix $\mathbf{X}$ is a row vector $\mathbf{x}_m$, each consisting of N entries. Figure 4-1 shows a training set that we will be using from the notebook.

In the example, the *mpg* column will be the label column, and the *weight* column will be the feature. We are using only a single feature for this example (the multivariate case is treated in the notebook).

The *loss function*, or *objective function*, is a function that is defined so that the parameters can be determined. In general, the parameters that minimize the loss function will be those that are used. These parameters are completely determined by the data provided and the loss function that is used. The resulting model will have a specific set of parameters $\widehat{\mathbf{w}}$. The loss function that is most commonly used is the *mean squared error* (MSE), which is defined as:

$$S(\mathbf{w}) = (\mathbf{y} - \mathbf{X}\mathbf{w})^T(\mathbf{y} - \mathbf{X}\mathbf{w})$$

and the optimal parameters $\widehat{\mathbf{w}}$ are those that minimize the loss function with respect to $\mathbf{w}$. This idea is expressed mathematically as:

$$\widehat{\mathbf{w}} = \min_{\mathbf{w}} S$$

```
1 df[['mpg','weight']].head(10)
```

	mpg	weight
0	18.0	3504.0
1	15.0	3693.0
2	18.0	3436.0
3	16.0	3433.0
4	17.0	3449.0
5	15.0	4341.0
6	14.0	4354.0
7	14.0	4312.0
8	14.0	4425.0
9	15.0	3850.0

Figure 4-1. The first few entries of the dataset used for the single feature regression example

The dataset and the objective function provide all of the information needed to define a given model. This basic formulation of the loss function is true for nearly every machine learning model that has labeled data (supervised learning), and this simple linear model provides the conceptual framework that illuminates our understanding of the entire field.

The next step is to solve for the parameters.

Solving for Parameters

A "trick" for linear models: The normal equation

For the simple linear model, a clever solution based on the *normal equation* can be invoked, which looks as follows (without derivation):

$$\hat{\mathbf{w}} = \left(\mathbf{X}^T\mathbf{X}\right)^{-1}\mathbf{X}^T\mathbf{y}$$

This approach works for any number of features, but is useful only for the linear regression case. Figure 4-2 shows the resulting line generated from this approach, which is described in more depth on GitHub. Any type of model other than a linear regression requires a process called *numerical optimization*, which is covered in the next section.

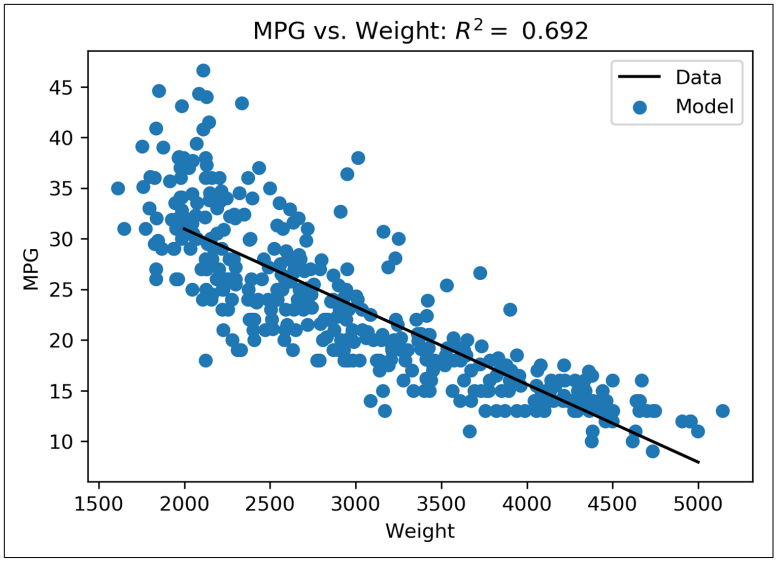

Figure 4-2. The best-fitting line through the data as generated with the normal equation

Numerical Optimization, the Workhorse of All Machine Learning

We presented the linear model in some detail because:

- It forms the basis of many other machine learning models.
- The normal equation solution is very convenient and numerically robust.

In fact, the normal equation method provides solutions that are of such high precision that we can use them as "ground truth"

solutions. As we explore other methods, it can be very helpful to have a solution with known properties for comparison.

In general, machine learning algorithms result in nonlinear objective functions, which means we need to find another (nonlinear) approach.

The most basic of nonlinear optimization algorithms is the gradient descent algorithm. The idea behind the gradient descent is simple: if the objective function forms a (multidimensional) surface, the gradient of this equation will point in the "downhill" direction. If we follow these downhill steps, we will eventually arrive at a minimum point. There are many caveats to consider here in practice, but the basic concept of an energy landscape can be a very helpful way of thinking about optimization problems. We define the gradient of the objective function as:

$$(\nabla_{\mathbf{w}} S)_i = \frac{\partial S}{\partial w_i}$$

which forms an N-dimensional vector. This vector tells us the slope of the surface. The parameters can be updated to follow the downhill pathway by the following gradient descent step, which takes us from parameter state $\widehat{\mathbf{w}}_k$ to $\widehat{\mathbf{w}}_{k+1}$:

$$\mathbf{w}_{k+1} = \mathbf{w}_k - \nabla_{\mathbf{w}_k} S \cdot \alpha$$

where α is adjusted so that step sizes are optimal. If the steps are too large, we risk leaping across the surface and overstepping the optimal location. If, however, the steps are too small, we may find ourselves taking too many unproductive (and costly) steps. The idea behind each optimization step is that the new parameters $\widehat{\mathbf{w}}_{k+1}$ result in a lower loss function. Looking at Figure 4-3, we see that, initially, the objective function is rapidly reduced. Later steps in the process (up to $K = 100$) gradually approach a convergent solution. This behavior is typical for a well-behaved optimization process. Had we chosen our α incorrectly, we might not have seen this smooth approach to a stable solution. In fact, you can adjust α yourself in the notebook and see how the behavior changes!

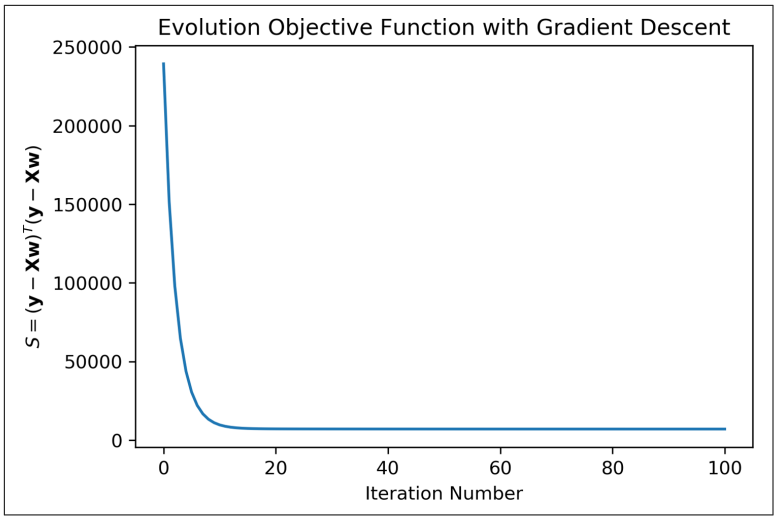

Figure 4-3. Typical convergence behavior of a well-behaved optimization

If we look at the evolution of parameters $\mathbf{w} = \begin{bmatrix} w_0 & w_1 & w_2 \end{bmatrix}^T$, shown in Figure 4-4, we can see that they gradually approach the ground truth solution from the normal equation. In general, we would not have a solution like this for comparison, and we usually only look at higher-level metrics like the objective function as an indicator of the solution's fitness. This is one reason why it is so interesting to look at the linear models, because we can make high-resolution comparisons and develop a deeper intuition about how these systems behave.

