

Python Machine Learning Blueprints

Second Edition

Put your machine learning concepts to the test by developing real-world smart projects



Alexander Combs and Michael Roman

Packt

www.packt.com

Python Machine Learning

Blueprints

Second Edition

Put your machine learning concepts to the test by developing real-world smart projects

Alexander Combs
Michael Roman

Packt

BIRMINGHAM - MUMBAI

Python Machine Learning Blueprints

Second Edition

Copyright © 2019 Packt Publishing

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

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

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

Commissioning Editor: Sunith Shetty

Acquisition Editor: Varsha Shetty

Content Development Editor: Snehal Kolte

Technical Editor: Naveen Sharma

Copy Editor: Safis Editing

Project Coordinator: Manthan Patel

Proofreader: Safis Editing

Indexer: Mariammal Chettiar

Graphics: Jisha Chirayil

Production Coordinator: Arvindkumar Gupta

First published: July 2016

Second edition: January 2019

Production reference: 1310119

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham

B3 2PB, UK.

ISBN 978-1-78899-417-0

www.packtpub.com



mapt.io

Mapt is an online digital library that gives you full access to over 5,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

Why subscribe?

- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Mapt is fully searchable
- Copy and paste, print, and bookmark content

Packt.com

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.packt.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at customercare@packtpub.com for more details.

At www.packt.com, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.

Contributors

About the authors

Alexander Combs is an experienced data scientist, strategist, and developer with a background in financial data extraction, natural language processing and generation, and quantitative and statistical modeling. He currently lives and works in New York City.

Writing a book is truly a massive undertaking that would not be possible without the support of others. I would like to thank my family for their love and encouragement and Jocelyn for her patience and understanding. I owe all of you tremendously.

Michael Roman is a data scientist at The Atlantic, where he designs, tests, analyzes, and productionizes machine learning models to address a range of business topics. Prior to this he was an associate instructor at a full-time data science immersive program in New York City. His interests include computer vision, propensity modeling, natural language processing, and entrepreneurship.

About the reviewer

Saurabh Chhajed is a machine learning and big data engineer with 9 years of professional experience in the enterprise application development life cycle using the latest frameworks, tools, and design patterns. He has experience of designing and implementing some of the most widely used and scalable customer-facing recommendation systems with extensive usage of the big data ecosystem—the batch, real-time, and machine learning pipeline. He has also worked for some of the largest investment banks, credit card companies, and manufacturing companies around the world, implementing a range of robust and scalable product suites.

Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit authors.packtpub.com and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Table of Contents

Preface	1
Chapter 1: The Python Machine Learning Ecosystem	6
Data science/machine learning workflow	7
Acquisition	8
Inspection	8
Preparation	9
Modeling	9
Evaluation	9
Deployment	10
Python libraries and functions for each stage of the data science workflow	10
Acquisition	10
Inspection	11
The Jupyter Notebook	11
Pandas	13
Visualization	21
The matplotlib library	23
The seaborn library	29
Preparation	32
map	33
apply	34
applymap	35
groupby	36
Modeling and evaluation	38
Statsmodels	38
Scikit-learn	42
Deployment	47
Setting up your machine learning environment	48
Summary	48
Chapter 2: Build an App to Find Underpriced Apartments	49
Sourcing apartment listing data	50
Pulling down listing data	50
Pulling out the individual data points	57
Parsing data	61
Inspecting and preparing the data	64
Sneak-peek at the data types	66
Visualizing our data	69
Visualizing the data	77
Modeling the data	79

Forecasting	82
Extending the model	86
Summary	86
Chapter 3: Build an App to Find Cheap Airfares	87
Sourcing airfare pricing data	88
Retrieving fare data with advanced web scraping	89
Creating a link	91
Parsing the DOM to extract pricing data	93
Parsing	96
Identifying outlier fares with anomaly detection techniques	106
Sending real-time alerts using IFTTT	113
Putting it all together	118
Summary	121
Chapter 4: Forecast the IPO Market Using Logistic Regression	122
The IPO market	123
What is an IPO?	123
Recent IPO market performance	124
Working on the DataFrame	126
Analyzing the data	131
Summarizing the performance of the stocks	132
Baseline IPO strategy	136
Data cleansing and feature engineering	139
Adding features to influence the performance of an IPO	139
Binary classification with logistic regression	141
Creating the target for our model	143
Dummy coding	143
Examining the model performance	145
Generating the importance of a feature from our model	149
Random forest classifier method	151
Summary	153
Chapter 5: Create a Custom Newsfeed	154
Creating a supervised training set with Pocket	155
Installing the Pocket Chrome Extension	155
Using the Pocket API to retrieve stories	157
Using the Embedly API to download story bodies	164
Basics of Natural Language Processing	166
Support Vector Machines	169
IFTTT integration with feeds, Google Sheets, and email	172
Setting up news feeds and Google Sheets through IFTTT	172
Setting up your daily personal newsletter	182
Summary	188
Chapter 6: Predict whether Your Content Will Go Viral	189

What does research tell us about virality?	190
Sourcing shared counts and content	192
Exploring the features of shareability	195
Exploring image data	196
Clustering	199
Exploring the headlines	202
Exploring the story content	207
Building a predictive content scoring model	210
Evaluating the model	213
Adding new features to our model	216
Summary	218
Chapter 7: Use Machine Learning to Forecast the Stock Market	219
Types of market analysis	221
What does research tell us about the stock market?	221
So, what exactly is a momentum strategy?	222
How to develop a trading strategy	223
Analysis of the data	225
Volatility of the returns	227
Daily returns	229
Statistics for the strategies	230
The mystery strategy	232
Building the regression model	237
Performance of the model	239
Dynamic time warping	246
Evaluating our trades	249
Summary	250
Chapter 8: Classifying Images with Convolutional Neural Networks	251
Image-feature extraction	252
Convolutional neural networks	255
Network topology	256
Convolutional layers and filters	258
Max pooling layers	268
Flattening	270
Fully-connected layers and output	271
Building a convolutional neural network to classify images in the Zalando Research dataset, using Keras	272
Summary	290
Chapter 9: Building a Chatbot	291
The Turing Test	292
The history of chatbots	293
The design of chatbots	298
Building a chatbot	303

Sequence-to-sequence modeling for chatbots	309
Summary	320
Chapter 10: Build a Recommendation Engine	321
Collaborative filtering	322
So, what's collaborative filtering?	322
Predicting the rating for the product	325
Content-based filtering	329
Hybrid systems	330
Collaborative filtering	330
Content-based filtering	330
Building a recommendation engine	331
Summary	348
Chapter 11: What's Next?	349
Summary of the projects	349
Summary	353
Other Books You May Enjoy	354
Index	357

Preface

Machine learning is transforming the way we understand and interact with the world around us. This book is the perfect guide for you to put your knowledge and skills into practice and use the Python ecosystem to cover the key domains in machine learning. This second edition covers a range of libraries from the Python ecosystem, including TensorFlow and Keras, to help you implement real-world machine learning projects.

The book begins by giving you an overview of machine learning with Python. With the help of complex datasets and optimized techniques, you'll learn how to apply advanced concepts and popular machine learning algorithms to real-world projects. Next, you'll cover projects in domains such as predictive analytics to analyze the stock market, and recommendation systems for GitHub repositories. In addition to this, you'll also work on projects from the NLP domain to create a custom news feed using frameworks such as scikit-learn, TensorFlow, and Keras. Following this, you'll learn how to build an advanced chatbot, and scale things up using PySpark. In the concluding chapters, you can look forward to exciting insights into deep learning and even create an application using computer vision and neural networks.

By the end of this book, you'll be able to analyze data seamlessly and make a powerful impact through your projects.

Who this book is for

This book is for machine learning practitioners, data scientists, and deep learning enthusiasts who want to take their machine learning skills to the next level by building real-world projects. This intermediate-level guide will help you to implement libraries from the Python ecosystem to build a variety of projects addressing various machine learning domains.

What this book covers

Chapter 1, *The Python Machine Learning Ecosystem*, discusses the features of key libraries and explains how to prepare your environment to best utilize them.

Chapter 2, *Build an App to Find Underpriced Apartments*, explains how to create a machine learning application that will make finding the right apartment a little bit easier.

Chapter 3, *Build an App to Find Cheap Airfares*, covers how to build an application that continually monitors fare pricing, checking for anomalous prices that will generate an alert we can quickly act on.

Chapter 4, *Forecast the IPO Market Using Logistic Regression*, takes a closer look at the IPO market. We'll see how we can use machine learning to help us decide which IPOs are worth a closer look and which ones we may want to take a pass on.

Chapter 5, *Create a Custom Newsfeed*, explains how to build a system that understands your taste in news, and will send you a personally tailored newsletter each day.

Chapter 6, *Predict whether Your Content Will Go Viral*, tries to unravel some of the mysteries. We'll examine some of the most commonly shared content and attempt to find the common elements that differentiate it from content people were less willing to share.

Chapter 7, *Use Machine Learning to Forecast the Stock Market*, discusses how to build and test a trading strategy. We'll spend more time, however, on how *not* to do it.

Chapter 8, *Classifying Images with Convolutional Neural Networks*, details the process of creating a computer vision application using deep learning.

Chapter 9, *Building a Chatbot*, explains how to construct a chatbot from scratch. Along the way, we'll learn more about the history of the field and its future prospects.

Chapter 10, *Build a Recommendation Engine*, explores the different varieties of recommendation systems. We'll see how they're implemented commercially and how they work. Finally, we'll implement our own recommendation engine for finding GitHub repositories.