NOTE

Don't forget to watch the evolution of parameters for different values of α.

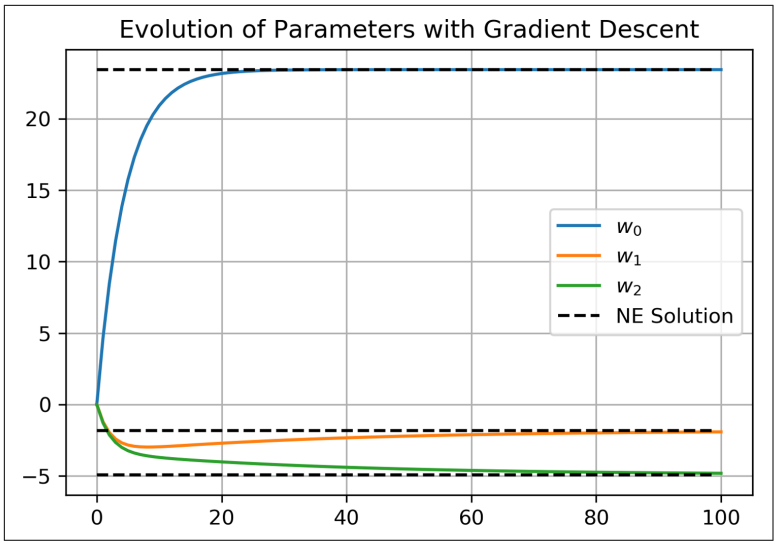

Figure 4-4. Parameters evolve to the ground truth solution as given by the normal equation

Feature Scaling

This well-behaved optimization was not only the result of adjusting the propagation step of gradient descent. It required an additional piece of preprocessing known as *feature scaling*. Feature scaling is not necessary in the normal equation approach, but is absolutely essential in even the simplest of numerical optimization algorithms. Recall that we like to think of our objective function as a surface. Imagine a surface, like a golf course, where the putting green is an ellipse. Ideally, we would like to have a circular putting green, but an ellipse is okay, right? What about an ellipse that is a foot wide and a mile long? You would want to putt in one direction, but use a driver for the other direction, which would make it hard. Feature scaling has the effect of taking your putting green, no matter how distorted, and making it more symmetric, so that propagation steps are of similar scale no matter what direction you are going.

The most popular way to scale features is *standardization scaling*, although there are many others. The standardization scaling is given by:

$$x'_{j,i} = \frac{x_{j,i} - \mu_i}{\sigma_i}$$

where $x_{j,i}$ is the jth example of ith feature. There are N features and M examples, and the primed notation $x'_{j,i}$ indicates the scaled feature. The summary variables μ_i and σ are the mean and standard deviation of the ith feature across the M examples, respectively.

Without this scaling, you would find yourself adjusting propagation step sizes for each dimension, which would add N more parameters to tune. Also, with this scaling, as well as scaling α by the number of samples, we can obtain more general insights into the range of α that works across many different optimization procedures. The typically recommended range for propagation step size under these conditions is somewhere between 0 and 1 as a result.

NOTE

Numerical algorithms can be incredibly sensitive to tuning and environments. You can spend an inordinate amount of time tuning the performance of these algorithms by adjusting their parameters or exploring other related gradient descent algorithms.

Letting the Libraries Do Their Job

The optimization, or training, phase of any machine learning or artificial intelligence algorithm is the most time-intensive portion of the entire calculation. This is because it needs to evaluate the objective function as well as the gradient function, and both of these calculations contain M data points of N features. Here, we only needed $K = 100$ steps and everything went very quickly, but this can very quickly become a cumbersome problem as M, N, and/or K increase.

Furthermore, when the number of data points M becomes very large, the data may need to reside on different machines in your cluster. While this seems like a natural extension, it actually changes the algorithmic approach to favor methods that more efficiently leverage data locality, such as stochastic gradient descent.

While you do want to satisfy yourself that the algorithms are performing as expected, and that you understand the inner workings of the approach, you will increasingly find yourself accessing only the APIs to prewritten algorithms. The frameworks we are discussing in

this book are very well validated, highly optimized, and designed with scale in mind.

In general, you can rely on the results of these frameworks, but you may wish to develop ways of comparing results from various approaches to ensure consistency. As these algorithms become more complex, you will want to make higher-level comparisons to ensure that your algorithms are performing as expected. The last section of the notebook contains a simple call to the PySpark linear regresssion algorithm with feature scaling, along with comparisons to the solutions that we studied in detail in earlier sections.

The Data Scientist Has a Job to Do Too

At the end of the notebook, we have looked at the three approaches (normal equation, gradient descent, and MLlib from PySpark). While we may not have all of the details of the results of the PySpark calculation at our immediate disposal, we can generate high-level metrics, such as the R^2 statistic. This simple comparison can provide enough of a sanity check for both the scientist and engineer to ensure that the calculations are performing as expected.

Recall that you may be comparing results across different frameworks, libraries, and/or languages. Just like a real scientist carrying out a measurement in a laboratory, as a data scientist you must have ways of interpreting the quality of the measurement, even if you may not have a complete understanding of the internals of the calculation. Something as simple as a summary statistic can at least rule out gross errors in inputs and outputs when you are comparing models and methods. If a deeper understanding is required, you can take steps to dive into the finer points of the model being evaluated.

Summary

In this chapter, we have provided the basis for understanding machine learning models. These concepts are surprisingly persistent in even the most complex models, up to and including deep learning models. In Chapter 5, we will build more on these concepts, and see how they are used in classic machine learning examples.

CHAPTER 5

Classic Machine Learning Examples and Applications

In this chapter we will expand upon our basic understanding of what it means to train a model and begin to apply more sophisticated machine learning concepts. This is only a brief introduction, and you are encouraged to read more deeply into each subject as needed for the problem at hand. We have tried to touch on the most important examples of machine learning, both in the text and in notebook form.

Supervised Learning Models

The most widely used and useful models in machine learning fall under the category of *supervised learning*. A supervised learning model is simply a model that has been trained on many examples, where a label is associated with a set of features. The most commonly provided label in these models is a binary classification, which is simply an indication that a set of features falls within one of two categories. Models with multiple categories are also possible, but are really just a generalization of the binary classification case, and are not treated in detail here.

The Activation Function: From a Value to a Label

In Chapter 4, we discussed the use of labels and features to train models. The "label" in these cases was actually just a floating-point number. In practice, most incoming data takes on discrete values.

The simplest case is a binary label, which indicates whether the data point falls into a particular category or not. We will address how multiple categories are treated a bit later, but first let's think about how a continuous function can be interpreted as a label.