Chapter 11, *What's Next?*, summarizes what has been covered so far in this book and what the next steps are from this point on. You will learn how to apply the skills you have gained to other projects, real-life challenges in building and deploying machine learning models, and other common technologies that data scientists frequently use.

To get the most out of this book

Knowledge of Python programming and machine learning concepts will be helpful.

Download the example code files

You can download the example code files for this book from your account at www.packt.com. If you purchased this book elsewhere, you can visit www.packt.com/support and register to have the files emailed directly to you.

You can download the code files by following these steps:

1. Log in or register at www.packt.com.
2. Select the **SUPPORT** tab.
3. Click on **Code Downloads & Errata**.
4. Enter the name of the book in the **Search** box and follow the onscreen instructions.

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

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

The code bundle for the book is also hosted on GitHub at <https://github.com/PacktPublishing/Python-Machine-Learning-Blueprints-Second-Edition>. In case there's an update to the code, it will be updated on the existing GitHub repository.

We also have other code bundles from our rich catalog of books and videos available at <https://github.com/PacktPublishing/>. Check them out!

Download the color images

We also provide a PDF file that has color images of the screenshots/diagrams used in this book. You can download it here: http://www.packtpub.com/sites/default/files/downloads/9781788994170_ColorImages.pdf.

Conventions used

There are a number of text conventions used throughout this book.

CodeInText: Indicates code words in text, database table names, folder names, filenames, file extensions, path names, dummy URLs, user input, and Twitter handles. Here is an example: "Let's take a look at a sample interaction, using `requests` to pull down data from GitHub's API. Here, we will make a call to the API and request a list of starred repositories for a user."

Any command-line input or output is written as follows:

```
import requests

r = requests.get("https://api.github.com/users/acombs/starred")

r.json()
```

Bold: Indicates a new term, an important word, or words that you see onscreen. For example, words in menus or dialog boxes appear in the text like this. Here is an example: "For Chrome, go to the Google app store and look for the **Extensions** section."

Warnings or important notes appear like this.



Tips and tricks appear like this.



Get in touch

Feedback from our readers is always welcome.

General feedback: If you have questions about any aspect of this book, mention the book title in the subject of your message and email us at customercare@packtpub.com.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you would report this to us. Please visit www.packt.com/submit-errata, selecting your book, clicking on the Errata Submission Form link, and entering the details.

Piracy: If you come across any illegal copies of our works in any form on the Internet, we would be grateful if you would provide us with the location address or website name. Please contact us at copyright@packt.com with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit authors.packtpub.com.

Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions, we at Packt can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about Packt, please visit packt.com.

1

The Python Machine Learning Ecosystem

Machine learning is rapidly changing our world. As the centerpiece of artificial intelligence, it is difficult to go a day without reading how it will lead us into either a techno-utopia along the lines of the Singularity, or into some sort of global Blade Runner-esque nightmare scenario. While pundits may enjoy discussing these hyperbolic futures, the more mundane reality is that machine learning is rapidly becoming a fixture of our daily lives. Through subtle but progressive improvements in how we interact with computers and the world around us, machine learning is progressively making our lives better.

If you shop at online retailers such as Amazon.com, use streaming music or movie services such as Spotify or Netflix, or have even just done a Google search, you have encountered an application that utilizes machine learning. These services collect vast amounts of data—much of it from their users—that is used to build models that improve the user experience.

It's an ideal time to dive into developing machine learning applications, and, as you will discover, Python is an ideal choice with which to develop them. Python has a deep and active developer community, many with roots in the scientific community. This heritage has provided Python with an unparalleled array of libraries for scientific computing. In this book, we will discuss and use a number of the libraries included in this **Python Scientific Stack**.

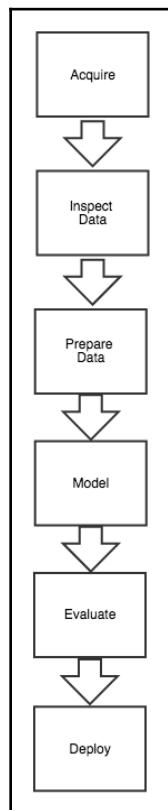
In the chapters that follow, we'll learn how to build a wide variety of machine learning applications step by step. Before we begin in earnest though, we'll spend the remainder of this chapter discussing the features of these key libraries and how to prepare your environment to best utilize them.

These are the topics that will be covered in this chapter:

- The data science/machine learning workflow
- Libraries for each stage of the workflow
- Setting up your environment

Data science/machine learning workflow

Building machine learning applications, while similar in many respects to the standard engineering paradigm, differs in one crucial aspect: the need to work with data as a raw material. The success of your project will, in large part, depend on the quality of the data you acquire, as well as your handling of that data. And because working with data falls into the domain of data science, it is helpful to understand the data science workflow:



Data science workflow

The process involves these six steps in the following order:

1. Acquisition
2. Inspection
3. Preparation
4. Modeling
5. Evaluation
6. Deployment

Frequently, there is a need to circle back to prior steps, such as when inspecting and preparing the data, or when evaluating and modeling, but the process at a high level can be as described in the preceding list.

Let's now discuss each step in detail.

Acquisition

Data for machine learning applications can come from any number of sources; it may be emailed to you as a CSV file, it may come from pulling down server logs, or it may require building a custom web scraper. Data is also likely to exist in any number of formats. In most cases, you will be dealing with text-based data, but, as we'll see, machine learning applications may just as easily be built that utilize images or even video files. Regardless of the format, once you have secured the data, it is crucial that you understand what's in the data, as well as what isn't.

Inspection

Once you have acquired your data, the next step is to inspect it. The primary goal at this stage is to sanity check the data, and the best way to accomplish this is to look for things that are either impossible or highly unlikely. As an example, if the data has a unique identifier, check to see that there is indeed only one; if the data is price-based, check that it is always positive; and whatever the data type, check the most extreme cases. Do they make sense? A good practice is to run some simple statistical tests on the data, and visualize it. The outcome of your models is only as good as the data you put in, so it is crucial to get this step right.

Preparation

When you are confident you have your data in order, next you will need to prepare it by placing it in a format that is amenable to modeling. This stage encompasses a number of processes, such as filtering, aggregating, imputing, and transforming. The type of actions you need to take will be highly dependent on the type of data you're working with, as well as the libraries and algorithms you will be utilizing. For example, if you are working with natural language-based texts, the transformations required will be very different from those required for time-series data. We'll see a number of examples of these types of transformations throughout the book.

Modeling

Once the data preparation is complete, the next phase is modeling. Here, you will be selecting an appropriate algorithm and using the data to train your model. There are a number of best practices to adhere to during this stage, and we will discuss them in detail, but the basic steps involve splitting your data into training, testing, and validation sets. This splitting up of the data may seem illogical—especially when more data typically yields better models—but as we'll see, doing this allows us to get better feedback on how the model will perform in the real world, and prevents us from the cardinal sin of modeling: overfitting. We will talk more about this in later chapters.

Evaluation

So, now you've got a shiny new model, but exactly how good is that model? This is the question that the evaluation phase seeks to answer. There are a number of ways to measure the performance of a model, and again it is largely dependent on the type of data you are working with and the type of model used, but on the whole, we are seeking to answer the question of how close the model's predictions are to the actual value. There is an array of confusing sounding terms, such as root mean-square error, or Euclidean distance, or F1 score. But in the end, they are all just a measure of distance between the actual prediction and the estimated prediction.

Deployment

Once you are comfortable with the performance of your model, you'll want to deploy it. This can take a number of forms depending on the use case, but common scenarios include utilization as a feature within another larger application, a bespoke web application, or even just a simple cron job.

Python libraries and functions for each stage of the data science workflow

Now that you have an understanding of each step in the data science workflow, we'll take a look at a selection of useful Python libraries and functions within those libraries for each step.

Acquisition

Since one of the more common ways to access data is through a RESTful API, one library that you'll want to be aware of is the Python Requests library, <http://www.python-requests.org/en/latest/>. Dubbed *HTTP for humans*, it makes interacting with APIs a clean and simple experience.

Let's take a look at a sample interaction, using `requests` to pull down data from GitHub's API. Here, we will make a call to the API and request a list of starred repositories for a user:

```
import requests r =  
requests.get(r"https://api.github.com/users/acombs/starred") r.json()
```

This will return a JSON of all the repositories the user has starred, along with attributes about each. Here is a snippet of the output for the preceding call:

```
In [1]: import requests
In [2]: r = requests.get("https://api.github.com/users/acombs/starred")

In [3]: r.json()
Out[3]:
[{'archive_url': 'https://api.github.com/repos/kennethreitz/requests/{archive_format}{{ref}}',
 'assignees_url': 'https://api.github.com/repos/kennethreitz/requests/assignees{/user}',
 'blobs_url': 'https://api.github.com/repos/kennethreitz/requests/git/blobs{/sha}',
 'branches_url': 'https://api.github.com/repos/kennethreitz/requests/branches{/branch}',
 'clone_url': 'https://github.com/kennethreitz/requests.git',
 'collaborators_url': 'https://api.github.com/repos/kennethreitz/requests/collaborators{/collaborator}',
 'comments_url': 'https://api.github.com/repos/kennethreitz/requests/comments{/number}',
 'commits_url': 'https://api.github.com/repos/kennethreitz/requests/commits{/sha}',
 'compare_url': 'https://api.github.com/repos/kennethreitz/requests/compare/{base}...{head}',
 'contents_url': 'https://api.github.com/repos/kennethreitz/requests/contents{/+path}',
 'contributors_url': 'https://api.github.com/repos/kennethreitz/requests/contributors',
 'created_at': '2011-02-13T18:38:17Z',
```

Output snippet when we return a JSON of all the repositories

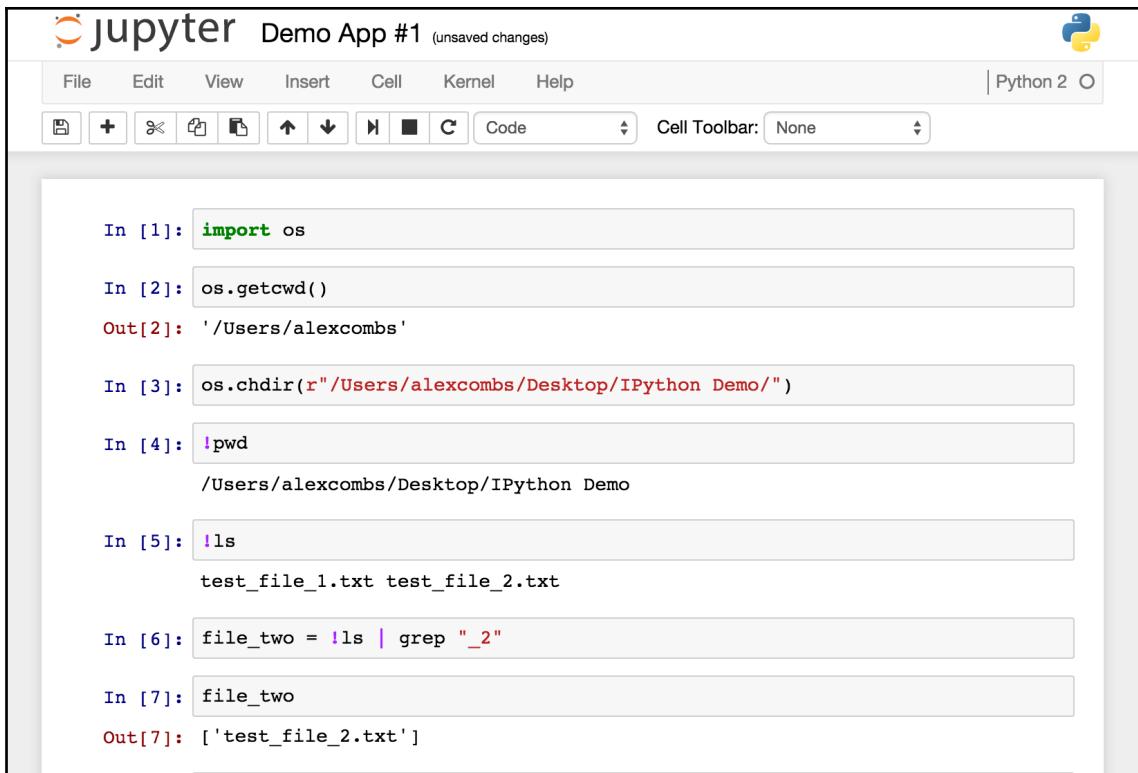
The `requests` library has an amazing number of features—far too many to cover here, but I do suggest you check out the documentation.

Inspection

Because inspecting your data is such a critical step in the development of machine learning applications, we'll now take an in-depth look at several libraries that will serve you well in this task.

The Jupyter Notebook

There are a number of libraries that will make the data inspection process easier. The first is Jupyter Notebook with IPython (<http://ipython.org/>). This is a fully-fledged, interactive computing environment, and it is ideal for data exploration. Unlike most development environments, Jupyter Notebook is a web-based frontend (to the IPython kernel) that is divided into individual code blocks or cells. Cells can be run individually or all at once, depending on the need. This allows the developer to run a scenario, see the output, then step back through the code, make adjustments, and see the resulting changes—all without leaving the notebook. Here is a sample interaction in the Jupyter Notebook:



The screenshot shows a Jupyter Notebook interface with the title "jupyter Demo App #1 (unsaved changes)". The toolbar includes File, Edit, View, Insert, Cell, Kernel, Help, and a Python 2 kernel icon. Below the toolbar is a toolbar with icons for file operations, cell selection, and cell execution. The main area displays the following code interactions:

```
In [1]: import os
In [2]: os.getcwd()
Out[2]: '/Users/alexcombs'
In [3]: os.chdir(r"/Users/alexcombs/Desktop/IPython Demo/")
In [4]: !pwd
/Users/alexcombs/Desktop/IPython Demo
In [5]: !ls
test_file_1.txt test_file_2.txt
In [6]: file_two = !ls | grep "_2"
In [7]: file_two
Out[7]: ['test_file_2.txt']
```

Sample interaction in the Jupyter Notebook

You will notice that we have done a number of things here and have interacted with not only the IPython backend, but the terminal shell as well. Here, I have imported the Python `os` library and made a call to find the current working directory (cell #2), which you can see is the output below my input code cell. I then changed directories using the `os` library in cell #3, but stopped utilizing the `os` library and began using Linux-based commands in cell #4. This is done by adding the `!` prepend to the cell. In cell #6, you can see that I was even able to save the shell output to a Python variable (`file_two`). This is a great feature that makes file operations a simple task.



Note that the results would obviously differ slightly on your machine, since this displays information on the user under which it runs.

Now, let's take a look at some simple data operations using the notebook. This will also be our first introduction to another indispensable library, pandas.

Pandas

Pandas is a remarkable tool for data analysis that aims to be the most powerful and flexible open source data analysis/manipulation tool available in any language. And, as you will soon see, if it doesn't already live up to this claim, it can't be too far off. Let's now take a look:

The screenshot shows a Jupyter Notebook interface with the title "jupyter Pandas Demo Last Checkpoint: an hour ago (unsaved changes)". The menu bar includes File, Edit, View, Insert, Cell, Kernel, Widgets, and Help. The toolbar includes icons for file operations like Open, Save, Run, and Cell. A status bar at the bottom right shows "Trusted" and "Python 3".

In [1]:

```
import pandas as pd
import sklearn
from sklearn import datasets

iris_data = datasets.load_iris()
```

In [2]:

```
iris_data.keys()
```

Out[2]:

```
dict_keys(['data', 'target', 'target_names', 'DESCR', 'feature_names'])
```

In [8]:

```
iris_data['target_names']
```

Out[8]:

```
array(['setosa', 'versicolor', 'virginica'], dtype='|<U10')
```

In [3]:

```
df_data = pd.DataFrame(iris_data['data'], columns=iris_data['feature_names'])
```

In [4]:

```
df_target = pd.DataFrame(iris_data['target'], columns=['species'])
```

In [5]:

```
df = pd.concat([df_data, df_target], axis=1)
```

In [6]:

```
df
```

Out[6]:

	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)	species
0	5.1	3.5	1.4	0.2	0
1	4.9	3.0	1.4	0.2	0

Importing the iris dataset

You can see from the preceding screenshot that I have imported a classic machine learning dataset, the `iris` dataset (also available at <https://archive.ics.uci.edu/ml/datasets/Iris>), using scikit-learn, a library we'll examine in detail later. I then passed the data into a `pandas` DataFrame, making sure to assign the column headers. One DataFrame contains flower measurement data, and the other DataFrame contains a number that represents the `iris` species. This is coded 0, 1, and 2 for `setosa`, `versicolor`, and `virginica` respectively. I then concatenated the two DataFrames.

For working with datasets that will fit on a single machine, pandas is the ultimate tool; you can think of it a bit like Excel on steroids. And, like the popular spreadsheet program, the basic units of operation are columns and rows of data that form tables. In the terminology of pandas, columns of data are series and the table is a DataFrame.

Using the same `iris` DataFrame we loaded previously, let's now take a look at a few common operations, including the following:

In [9]:	<code>df.head()</code>																																				
Out[9]:	<table><thead><tr><th></th><th>sepal length (cm)</th><th>sepal width (cm)</th><th>petal length (cm)</th><th>petal width (cm)</th><th>species</th></tr></thead><tbody><tr><td>0</td><td>5.1</td><td>3.5</td><td>1.4</td><td>0.2</td><td>0</td></tr><tr><td>1</td><td>4.9</td><td>3.0</td><td>1.4</td><td>0.2</td><td>0</td></tr><tr><td>2</td><td>4.7</td><td>3.2</td><td>1.3</td><td>0.2</td><td>0</td></tr><tr><td>3</td><td>4.6</td><td>3.1</td><td>1.5</td><td>0.2</td><td>0</td></tr><tr><td>4</td><td>5.0</td><td>3.6</td><td>1.4</td><td>0.2</td><td>0</td></tr></tbody></table>		sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)	species	0	5.1	3.5	1.4	0.2	0	1	4.9	3.0	1.4	0.2	0	2	4.7	3.2	1.3	0.2	0	3	4.6	3.1	1.5	0.2	0	4	5.0	3.6	1.4	0.2	0
	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)	species																																
0	5.1	3.5	1.4	0.2	0																																
1	4.9	3.0	1.4	0.2	0																																
2	4.7	3.2	1.3	0.2	0																																
3	4.6	3.1	1.5	0.2	0																																
4	5.0	3.6	1.4	0.2	0																																
In [10]:	<code>df['sepal length (cm)']</code>																																				
Out[10]:	0 5.1 1 4.9 2 4.7 3 4.6 4 5.0 5 5.4																																				

The first action was just to use the `.head()` command to get the first five rows. The second command was to select a single column from the DataFrame by referencing it by its column name. Another way we perform this **data slicing** is to use the `.iloc[row, column]` or `.loc[row, column]` notation. The former slices data using a numeric index for the columns and rows (positional indexing), while the latter uses a numeric index for the rows, but allows for using named columns (label-based indexing).