In Chapter 4, we defined the linear model as:

$$z = w_0 + \sum_{i=1}^{N} w_i x_i = \mathbf{xw}$$

and the trivial transformation of the linear model is to simply make no modification, which is called the *identity activation function*, given as:

$$f(z) = z$$

The most basic transformation of the linear model is the *logistic function*, which is:

$$f(z) = \frac{1}{1 + e^{-z}}$$

This equation transforms the linear model to a value between 0 and 1, as shown in the second plot of Figure 5-1. In practice, a *threshold* is assigned (usually halfway between the minimum and maximum value of the function). If the value of the function falls below the threshold, then the lower bound is taken as the return value, and the upper bound is returned if the function is above the return value.

NOTE

The activation function provides a means for transforming continuous values into labels. Typically, this means that there are two labels returned ("true" or "false," for example). Multiple category functions are also available, but the basic idea is conveyed through the binary models.

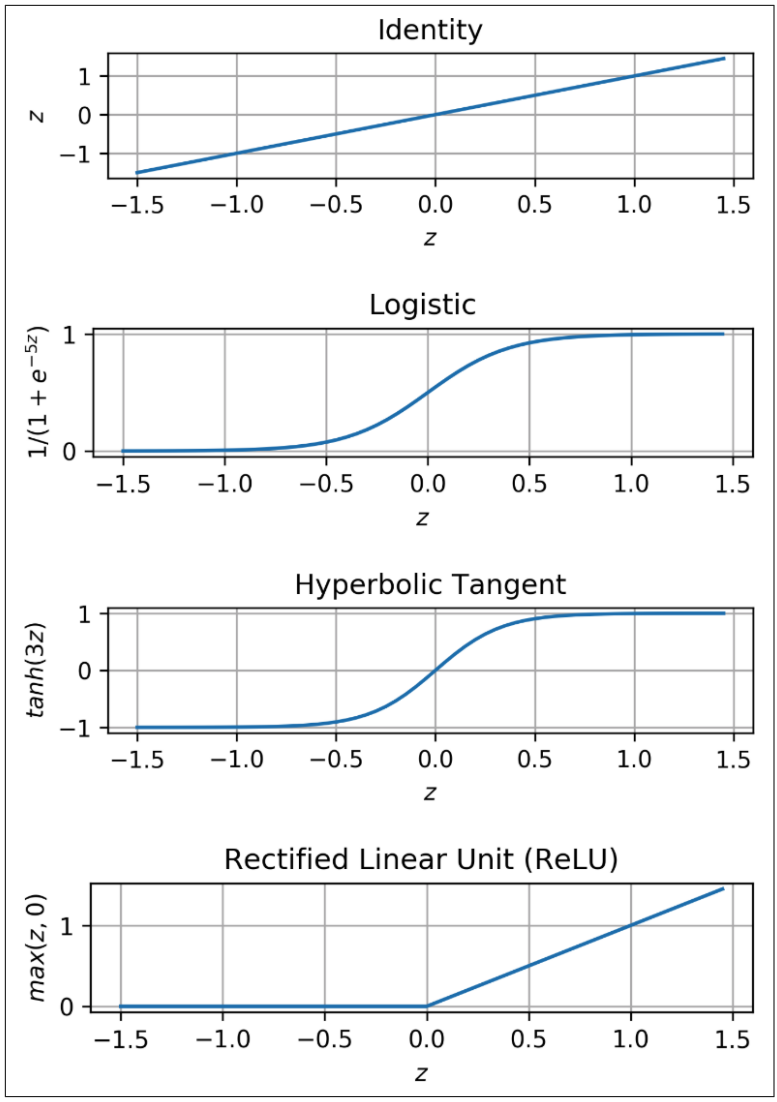

Figure 5-1. Some of the most popular activation functions

The logistic function is by far the most frequently used function in machine learning. It has many important conceptual underpinnings, and is often the starting point for thinking of a continuous function as a way of returning labels. Another function with similar characteristics that can be used in this capacity is the *hyperbolic tangent*, which is given as:

$$f(z) = \tanh(z)$$

and is the third plot of Figure 5-1. In fact, the term *activation function* is not always used in the machine learning field. It is used more frequently in deep learning, and is discussed in more detail in the next chapter.

In deep learning, the activation functions are sometimes taken as labels when used as the outputs, but are also used as intermediate transformations. In those intermediate cases, it is sometimes more convenient to use a function that does not have an upper maximum limit, because the numerical optimization algorithm will perform better in cases where the gradient does not vanish at high values of z. The reasons for this are beyond the scope of this text, but we present one activation function that is very popular in the deep learning field—the *rectified linear unit* (ReLU) function, given as:

$$f(x) = \max(z, 0)$$

Using Labeled Data for Training Your Model

The easiest way to build your first machine learning model is to start with a training set that contains a binary classification. We will study a dataset that will help us predict whether a person is at risk for heart failure from a subset of the data used in the notebook, as shown in Figure 5-2.

```
In [11]: df_data.select("AVGHEARTBEATSPERMIN", "PALPITATIONSPERDAY","CHOLESTEROL","BMI","HEARTFAILURE","SEX").toPandas().head()
Out[11]:
```

	AVGHEARTBEATSPERMIN	PALPITATIONSPERDAY	CHOLESTEROL	BMI	HEARTFAILURE	SEX
0	93	22	163	25	N	F
1	108	22	181	24	N	F
2	86	0	239	20	N	F
3	80	36	164	31	Y	F
4	66	36	185	23	N	F

Figure 5-2. Displaying a subset of the columns used in training the heart failure prediction model

We will use the column "Heart Failure" as the label and treat the remaining columns as features. Note that some columns are reported as categories. Fortunately, Apache Spark (as well as Pandas and many other machine learning libraries) has automated methods for converting fields like this into numerical values so that they can be input into a numerical model. The input labels will be changed into 0 or 1. The resulting prediction will be a floating-point number.

It will be subjected to a threshold comparison, such that any numerical prediction below that value will be between 0 and 1, and the resulting integer will be reassigned to the label originally presented in the dataset. Note also that there is a feature column that will also need to be mapped to a numerical value. For our notebook, this is accomplished with the `StringIndexer` method. Rather than go into too much detail about the method, check out the notebook to see how it is used in practice.

The general procedure for supervised machine learning is illustrated in Figure 5-3. Much of the workflow should look familiar from our regression discussion, since we are providing numerical data for a model, resulting in a trained model that is then applied to incoming data to make predictions. The data preparation step is a very important component of this workflow, and can realistically constitute as much as 80% of a data scientist's time. Data preparation can include tasks like indexing categorical data, processing missing entries, and ensuring data integrity. These processes can be automated, but they still require careful inspection and judgment. The training and prediction steps are often carried out with mature APIs to well-understood algorithms, and often require less attention because they have been so well studied and validated before being made available. Nonetheless, we must understand how these algorithms work if we are to make the best choices for them. It is also important to understand these algorithms so that we can carry out hyperparameter tuning effectively.