Let's select the first two columns and the first four rows using the `.iloc` notation. We'll then look at the `.loc` notation:

In [15]:	<code>df.head()</code>																																				
Out[15]:	<table><thead><tr><th></th><th>sepal length (cm)</th><th>sepal width (cm)</th><th>petal length (cm)</th><th>petal width (cm)</th><th>species</th></tr></thead><tbody><tr><td>0</td><td>5.1</td><td>3.5</td><td>1.4</td><td>0.2</td><td>0</td></tr><tr><td>1</td><td>4.9</td><td>3.0</td><td>1.4</td><td>0.2</td><td>0</td></tr><tr><td>2</td><td>4.7</td><td>3.2</td><td>1.3</td><td>0.2</td><td>0</td></tr><tr><td>3</td><td>4.6</td><td>3.1</td><td>1.5</td><td>0.2</td><td>0</td></tr><tr><td>4</td><td>5.0</td><td>3.6</td><td>1.4</td><td>0.2</td><td>0</td></tr></tbody></table>		sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)	species	0	5.1	3.5	1.4	0.2	0	1	4.9	3.0	1.4	0.2	0	2	4.7	3.2	1.3	0.2	0	3	4.6	3.1	1.5	0.2	0	4	5.0	3.6	1.4	0.2	0
	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)	species																																
0	5.1	3.5	1.4	0.2	0																																
1	4.9	3.0	1.4	0.2	0																																
2	4.7	3.2	1.3	0.2	0																																
3	4.6	3.1	1.5	0.2	0																																
4	5.0	3.6	1.4	0.2	0																																
In [14]:	<code>df.iloc[:3,:2]</code>																																				
Out[14]:	<table><thead><tr><th></th><th>sepal length (cm)</th><th>sepal width (cm)</th></tr></thead><tbody><tr><td>0</td><td>5.1</td><td>3.5</td></tr><tr><td>1</td><td>4.9</td><td>3.0</td></tr><tr><td>2</td><td>4.7</td><td>3.2</td></tr></tbody></table>		sepal length (cm)	sepal width (cm)	0	5.1	3.5	1	4.9	3.0	2	4.7	3.2																								
	sepal length (cm)	sepal width (cm)																																			
0	5.1	3.5																																			
1	4.9	3.0																																			
2	4.7	3.2																																			

Using the `.iloc` notation and the Python list slicing syntax, we were able to select a slice of this DataFrame.

Now, let's try something more advanced. We'll use a list iterator to select just the width feature columns:

```
In [23]: df.head()
```

```
Out[23]:
```

	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)	species
0	5.1	3.5	1.4	0.2	0
1	4.9	3.0	1.4	0.2	0
2	4.7	3.2	1.3	0.2	0
3	4.6	3.1	1.5	0.2	0
4	5.0	3.6	1.4	0.2	0

```
In [24]: df.loc[:3, [x for x in df.columns if 'width' in x]]
```

```
Out[24]:
```

	sepal width (cm)	petal width (cm)
0	3.5	0.2
1	3.0	0.2
2	3.2	0.2
3	3.1	0.2

What we have done here is create a list that is a subset of all columns. `df.columns` returns a list of all columns, and our iteration uses a conditional statement to select only those with `width` in the title. Obviously, in this situation, we could have just as easily typed out the columns we wanted into a list, but this gives you a sense of the power available when dealing with much larger datasets.

We've seen how to select slices based on their position within the DataFrame, but let's now look at another method to select data. This time, we will select a subset of the data based upon satisfying conditions that we specify:

1. Let's now see the unique list of `species` available, and select just one of those:

In [29]:	<code>df['species'].unique()</code>																																																												
Out[29]:	<code>array([0, 1, 2])</code>																																																												
In [30]:	<code>df[df['species']==2]</code>																																																												
Out[30]:	<table><thead><tr><th></th><th>sepal length (cm)</th><th>sepal width (cm)</th><th>petal length (cm)</th><th>petal width (cm)</th><th>species</th></tr></thead><tbody><tr><td>100</td><td>6.3</td><td>3.3</td><td>6.0</td><td>2.5</td><td>2</td></tr><tr><td>101</td><td>5.8</td><td>2.7</td><td>5.1</td><td>1.9</td><td>2</td></tr><tr><td>102</td><td>7.1</td><td>3.0</td><td>5.9</td><td>2.1</td><td>2</td></tr><tr><td>103</td><td>6.3</td><td>2.9</td><td>5.6</td><td>1.8</td><td>2</td></tr><tr><td>104</td><td>6.5</td><td>3.0</td><td>5.8</td><td>2.2</td><td>2</td></tr><tr><td>105</td><td>7.6</td><td>3.0</td><td>6.6</td><td>2.1</td><td>2</td></tr><tr><td>106</td><td>4.9</td><td>2.5</td><td>4.5</td><td>1.7</td><td>2</td></tr><tr><td>107</td><td>7.3</td><td>2.9</td><td>6.3</td><td>1.8</td><td>2</td></tr><tr><td>108</td><td>6.7</td><td>2.5</td><td>5.8</td><td>1.8</td><td>2</td></tr></tbody></table>		sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)	species	100	6.3	3.3	6.0	2.5	2	101	5.8	2.7	5.1	1.9	2	102	7.1	3.0	5.9	2.1	2	103	6.3	2.9	5.6	1.8	2	104	6.5	3.0	5.8	2.2	2	105	7.6	3.0	6.6	2.1	2	106	4.9	2.5	4.5	1.7	2	107	7.3	2.9	6.3	1.8	2	108	6.7	2.5	5.8	1.8	2
	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)	species																																																								
100	6.3	3.3	6.0	2.5	2																																																								
101	5.8	2.7	5.1	1.9	2																																																								
102	7.1	3.0	5.9	2.1	2																																																								
103	6.3	2.9	5.6	1.8	2																																																								
104	6.5	3.0	5.8	2.2	2																																																								
105	7.6	3.0	6.6	2.1	2																																																								
106	4.9	2.5	4.5	1.7	2																																																								
107	7.3	2.9	6.3	1.8	2																																																								
108	6.7	2.5	5.8	1.8	2																																																								

2. In the far-right column, you will notice that our DataFrame only contains data for the `Iris-virginica` species (represented by the 2) now. In fact, the size of the DataFrame is now 50 rows, down from the original 150 rows:

```
In [31]: df.count()  
  
Out[31]: sepal length (cm)    150  
          sepal width (cm)     150  
          petal length (cm)    150  
          petal width (cm)     150  
          species              150  
          dtype: int64  
  
In [32]: df[df['species']==2].count()  
  
Out[32]: sepal length (cm)    50  
          sepal width (cm)     50  
          petal length (cm)    50  
          petal width (cm)     50  
          species              50  
          dtype: int64
```

3. You can also see that the index on the left retains the original row numbers. If we wanted to save just this data, we could save it as a new DataFrame, and reset the index as shown in the following diagram:

```
In [33]: virginica = df[df['species']==2].reset_index(drop=True)  
  
In [34]: virginica  
  
Out[34]:  
      sepal length (cm)  sepal width (cm)  petal length (cm)  petal width (cm)  species  
0       6.3           3.3            6.0            2.5           2  
1       5.8           2.7            5.1            1.9           2  
2       7.1           3.0            5.9            2.1           2  
3       6.3           2.9            5.6            1.8           2  
4       6.5           3.0            5.8            2.2           2  
5       7.6           3.0            6.6            2.1           2  
6       4.9           2.5            4.5            1.7           2  
7       7.3           2.9            6.3            1.8           2  
8       6.7           2.5            5.8            1.8           2
```

4. We have selected data by placing a condition on one column; let's now add more conditions. We'll go back to our original DataFrame and add two conditions:

In [36]:	df[(df['species'] == 2) & (df['petal width (cm)'] > 2.2)]				
Out[36]:	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)	species
100	6.3	3.3	6.0	2.5	2
109	7.2	3.6	6.1	2.5	2
114	5.8	2.8	5.1	2.4	2
115	6.4	3.2	5.3	2.3	2
118	7.7	2.6	6.9	2.3	2
120	6.9	3.2	5.7	2.3	2
135	7.7	3.0	6.1	2.3	2
136	6.3	3.4	5.6	2.4	2

The DataFrame now only includes data from the `virginica` species with a petal width greater than `2.2`.

Let's now move on to using pandas to get some quick descriptive statistics from our `iris` dataset:

In [37]:	df.describe()				
Out[37]:	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)	species
count	150.000000	150.000000	150.000000	150.000000	150.000000
mean	5.843333	3.054000	3.758667	1.198667	1.000000
std	0.828066	0.433594	1.764420	0.763161	0.819232
min	4.300000	2.000000	1.000000	0.100000	0.000000
25%	5.100000	2.800000	1.600000	0.300000	0.000000
50%	5.800000	3.000000	4.350000	1.300000	1.000000
75%	6.400000	3.300000	5.100000	1.800000	2.000000
max	7.900000	4.400000	6.900000	2.500000	2.000000

With a call to the `.describe()` function, I have received a breakdown of the descriptive statistics for each of the relevant columns. (Notice that species was automatically removed as it is not relevant for this.) I could also pass in my own percentiles if I wanted more granular information:

```
In [38]: df.describe(percentiles=[.10,.20,.30,.40,.50,.60,.70,.80,.90])
```

Out[38]:

	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)	species
count	150.000000	150.000000	150.000000	150.000000	150.000000
mean	5.843333	3.054000	3.758667	1.198667	1.000000
std	0.828066	0.433594	1.764420	0.763161	0.819232
min	4.300000	2.000000	1.000000	0.100000	0.000000
10%	4.800000	2.500000	1.400000	0.200000	0.000000
20%	5.000000	2.700000	1.500000	0.200000	0.000000
30%	5.270000	2.800000	1.700000	0.400000	0.000000
40%	5.600000	3.000000	3.900000	1.160000	1.000000
50%	5.800000	3.000000	4.350000	1.300000	1.000000
60%	6.100000	3.100000	4.640000	1.500000	1.000000
70%	6.300000	3.200000	5.000000	1.800000	2.000000
80%	6.520000	3.400000	5.320000	1.900000	2.000000
90%	6.900000	3.610000	5.800000	2.200000	2.000000
max	7.900000	4.400000	6.900000	2.500000	2.000000

Next, let's check whether there is any correlation between these features. That can be done by calling `.corr()` on our DataFrame:

In [39]:	df.corr()																																				
Out[39]:	<table><thead><tr><th></th><th>sepal length (cm)</th><th>sepal width (cm)</th><th>petal length (cm)</th><th>petal width (cm)</th><th>species</th></tr></thead><tbody><tr><th>sepal length (cm)</th><td>1.000000</td><td>-0.109369</td><td>0.871754</td><td>0.817954</td><td>0.782561</td></tr><tr><th>sepal width (cm)</th><td>-0.109369</td><td>1.000000</td><td>-0.420516</td><td>-0.356544</td><td>-0.419446</td></tr><tr><th>petal length (cm)</th><td>0.871754</td><td>-0.420516</td><td>1.000000</td><td>0.962757</td><td>0.949043</td></tr><tr><th>petal width (cm)</th><td>0.817954</td><td>-0.356544</td><td>0.962757</td><td>1.000000</td><td>0.956464</td></tr><tr><th>species</th><td>0.782561</td><td>-0.419446</td><td>0.949043</td><td>0.956464</td><td>1.000000</td></tr></tbody></table>		sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)	species	sepal length (cm)	1.000000	-0.109369	0.871754	0.817954	0.782561	sepal width (cm)	-0.109369	1.000000	-0.420516	-0.356544	-0.419446	petal length (cm)	0.871754	-0.420516	1.000000	0.962757	0.949043	petal width (cm)	0.817954	-0.356544	0.962757	1.000000	0.956464	species	0.782561	-0.419446	0.949043	0.956464	1.000000
	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)	species																																
sepal length (cm)	1.000000	-0.109369	0.871754	0.817954	0.782561																																
sepal width (cm)	-0.109369	1.000000	-0.420516	-0.356544	-0.419446																																
petal length (cm)	0.871754	-0.420516	1.000000	0.962757	0.949043																																
petal width (cm)	0.817954	-0.356544	0.962757	1.000000	0.956464																																
species	0.782561	-0.419446	0.949043	0.956464	1.000000																																

The default returns the **Pearson correlation coefficient** for each row-column pair. This can be switched to **Kendall's Tau** or **Spearman's rank correlation coefficient** by passing in a method argument (for example, `.corr(method="spearman")` or `.corr(method="kendall")`).

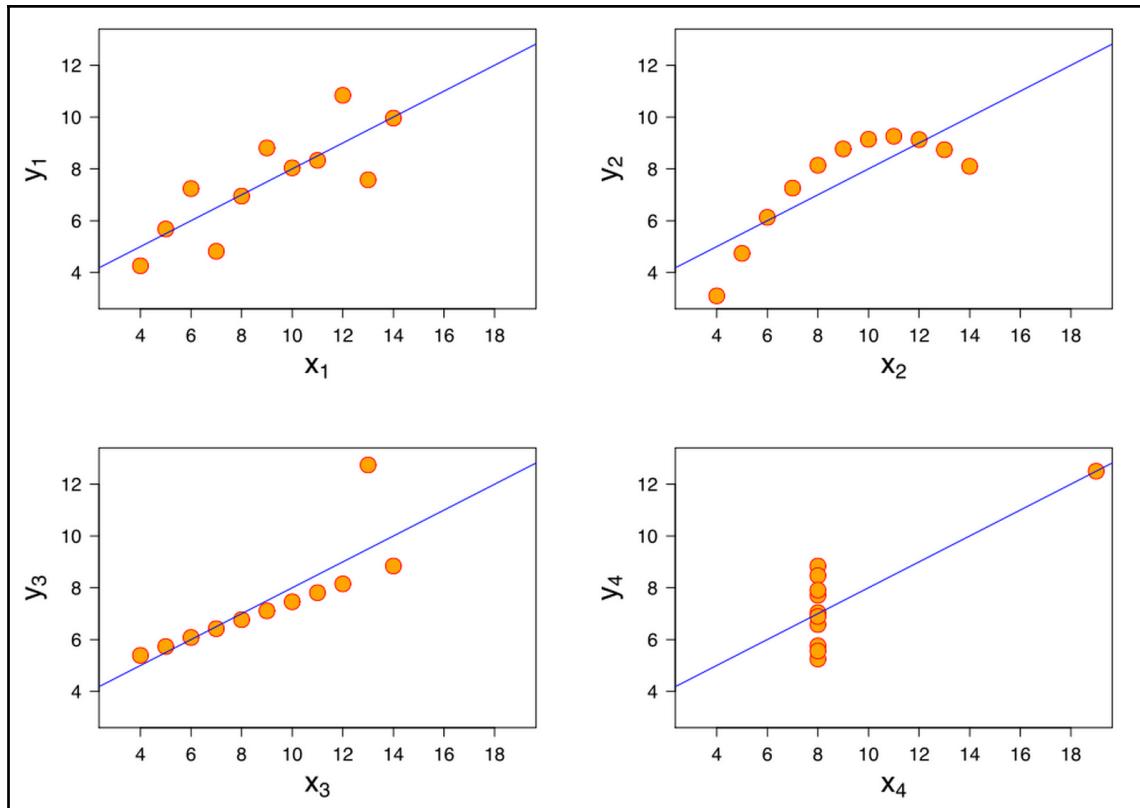
Visualization

So far, we have seen how to select portions of a DataFrame and how to get summary statistics from our data, but let's now move on to learning how to visually inspect the data. But first, why even bother with visual inspection? Let's see an example to understand why.

Here is the summary statistics for four distinct series of x and y values:

Series of x and y	Values
Mean of x	9
Mean of y	7.5
Sample variance of x	11
Sample variance of y	4.1
Correlation between x and y	0.816
Regression line	$y = 3.00 + 0.500x$

Based on the series having identical summary statistics, you might assume that these series would appear visually similar. You would, of course, be wrong. Very wrong. The four series are part of **Anscombe's quartet**, and they were deliberately created to illustrate the importance of visual data inspection. Each series is plotted as follows:



Clearly, we would not treat these datasets as identical after having visualized them. So, now that we understand the importance of visualization, let's take a look at a pair of useful Python libraries for this.

The matplotlib library

The first library we'll take a look at is `matplotlib`. The `matplotlib` library is the center of the Python plotting library universe. Originally created to emulate the plotting functionality of MATLAB, it grew into a fully-featured library in its own right with an enormous range of functionality. If you have not come from a MATLAB background, it can be hard to understand how all the pieces work together to create the graphs you see. I'll do my best to break down the pieces into logical components so you can get up to speed quickly. But before diving into `matplotlib` in full, let's set up our Jupyter Notebook to allow us to see our graphs inline. To do this, add the following lines to your `import` statements:

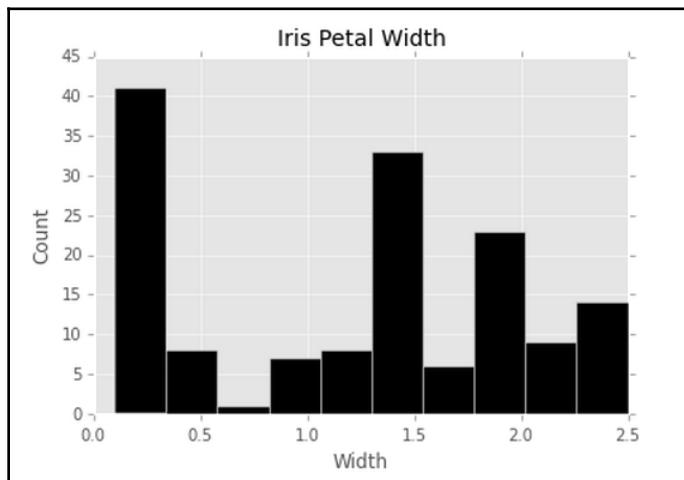
```
import matplotlib.pyplot as plt
plt.style.use('ggplot')
%matplotlib inline
```

The first line imports `matplotlib`, the second line sets the styling to approximate R's `ggplot` library (requires `matplotlib` 1.41 or greater), and the last line sets the plots so that they are visible within the notebook.