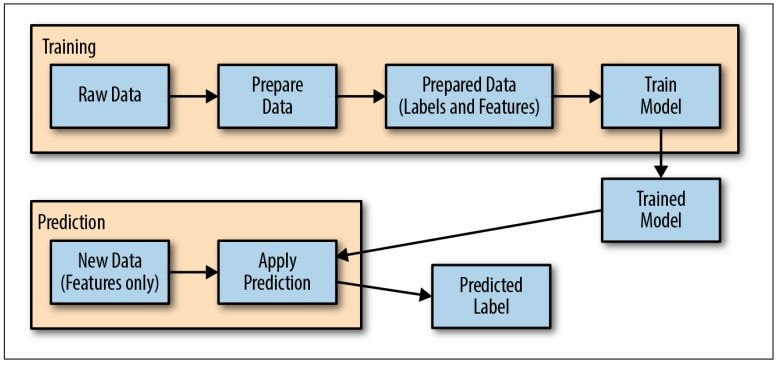

Figure 5-3. Typical workflow for supervised learning models

In most cases, the training step is the most computationally intensive portion of the workflow. Once the model is trained, however, it

can be used repeatedly to make predictions on incoming data. The training of the model is typically something that is done in batch mode and updated periodically with new data, while the predictions are made on data as it comes in through a live online system.

Making Predictions with the Trained Model

The predictions are what your company will be looking for to increase its bottom line, so providing a high-quality model for online use is clearly a deliverable that your team can point to as an achievement.

Evaluating Model Performance and Deploying the Model

It is not enough, however, to provide the model alone. Additional quality metrics should be provided or computed internally to estimate the performance of the chosen model. The usual way to accomplish this is by separating the set of examples into two portions: the training set and the test set. The training set is what is used to generate the parameters of the model. Once the model is generated, the test set is then used to evaluate its performance. Each data point in the test set has a known label, but the prediction is made without this knowledge and compared to the ground truth label. Since we are primarily concerned with binary label predictions, we divide the predictions into positives and negatives. Depending on how you choose to assign labels, the positive would correspond to a prediction of 1, and a negative would correspond to a prediction of 0. If the label is correctly predicted by the model to be a positive, it is called a *true positive*. If it incorrectly predicts a positive result, it is called a *false positive*. Other fitness measures are listed in Table 5-1. The idea behind using a test set is to estimate the performance of the model on incoming data (with unknown labels).

It may not always be the case that your training and test data is actually representative of the incoming features, and you should take care to ensure that your fitness measures on incoming data are similar. How can you do this if your incoming data has unknown labels? Is there some way that you can obtain label assignments on the incoming data to evaluate these metrics? These can be tricky questions to answer, depending on the workflow, but it is something to keep in mind.

Table 5-1. Some commonly used measures of model fitness

Term	Description
Training set	The portion of the dataset used to determine model parameters, or to *train* the model.
Test set	The portion of the dataset held out from training and used to estimate the performance of the model for data with unknown labels.
True positives (*TP*)	The total number of test data samples correctly predicted to be positive.
True negatives (*TN*)	The total number of test data samples correctly predicted to be negative.
False positives (*FP*)	The total number of test data samples incorrectly predicted to be positive.
False negatives (*FN*)	The total number of test data samples incorrectly predicted to be negative.
True positive rate (*TPR*)	$TPR = TP/P$. Also known as *sensitivity* or *specificity*.
False positive rate (*FPR*)	$FPR = FP/P$.
Accuracy	$a = \frac{TP + TN}{P + N}$ Total number of correct predictions normalized by the total number of test data samples.
Precision	$p = \frac{TP}{TP + FP}$ Total number of correctly predicted positives normalized by the total number of positive predictions.

Deployment of the model may often require some engineering. In the example we provide, we evaluate the fitness of the model only on the training set. In all likelihood, once the model has been developed, it may reside in a very large, complex ecosystem of web services that are running continuously and at a large scale. Thus, you may find yourself handing the model off to another team whose main concern is implementing code changes without breaking anything. As a data scientist, you may be asked to provide the model, which may be a formatted file with parameters, or a small piece of code that loads a running instance of the model.

You should take special care to provide much more than this, however. For example, we discussed logistic regression as a candidate for binary category prediction, but there are many more options to choose from. A discussion of these models is beyond the scope of this book, but there are many resources (including the Apache Spark documentation) that provide more in-depth discussions. For our purposes, we will simply note that they are supervized learning models with a binary classification. We can evaluate the performance of these models based on this understanding. For example, we have evaluated some of the most popular models in Spark,

including Logistic Regression, Naive Bayes, and the Random Forest Model. We found that the Random Forest had great performance, which is often the case. It is always worthwhile to try a few out, however, and even be prepared to provide more than one model in production if needed.

Notebooks provide an excellent way to document the process of studying the model. As was the case in Chapter 1, the actual deliverable may only be a few lines of code, or even just a formatted file. We should also strive to document the thought process that went into developing this model. It is important to understand the reasoning behind the choices that resulted in the model in deployment, rather than just an opaque listing of the model type and its parameters. This documentation will be valuable to those who want to better understand what is happening with the code. It also gives insight into your thinking as you developed the model.

Collaborative Filtering

Recommendation systems based on the Alternating Least Squares (ALS) algorithm have gained popularity in recent years because, in general, they perform better as compared to content-based approaches. A recommmendation system suggests items to a new user based on the known preferences of previous users. It has elements of both supervised and unsupervised learning in its formulation. Since it uses labeled data as part of the model training, however, we can think of it somewhat as a supervised learning algorithm.

ALS is a matrix factorization algorithm, where a user-item matrix is factorized into two low-rank non-orthogonal matrices:

$$\mathbf{R} = \mathbf{U}^T\mathbf{M}$$

The elements, r_{ij} of matrix RR can represent, for example, ratings assigned to the jth movie by the ith user.

This matrix factorization assumes that each user can be described by K latent features. Each matrix has K columns. Similarly, each item/movie can also be represented by K latent features. The user rating of a particular movie can thus be approximated by the product of two K-dimensional vectors:

$$r_{ij} = \mathbf{u}_i^T \mathbf{m}_j$$

The vectors $\mathbf{u}_i$ are rows of $\mathbf{U}$ and the vectors of $\mathbf{m}_j$ are columns of $\mathbf{M}$. These can be learned by minimizing the cost function:

$$S(U, M) = \sum_{i,j} \left(\mathbf{u}_i^T \mathbf{m}_j \right)^2 = |R - UM|^2$$

Understanding the Model as a Latent Feature Model

The "score" or label of a particular movie/user pair is:

$$r_{ij} = \mathbf{u}_i^T \mathbf{m}_j$$

for the *i*th user and *j*th movie. One way to think of this is in terms of the movie score only as a linear model:

$$r_{ij} = \sum_{k=1}^{K} = \alpha_k m_{k,j}$$

where the movie vector is now represented as a feature vector with K features, and the user vector $\alpha_k = u_{i,k}$ is simply a set of learned parameters from the data. We define how many features this internal representation will have, but the resulting output is really the only thing we are concerned with. These *latent features* are simply learned during the process of fitting the objective function, and may or may not be mapped to an actual, explainable feature set that we might understand.

In this sense, these latent features represent an intrinsic structure that is obtained simply through numerical optimization, and are akin to clusters that you might see in an unsupervised learning model. This makes the model interesting as a hybrid technique.