Now, let's generate our first graph using our `iris` dataset:

```
fig, ax = plt.subplots(figsize=(6, 4))
ax.hist(df['petal width (cm)'], color='black');
ax.set_ylabel('Count', fontsize=12)
ax.set_xlabel('Width', fontsize=12)
plt.title('Iris Petal Width', fontsize=14, y=1.01)
```

The preceding code generates the following output:



There is a lot going on even in this simple example, but we'll break it down line by line. The first line creates a single subplot with a width of 6 inches and a height of 4 inches. We then plot a histogram of the petal width from our `iris` DataFrame by calling `.hist()` and passing in our data. We also set the bar color to `black` here. The next two lines place labels on our `y` and `x` axes, respectively, and the final line sets the title for our graph. We tweak the title's `y` position relative to the top of the graph with the `y` parameter, and increase the font size slightly over the default. This gives us a nice histogram of our petal width data. Let's now expand on that, and generate histograms for each column of our `iris` dataset:

```
fig, ax = plt.subplots(2, 2, figsize=(6, 4))

ax[0][0].hist(df['petal width (cm)'], color='black');
ax[0][0].set_ylabel('Count', fontsize=12)
ax[0][0].set_xlabel('Width', fontsize=12)
ax[0][0].set_title('Iris Petal Width', fontsize=14, y=1.01)

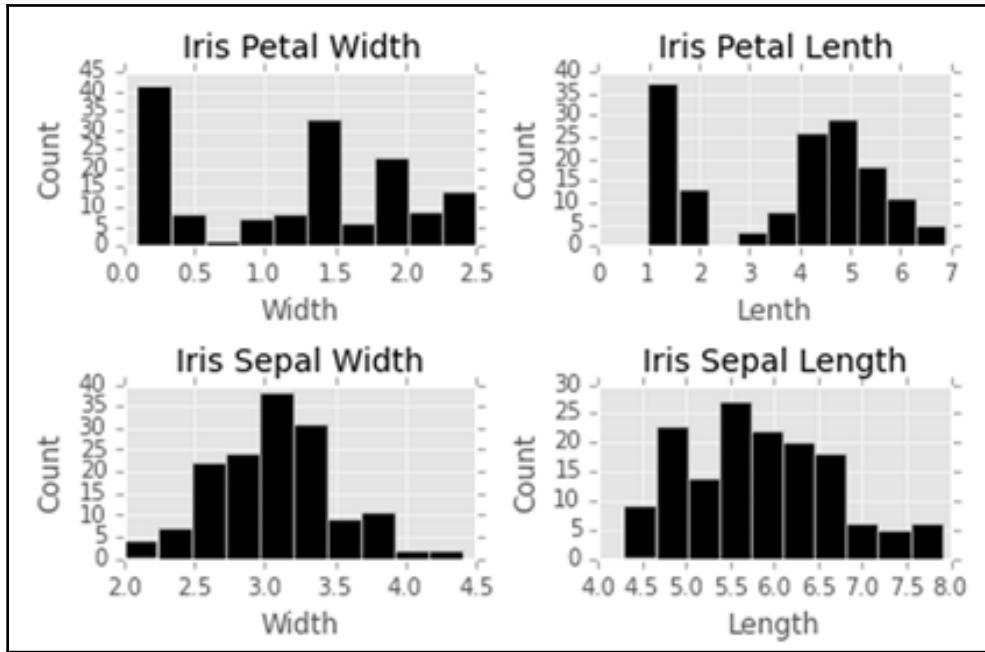
ax[0][1].hist(df['petal length (cm)'], color='black');
ax[0][1].set_ylabel('Count', fontsize=12)
ax[0][1].set_xlabel('Length', fontsize=12)
ax[0][1].set_title('Iris Petal Length', fontsize=14, y=1.01)

ax[1][0].hist(df['sepal width (cm)'], color='black');
ax[1][0].set_ylabel('Count', fontsize=12)
ax[1][0].set_xlabel('Width', fontsize=12)
ax[1][0].set_title('Iris Sepal Width', fontsize=14, y=1.01)

ax[1][1].hist(df['sepal length (cm)'], color='black');
ax[1][1].set_ylabel('Count', fontsize=12)
ax[1][1].set_xlabel('Length', fontsize=12)
ax[1][1].set_title('Iris Sepal Length', fontsize=14, y=1.01)

plt.tight_layout()
```

The output for the preceding code is shown in the following diagram:

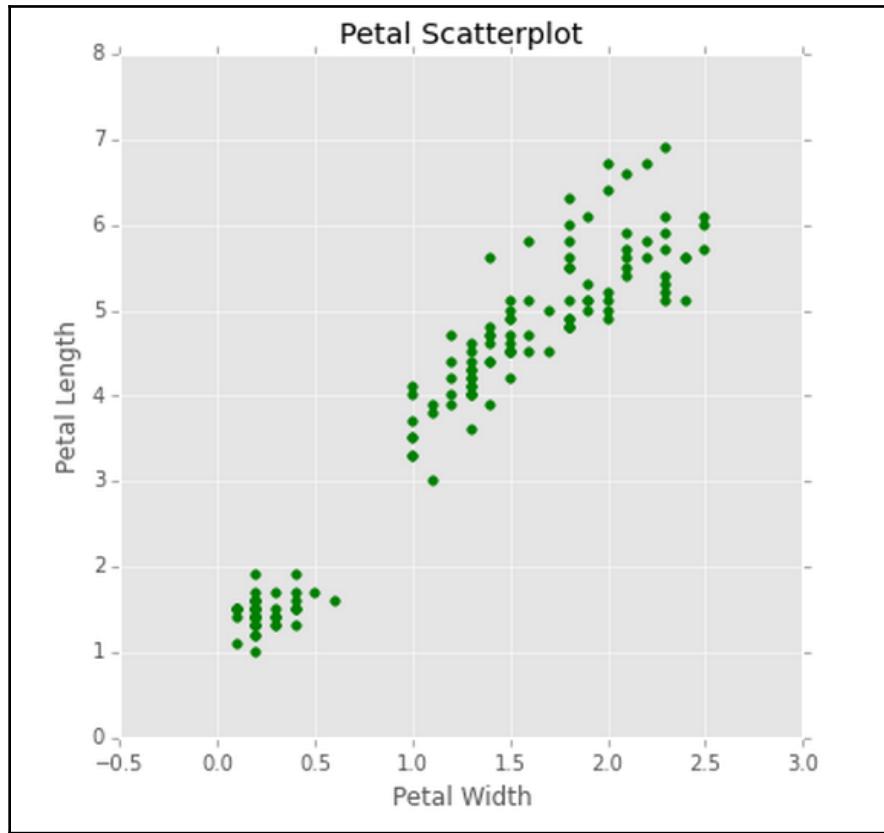


Obviously, this is not the most efficient way to code this, but it is useful for demonstrating how `matplotlib` works. Notice that instead of the single subplot object, `ax`, as we had in the first example, we now have four subplots, which are accessed through what is now the `ax` array. A new addition to the code is the call to `plt.tight_layout()`; this function will nicely auto-space your subplots to avoid crowding.

Let's now take a look at a few other types of plots available in `matplotlib`. One useful plot is a `scatterplot`. Here, we will plot the petal width against the petal length:

```
fig, ax = plt.subplots(figsize=(6, 6))
ax.scatter(df['petal width (cm)'], df['petal length (cm)'],
color='green')
ax.set_xlabel('Petal Width')
ax.set_ylabel('Petal Length')
ax.set_title('Petal Scatterplot')
```

The preceding code generates the following output:

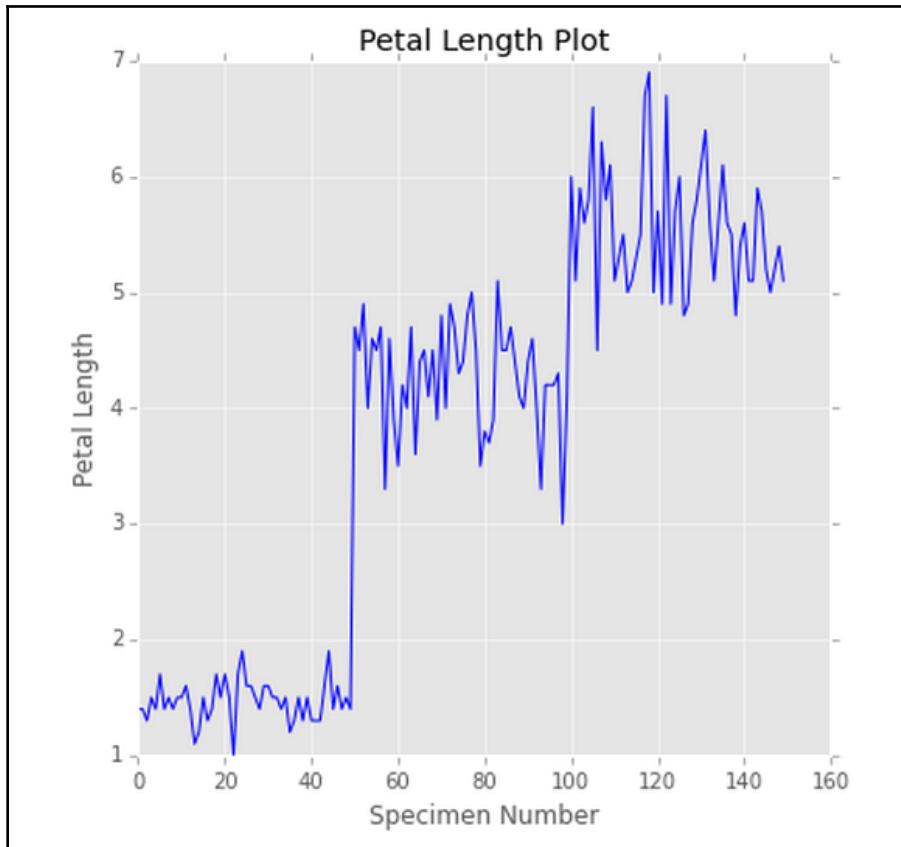


As before, we could add in multiple subplots to examine each facet.