Yet another way to think of the same score is the user score as a linear model:

$$r_{ij} = \sum_{k=1}^{K} = \beta_k u_{i,k}$$

Now, the features are related to the user vector and the parameters are learned weights $\beta_k = m_{k,j}$. In fact, the training process alternates between solving for the weights of the movie vector while holding the user features constant, and solving for the weights of the user vector while holding the movie features constant. The numerical solution of the objective is obtained through this alternating least squares approach.

If this model were not so popular and powerful, it might be considered an interesting peculiarity of machine learning. It is, however, the most widely used machine learning algorithm in ecommerce, and is particularly effective for recommendation systems, whereby a list of recommended items can be generated for new users based on the preferences of similar previous users.

Unsupervised Learning Models

An unsupervised learning model is a model that attempts to make sense of unlabeled data, which is simply a list of features. No fitting to a function is used in these algorithms, and the connection to regression of any kind is no longer a useful way to think about the data. We are now searching for structure within the data itself to point us to a better understanding of it. Because of this unstructured way of looking at the data, it is almost necessarily an exploratory or preliminary step in determining what to do with the data.

Mostly, what we seek in these models are ways of organizing data into groups of points with similar features. These groups are referred to as *clusters*, and the process of grouping them is called *clustering*. Clusters are almost always identified as sets of points that are close to each other in *N*-dimensional space. There are many ways to attempt to identify these clusters, but the most popular is the K Means algorithm.

The basic idea behind K Means is to find clusters of data. The number of clusters is fixed at a value *K*. For a 2D feature vector with floating-point values, clusters are very easy to see, and the generalization to higher-dimensional vectors is straightforward.

K Means Clustering

The process of K Means clustering is simple:

1. Randomly assign centroids.
2. Compute distances of every point to every center.
3. Assign each point to the closest centroid.
4. Take all points of each cluster, and compute the average, or mean, position. These new positions (the K-Means) are assigned to be the centroid.
5. Rinse and repeat until the location of the centroid is unchanged.

The distances are computed as the Euclidian norm between two vectors, given as:

$$d_{ij}^2 = \left(\mathbf{x}_i - \mathbf{x}_j\right)^T \left(\mathbf{x}_i - \mathbf{x}_j\right)$$

The important thing to remember here is that these feature vectors have a geometric interpretation. As we will see, it takes extra effort to wrangle features like word counts into this formulation, but once you have, a very straightforward interpretation is available. Let's look at the simple example summarized in Figure 5-4.

In this notebook, we have created our own set of clusters by choosing five points at random. From these random points we add some random noise to generate our clusters. We then take those generated points and cluster them with the K Means model.

It is interesting to note that the clusters are different. But which is the best clustering? It is not entirely clear, and that is the point of unsupervised learning. There is no ground truth, or label available. In fact, it is the data scientist's job to look at data like this and begin to assign labels based on deeper understanding. Is $K = 5$ the best number of clusters to search for? Would more (or fewer) categories make for better clusters?

The other thing to note about this data is that, while the clustering is different from the control set, it is still very similar. Even more interesting is that the clustering that is revealed algorithmically is still better than what we might have been able to identify manually. So, while it may not give a perfect solution, it improves our understanding considerably. For larger datasets, this effect is more and more prominent.

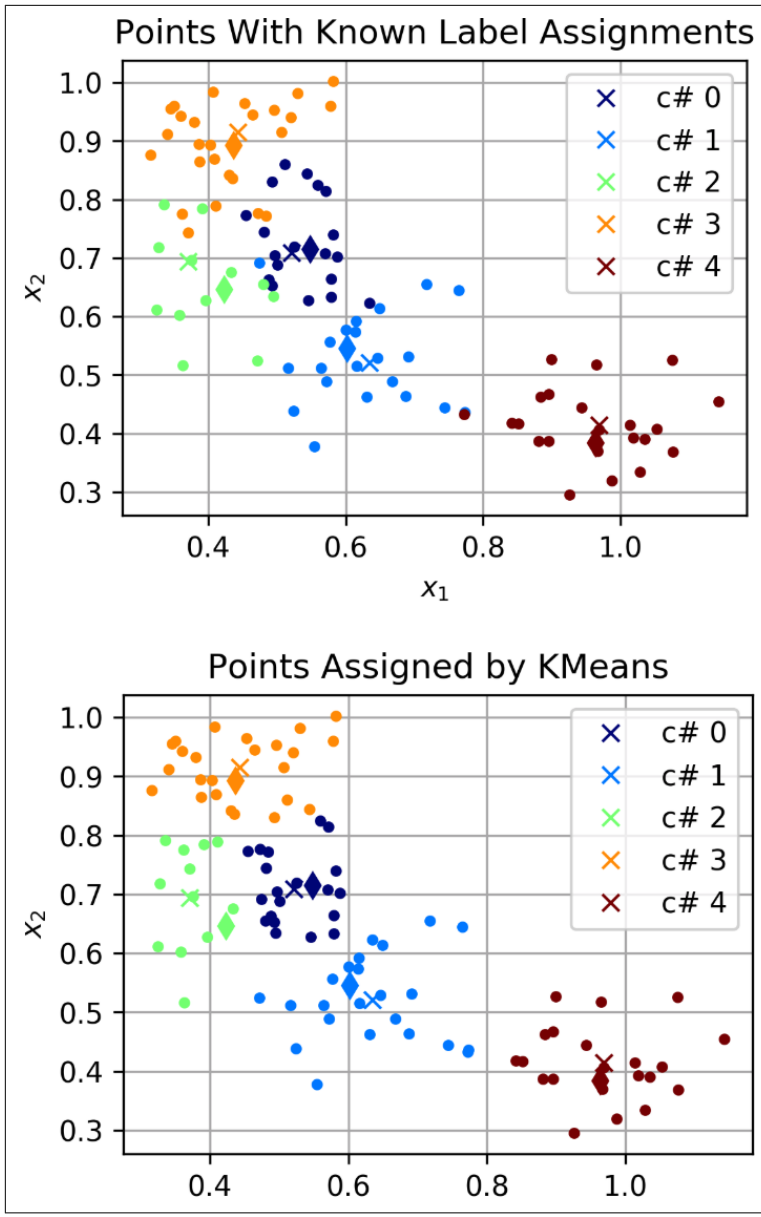

Figure 5-4. The top figure represents the cluster assignments from a dataset of "known clusters," while the bottom assignment represents cluster assignments from the K Means algorithm. Points marked by an x are the original center points, while the diamond-shaped points are the K Means centroids. Note that there are some differences in how labels are assigned.

From Clusters to Topics: Text Analytics with Unsupervised Learning

K Means Clustering Using Word2Vec

Using an *N*-dimensional feature vector for unsupervised learning is a powerful concept. It is such a useful way to think of data that, even in the case of text mining, an algorithm called Word2Vec is widely used because of its ability to represent words as a multidimensional vector. If we can take a body of words and consider them as a vector, we can begin to interpret words and sentences as points in *N*-dimensional space, and use traditional methods, including the K Means algorithm, to assign labels to unstructured data.

In fact, the most complex algorithm in this algorithm is *not* the K Means algorithm, but rather the preparation of the words to be used by the algorithm, which is the Word2Vec algorithm. Word2Vec is a member of the class of algorithms that carry out *word embedding*, which simply means that we take a collection of words and return a feature vector. The feature vector is always of the same dimension, even though the collection of words may not be.

An amazing property of Word2Vec is that vector embeddings can often result in interesting geometric interpretations, including additive properties like:

$$\text{w2v}(king) - \text{w2v}(queen) = \text{w2v}(man) - \text{w2v}(woman)$$

which suggests that an article, or body of text, can be a superposition of word vectors whose position in feature space relative to other articles can indicate their similarity. This is a very helpful insight.

The training of the Word2Vec model is more compute-intensive than the actual clustering. This is because the embedding uses a *neural network*. Neural networks are briefly discussed in the next chapter, but a detailed discussion of the machinery in the Word2Vec model is beyond the scope of this book.

The Latent Dirichlet Allocation

The Latent Dirichlet Allocation (LDA) is a significantly more complex unsupervised learning algorithm, and is widely used to model

topics. It has some important features that make it very useful for modeling topics in compendia of text:

- It can model sparse feature sets very well, which makes it ideal for dealing directly with word counts as feature vectors, rather than having to generate word embedding using topics as with Word2Vec.
- It admits multiple membership in topics, meaning every topic is able to emit its own distribution of words, and those words can come from multiple topics.

The notebooks for the LDA algorithm can be found on GitHub and a display of the discovered topics is shown in the second notebook. Figure 5-5 shows a word cloud for one of the discovered topics.

Figure 5-5. Word cloud for a topic discovered by the LDA algorithm applied to the Yahoo Newsgroups dataset

Summary

In this chapter, we have provided some of the canonical machine learning use cases, and have built upon the foundations presented in the previous chapter. In the next chapter, we will provide some advanced examples of machine learning, which will include deep learning and graph analytics.

CHAPTER 6

Advanced Machine Learning Examples and Applications

In this chapter we will provide advanced machine learning use cases, along with examples of how to use them. The examples here are slightly outside of the canon of Apache Spark machine learning, but are extremely helpful if you have occasion to use them.

Deep Learning Models with Spark and TensorFlow

The field of deep learning is yet another topic that is so vast that only a small portion of it can be covered here. In principle, deep learning is simply another type of supervised learning, as it almost always requires as its inputs a set of labeled data and features. There is a training stage and a prediction phase, and the workflow described earlier in Figure 5-3 is just as relevant to deep learning models as any supervised learning protocols.

The deep learning models themselves are, however, much more complex, and the software and hardware required for these models are typically outside the Spark ecosystem. There are many other very good frameworks to choose from, but we will provide an example with the very popular TensorFlow framework. The complexity of the inputs can also increase vastly, as well as the amount of data used to train the models. The inputs can have thousands of features, with millions of parameters, and the size of the training sets can be

orders of magnitude larger than the classic machine learning workflows. With models this large, it is difficult to interpret the parameters in any intuitive way, or even understand the importance of particular features. With all of these new elements to consider, then, we should think of deep learning as a completely new field of knowledge, while being mindful of its similarities to machine learning.

The Neural Network

The neural network is the fundamental model of deep learning. It is a remarkably versatile way of constructing complex models, and has evolved to the point that we can create arbitrarily large models with complete confidence in the ability to effectively train them (given enough data and compute time, of course). The original spirit of the neural network was to mimic the brain in its architecture. It has since been subjected to a much more numerically rigorous treatment, and is now thought of as an organizing principle for massive supervised learning efforts.

An *activation function* is defined as a scalar function of a linear model, which consists of weights and features, as described in Chapter 5. In Chapter 5 we also discussed a few different types of activation functions, and a notebook with these functions can be found on GitHub. As we mentioned, the original idea behind activation functions was to mimic the response of a biological neuron. Now, however, we choose them based on efficiency of calculation, and performance related to the numerical optimization steps.

We can regard the logistic regression model as a single neuron with the logistic function as the activation function. Of course, this is really not a network, because you need to connect neurons together in some way. Looking at Figure 6-1, we see one neuron that takes as its input a linear model and produces an output. The input could come directly from features, or from the output of other neurons. In turn, the output of that neuron can be taken directly as output, or as input features to other neurons. The network is arranged in layers. Figure 6-1 shows one neuron in the *n*th layer of a neural network. There are typically many neurons per layer, and each layer is connected only to the previous and the next layer. This architecture is often referred to as a *feed forward* neural network. For example, the inputs of layer *n* always come only from layer *m*, and the outputs of layer *n* are used only for the next layer. Layers can be connected in

many ways, but are almost always chosen to be *fully connected*, which simply means that every output of the previous layer feeds into every input of the following layer (see Figure 6-2).

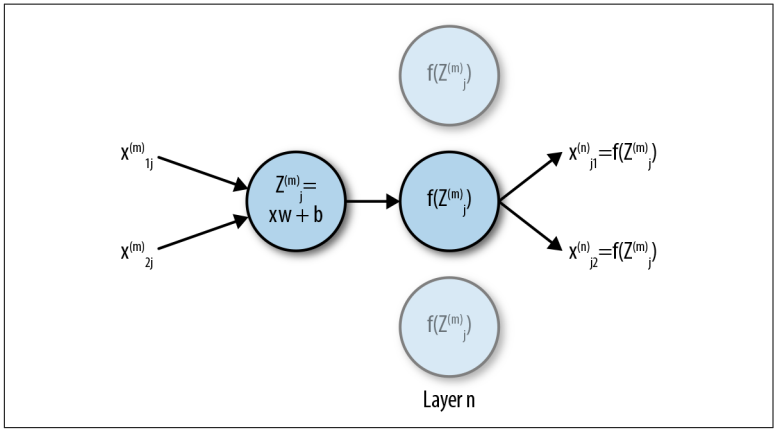

Figure 6-1. Linear model and activation function in a neural network. The neuron is the node where the activation function is applied.

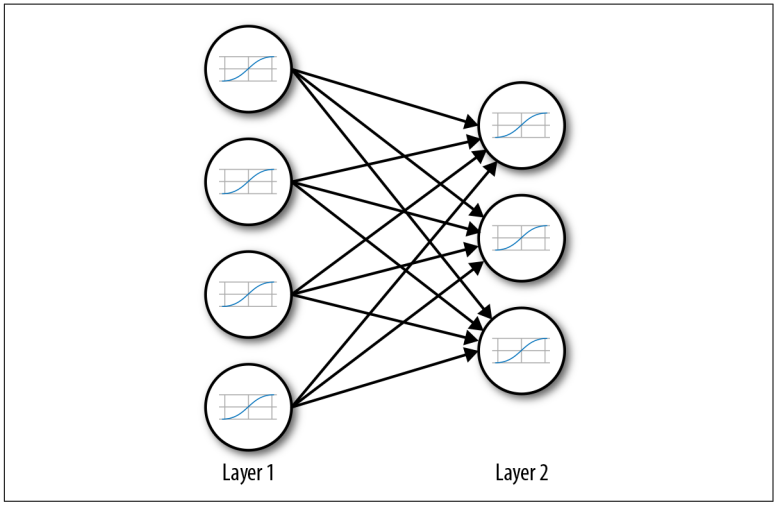

Figure 6-2. Layer 2 is a fully connected layer relative to layer 1

We will consider only three kinds of layers here: the input layer, the output layer, and the hidden layer. The *input layer* is the layer of neurons that receives the features and transforms them into something to feed into the network. The number of features in neural networks is typically much higher than that in machine learning.