Another plot we could examine is a simple line plot. Here, we will look at a plot of the petal length:

```
fig, ax = plt.subplots(figsize=(6, 6))
ax.plot(df['petal length (cm)'], color='blue')
ax.set_xlabel('Specimen Number')
ax.set_ylabel('Petal Length')
ax.set_title('Petal Length Plot')
```

The preceding code generates the following output:



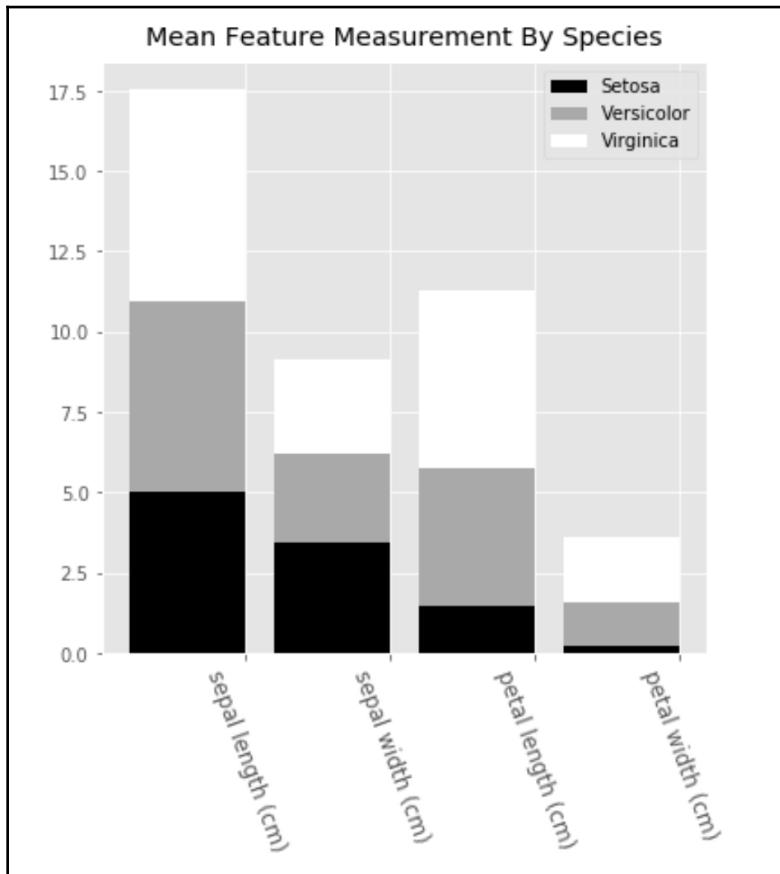
We can already begin to see, based on this simple line plot, that there are distinctive clusters of lengths for each species—remember our sample dataset had 50 ordered examples of each type. This tells us that petal length is likely to be a useful feature to discriminate between the species if we were to build a classifier.

Let's look at one final type of chart from the `matplotlib` library, the bar chart. This is perhaps one of the more common charts you'll see. Here, we'll plot a bar chart for the mean of each feature for the three species of irises, and to make it more interesting, we'll make it a stacked bar chart with a number of additional `matplotlib` features:

```
import numpy as np
fig, ax = plt.subplots(figsize=(6, 6))
bar_width = .8
labels = [x for x in df.columns if 'length' in x or 'width' in x]
```

```
set_y = [df[df['species']==0][x].mean() for x in labels]
ver_y = [df[df['species']==1][x].mean() for x in labels]
vir_y = [df[df['species']==2][x].mean() for x in labels]
x = np.arange(len(labels))
ax.bar(x, set_y, bar_width, color='black')
ax.bar(x, ver_y, bar_width, bottom=set_y, color='darkgrey')
ax.bar(x, vir_y, bar_width, bottom=[i+j for i,j in zip(set_y, ver_y)], color='white')
ax.set_xticks(x + (bar_width/2))
ax.set_xticklabels(labels, rotation=-70, fontsize=12);
ax.set_title('Mean Feature Measurement By Species', y=1.01)
ax.legend(['Setosa', 'Versicolor', 'Virginica'])
```

The output for the preceding snippet is given here:



To generate the bar chart, we need to pass the `x` and `y` values into the `.bar()` function. In this case, the `x` values will just be an array of the length of the features we are interested in—four here, or one for each column in our DataFrame. The `np.arange()` function is an easy way to generate this, but we could nearly as easily input this array manually. Since we don't want the `x` axis to display this as 1 through 4, we call the `.set_xticklabels()` function and pass in the column names we wish to display. To line up the `x` labels properly, we also need to adjust the spacing of the labels. This is why we set the `xticks` to `x` plus half the size of the `bar_width`, which we also set earlier at `0.8`. The `y` values come from taking the mean of each feature for each species. We then plot each by calling `.bar()`. It is important to note that we pass in a `bottom` parameter for each series, which sets the minimum `y` point and the maximum `y` point of the series below it. This creates the stacked bars. And finally, we add a legend, which describes each series. The names are inserted into the legend list in order of the placement of the bars from top to bottom.

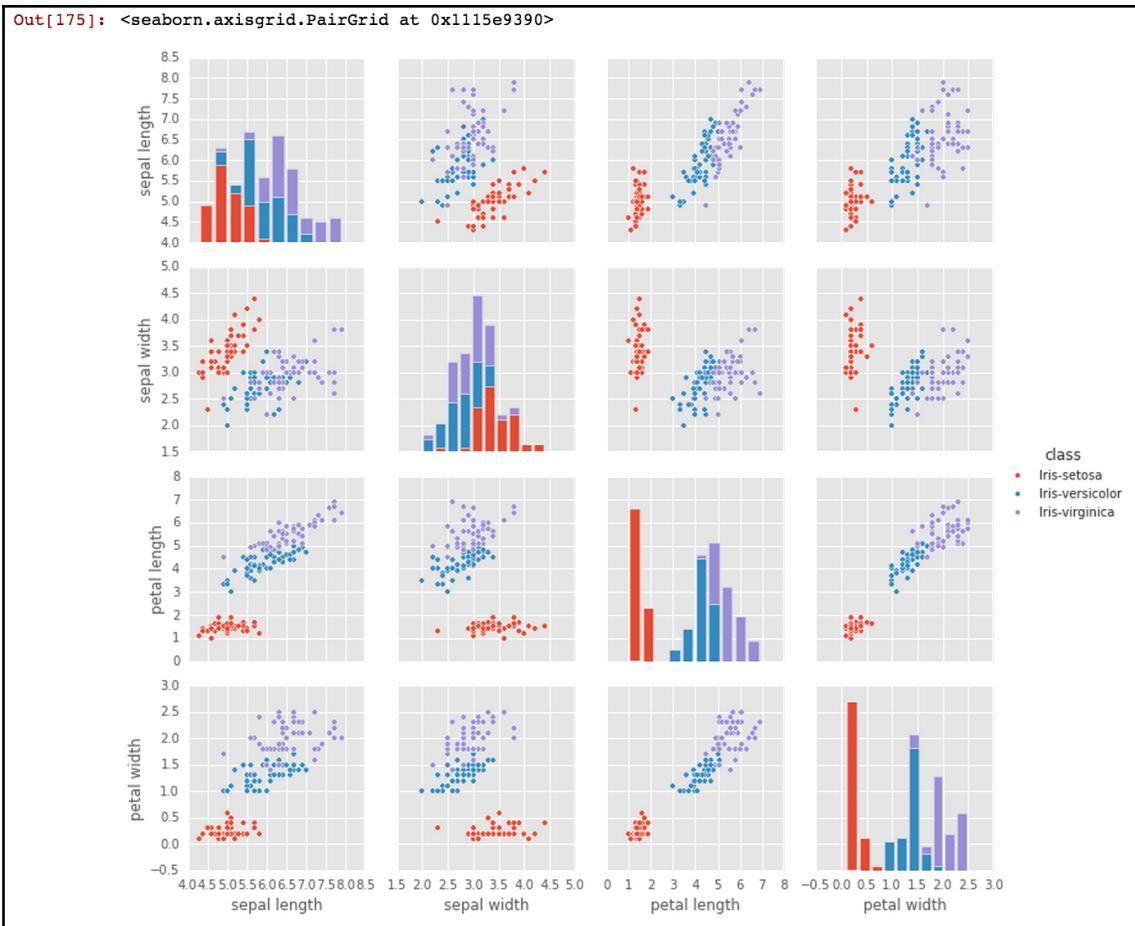
The seaborn library

The next visualization library we'll look at is called `seaborn`, (<http://seaborn.pydata.org/index.html>). It is a library that was created specifically for statistical visualizations. In fact, it is perfect for use with `pandas` DataFrames, where the columns are features and the rows are observations. This style of DataFrame is called `tidy` data, and is the most common form for machine learning applications.