There is also much less attention paid to feature scaling (see Chapter 4), as we can rely more heavily on the model to ameliorate these types of numerical issues. There are other preprocessing steps to consider, however (such as convolution for images), but we will not touch on them in this book.

Since images feature so prominently in neural network models, we will discuss how a simple image can become a feature vector. In Figure 6-3, we use simple black-and-white images of the number 7 and take each pixel and its intensity as a feature. The images here are 28 pixels by 28 pixels, and so the feature vector is 784 features. Each number 7 produces a very different feature vector, and it seems like a daunting task to train a model to identify so many different vectors. This is the beauty of a neural network, however, because as long as it has lots and lots of examples, it can learn to recognize these patterns. Simple machine learning models do not have enough parameters to manage such complexity. Figure 6-4 shows how the image is transformed into a feature vector that feeds into the input layer.

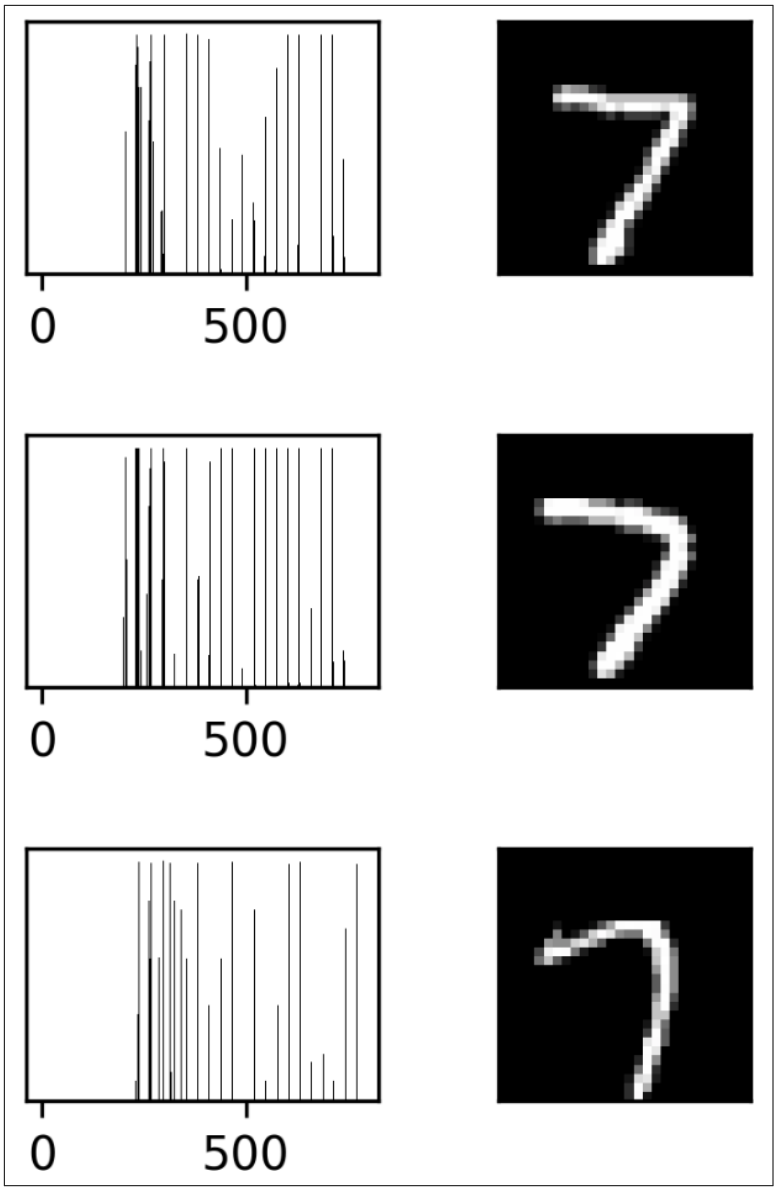

Figure 6-3. Three different 7s from the MNIST dataset, and their corresponding feature vectors. The left column represents the image as a vector with the pixel intensity as the value. Notice that there are similarities in the patterns, but the differences are significant enough to suggest that many examples are required for proper training.

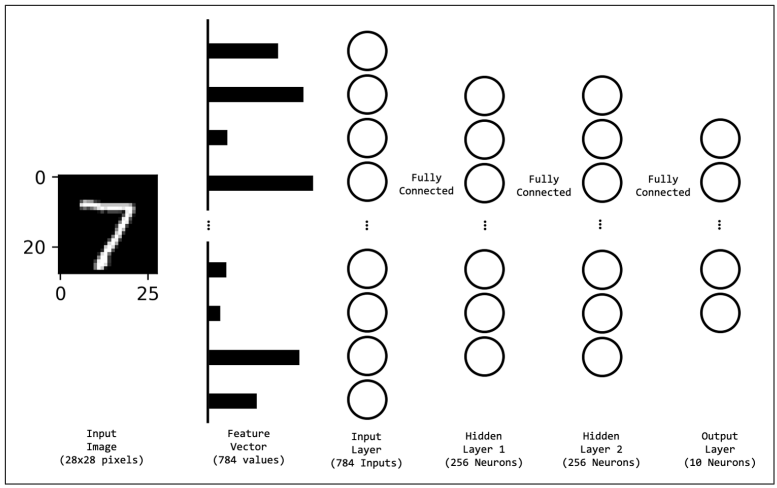

Figure 6-4. A fully connected neural network used to identify numbers in the MNIST dataset. The number image is converted into a feature vector and sent to a fully connected neural network with two hidden layers. The output layer consists of 10 neurons, corresponding to the 10 possible labels (0 through 9) for each input.

The *output layer* is the very last layer of neurons, and the output value is taken as the label. Each neuron typically reports a value between 0 and 1, which indicates membership in the category associated with that neuron. For the number identification example shown in Figure 6-4, there are 10 neurons in the output layer because there are 10 possible labels in the dataset (0 through 9). During training, these neurons are assigned "one hot" labels (neuron 0 has a value of 1 if the image is a 0). For the prediction, the neuron with the largest value is taken as the label.

Finally, *hidden layers* sit in between the input and output layers. The existence of hidden layers in a neural network is what gives the neural network depth, which is why we call these models *deep* learning models.

The input layer is constrained to have the number of neurons equal to the number of features, and the output layer is constrained to have the number of categories in the labels. There are no constraints, however, on the number of hidden layers, or the number of neurons per hidden layer. In fact, it is often recommended to have as many hidden layers as is computationally feasible, with less concern given to the overfitting issues that typically arise in simpler models

as the number of adjustable parameters increases. In practice, the design of the hidden layers comes from lots of tuning, experience, and intuition.

Training the Neural Network

The API for TensorFlow is sufficiently mature that we can call the numerical optimization routines while being blissfully unaware of their inner workings in neural networks. This is the culmination of years of research and fine tuning. For a mature framework like TensorFlow, we can rely on these algorithms to perform as expected. One of the breakthroughs in neural networks was the implementation of the *back propagation algorithm*, which is a clever way of storing derivatives during the gradient calculation to prevent unnecessary overhead. This all happens automatically, as long as we define our network according to what the API is expecting.

Training a neural network follows essentially the same principles as training a machine learning model. Gradient descent algorithms are used to minimize a loss function, one of the most popular of which is the *cross-entropy* loss function, which has particularly nice numerical properties, including efficient derivative computation. There are many other issues to consider, however, such as the choice of activation function. For example, the activation functions in the hidden layers are typically either linear or ReLU, in order to remedy what is known as the *gradient vanishing* problem.

While these topics are fascinating and form the foundation of the field, we won't touch on them here in the interest of brevity. An example of a simple neural network is available on GitHub. This is really only the "Hello world" of deep learning, and there is so much more to understand about this field. We hope you continue to learn more!

Graph Analytics

Graph analytics is not always considered a part of the machine learning corpus. It is, however, a powerful way of understanding intricate connections between data that are really only discoverable within a graphical structure. Thus, as a data scientist you should always have this in your repertoire. Fortunately, if you have already gone through the trouble of setting up your Apache Spark environ-

ment, you will have ready access to the GraphX library, providing a low barrier to entry.

What Is a Graph and Why Should We Care?

A *graph* is a way of organizing and thinking about data that allows you to discover relationships that are not easily discoverable in other data formats. When we say "graph," we mean a data structure, not a graphical representation of data, like a plot. It is a data type or way of organizing data. For example, there are 10 *vertices* in Figure 6-5, and they have *relationships* to each other that are defined by 15 *edges*. Graphs with symmetric relationships $e_{ab} = e_{ba}$ are called *undirected graphs*. In Apache Spark, these relationships do not need to be symmetric, $e_{ab} \neq e_{ba}$, and these graphs are called *directed graphs*.

The most important thing that graphs can discover is an indirect relationship between entities. The classic example is a social graph. While many data structures make it easy to discover all the people who are friends with you, it is not as easy to efficiently discover who your friends' friends are. While your most familiar idea of a social graph is likely to be undirected, it is entirely possible to construct a directed social graph, where friendships do not need to be mutual. Each relationship can actually be completely different. For example, in the relationships `a is the father of b`, and `b is the daughter of a`, each edge has a different value e_{ab} = `is the father of`, and e_{ba} = `is the daughter of`. The relationships need not be reciprocated in any way. For example, if e_{ac} `is in love with`, it has no bearing on the relationship e_{ca} (regardless of how entity `a` feels about the whole arrangement!). See the simple graph example online, which is modified from the Apache Spark documentation and shows how some of the basic functions in the Graph API are used.

Figure 6-5 shows an example of a distant relationship that a graph traversal can yield. Let's see how entities `a` and `f` are related to each other. To traverse the graph from `f` to `a`, we follow the path `f` → `e` → `a`. Notice that there is no path from `a` to `f`, however.

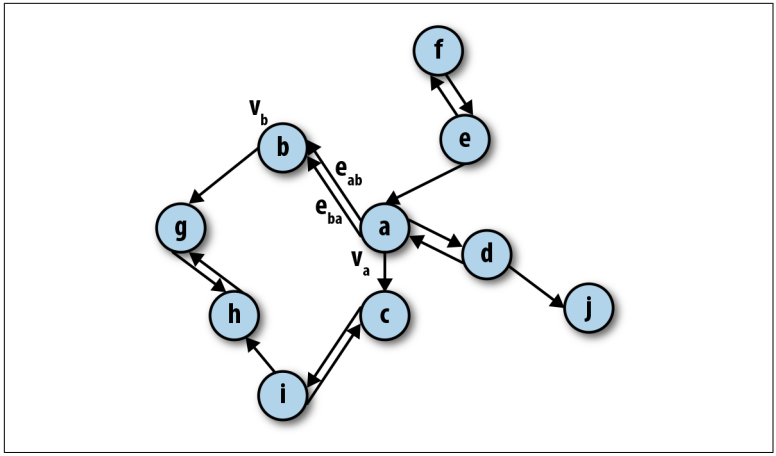

Figure 6-5. An example of a simple directed graph. Each vertex is a circle, and each edge is an arrow connecting two circles. Vertices a and b, along with the edges connecting them, are also shown.

The body of work around graph traversal algorithms is called *graph theory*. Many of these algorithms involve some sort of path construction through the graph. The most famous of these is the *Dijkstra algorithm*, which will find the shortest distance between two vertices, assuming that the edges are symmetric and represent distances. Another famous algorithm is the *PageRank algorithm*, which will take a large graph of web pages and the hyperlinks connecting them and return a rank-ordered list of pages based on the frequency of clicks from one page to the next. In the online example, a visualization of a similar *clickstream* network is constructed, showing small networks of Wikipedia entries as they relate to a particular page (IBM Watson). Figure 6-6 shows one of the graphs.

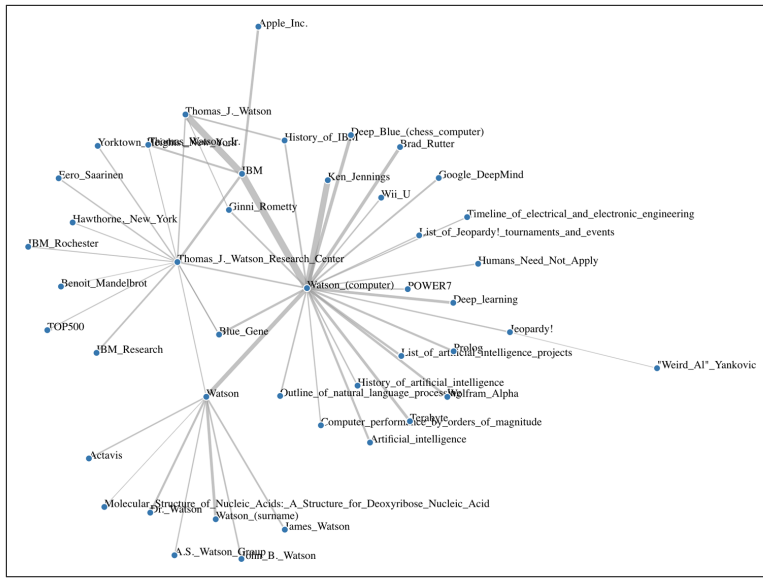

Figure 6-6. Clickstream graph of Wikipedia entries related to IBM Watson

Summary

In this final chapter, we have covered some additional examples in the field of machine learning that will be very helpful for you to understand. There are many more examples in deep learning and graph theory that you can run using the frameworks that are in place, and you are encouraged to try them out. The frameworks that we have installed for this book are very robust and well understood, and you should be able to grow your understanding of the field with these technologies as a starting point. Happy learning!

About the Author

Jerome Nilmeier is a data scientist and developer advocate at the Center for Open Source Data and Artificial Intelligence Technology (CODAIT) at IBM. His duties include development, enablement, and advocacy for IBM clients and the community at large, which includes teaching, outreach, consulting, and technical support for open source AI projects such as Apache Spark, TensorFlow, and other deep learning and big data technologies. Jerome has been with IBM since 2015.

He has a BS in Chemical Engineering from UC Berkeley, and a PhD in Computational Biophysics from UC San Francisco. He has carried out postdoctoral research in biophysics and bioinformatics at UC Berkeley, Lawrence Berkeley and Livermore Laboratories, and at Stanford as an OpenMM Fellow. Just prior to joining IBM, he completed the Insight Data Engineering program in late 2014.