Let's now take a look at the power of `seaborn`:

```
import seaborn as sns
sns.pairplot(df, hue='species')
```

With just those two lines of code, we get the following:



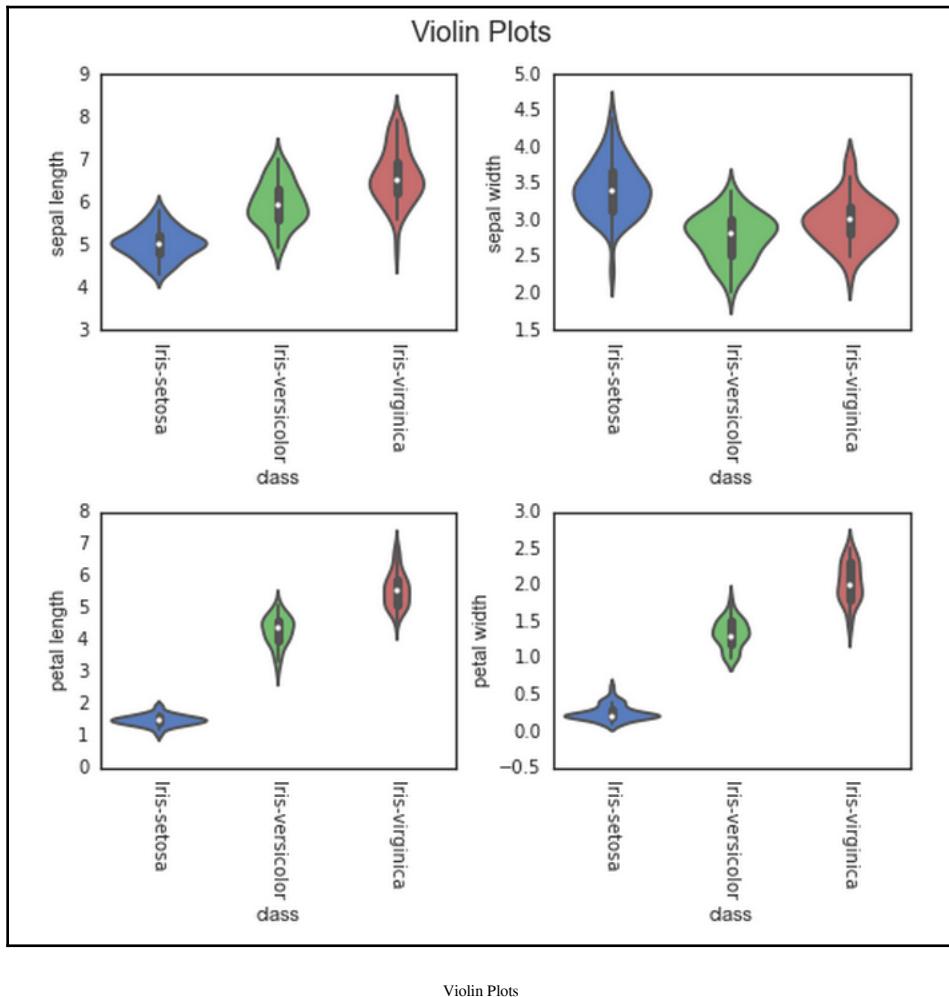
Seaborn plot

Having just detailed the intricate nuances of `matplotlib`, you will immediately appreciate the simplicity with which we generated this plot. All of our features have been plotted against each other and properly labeled with just two lines of code. You might wonder if I just wasted dozens of pages teaching you `matplotlib` when `seaborn` makes these types of visualizations so simple. Well, that isn't the case, as `seaborn` is built on top of `matplotlib`. In fact, you can use all of what you learned about `matplotlib` to modify and work with `seaborn`. Let's take a look at another visualization:

```
fig, ax = plt.subplots(2, 2, figsize=(7, 7))
sns.set(style='white', palette='muted')
sns.violinplot(x=df['species'], y=df['sepal length (cm)'], ax=ax[0,0])
```

```
sns.violinplot(x=df['species'], y=df['sepal width (cm)'], ax=ax[0,1])
sns.violinplot(x=df['species'], y=df['petal length (cm)'], ax=ax[1,0])
sns.violinplot(x=df['species'], y=df['petal width (cm)'], ax=ax[1,1])
fig.suptitle('Violin Plots', fontsize=16, y=1.03)
for i in ax.flat:
    plt.setp(i.get_xticklabels(), rotation=-90)
fig.tight_layout()
```

The preceding code generates the following output:



Violin Plots

Here, we have generated a violin plot for each of the four features. A violin plot displays the distribution of the features. For example, you can easily see that the **petal length** of *setosa* (0) is highly clustered between 1 cm and 2 cm, while *virginica* (2) is much more dispersed, from nearly 4 cm to over 7 cm. You will also notice that we have used much of the same code we used when constructing the `matplotlib` graphs. The main difference is the addition of the `sns.plot()` calls, in place of the `ax.plot()` calls previously. We have also added a title above all of the subplots, rather than over each individually, with the `fig.suptitle()` function. One other notable addition is the iteration over each of the subplots to change the rotation of the `xticklabels`. We call `ax.flat()` and then iterate over each subplot axis to set a particular property using `.setp()`. This prevents us from having to individually type out `ax[0][0]...ax[1][1]` and set the properties, as we did previously in the earlier `matplotlib` subplot code.

There are hundreds of styles of graphs you can generate using `matplotlib` and `seaborn`, and I highly recommend digging into the documentation for these two libraries—it will be time well spent—but the graphs I have detailed in the preceding section should go a long way toward helping you to understand the dataset you have, which in turn will help you when building your machine learning models.

Preparation

We've learned a great deal about inspecting the data we have, but now let's move on to learning how to process and manipulate our data. Here, we will learn about the `.map()`, `.apply()`, `.applymap()`, and `.groupby()` functions of `pandas`. These are invaluable for working with data, and are especially useful in the context of machine learning for feature engineering, a concept we will discuss in detail in later chapters.

map

We'll now begin with the `map` function. The `map` function works on series, so in our case we will use it to transform a column of our DataFrame, which you will recall is just a pandas series. Suppose we decide that the species numbers are not suitable for our needs. We'll use the `map` function with a Python dictionary as the argument to accomplish this. We'll pass in a replacement for each of the unique `iris` types:

```
In [30]: df['species'] = df['species'].map({0: 'SET', 1: 'VER', 2: 'VIR'})  
  
In [31]: df  
Out[31]:
```

	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)	species
0	5.1	3.5	1.4	0.2	SET
1	4.9	3.0	1.4	0.2	SET
2	4.7	3.2	1.3	0.2	SET
3	4.6	3.1	1.5	0.2	SET
4	5.0	3.6	1.4	0.2	SET
5	5.4	3.9	1.7	0.4	SET
6	4.6	3.4	1.4	0.3	SET
7	5.0	3.4	1.5	0.2	SET
8	4.4	2.9	1.4	0.2	SET
9	4.9	3.1	1.5	0.1	SET
10	5.4	3.7	1.5	0.2	SET
..

Let's look at what we have done here. We have run the `map` function over each of the values of the existing `species` column. As each value was found in the Python dictionary, it was added to the return series. We assigned this return series to the same `species` name, so it replaced our original `species` column. Had we chosen a different name, say `short_code`, that column would have been appended to the DataFrame, and we would then have the original `species` column plus the new `short_code` column.

We could have instead passed the `map` function a series or a function to perform this transformation on a column, but this is a functionality that is also available through the `apply` function, which we'll take a look at next. The dictionary functionality is unique to the `map` function, and the most common reason to choose `map` over `apply` for a single column transformation. But, let's now take a look at the `apply` function.

apply

The `apply` function allows us to work with both `DataFrames` and `series`. We'll start with an example that would work equally well with `map`, before moving on to examples that would only work with `apply`.

Using our `iris` `DataFrame`, let's make a new column based on petal width. We previously saw that the mean for the petal width was `1.3`. Let's now create a new column in our `DataFrame`, `wide_petal`, that contains binary values based on the value in the petal width column. If the petal width is equal to or wider than the median, we will code it with a `1`, and if it is less than the median, we will code it `0`. We'll do this using the `apply` function on the petal width column:

```
In [32]: df['wide petal'] = df['petal width (cm)'].apply(lambda v: 1 if v >= 1.3 else 0)

In [33]: df

Out[33]:
   sepal length (cm)  sepal width (cm)  petal length (cm)  petal width (cm)  species  wide petal
0                 5.1             3.5              1.4            0.2      SET          0
1                 4.9             3.0              1.4            0.2      SET          0
2                 4.7             3.2              1.3            0.2      SET          0
3                 4.6             3.1              1.5            0.2      SET          0
4                 5.0             3.6              1.4            0.2      SET          0
5                 5.4             3.9              1.7            0.4      SET          0
6                 4.6             3.4              1.4            0.3      SET          0
7                 5.0             3.4              1.5            0.2      SET          0
8                 4.4             2.9              1.4            0.2      SET          0
```

A few things happened here, so let's walk through them step by step. The first is that we were able to append a new column to the `DataFrame` simply by using the column selection syntax for a column name, which we want to create, in this case `wide_petal`. We set that new column equal to the output of the `apply` function. Here, we ran `apply` on the `petal width` column that returned the corresponding values in the `wide_petal` column. The `apply` function works by running through each value of the `petal width` column. If the value is greater than or equal to `1.3`, the function returns `1`, otherwise it returns `0`. This type of transformation is a fairly common feature engineering transformation in machine learning, so it is good to be familiar with how to perform it.

Let's now take a look at using `apply` on a DataFrame rather than a single series. We'll now create a feature based on the petal area:

In [34]:	<code>df['petal area'] = df.apply(lambda r: r['petal width (cm)'] * r['petal length (cm)'], axis=1)</code>																																																																								
In [35]:	<code>df</code>																																																																								
Out[35]:	<table><thead><tr><th></th><th>sepal length (cm)</th><th>sepal width (cm)</th><th>petal length (cm)</th><th>petal width (cm)</th><th>species</th><th>wide petal</th><th>petal area</th></tr></thead><tbody><tr><td>0</td><td>5.1</td><td>3.5</td><td>1.4</td><td>0.2</td><td>SET</td><td>0</td><td>0.28</td></tr><tr><td>1</td><td>4.9</td><td>3.0</td><td>1.4</td><td>0.2</td><td>SET</td><td>0</td><td>0.28</td></tr><tr><td>2</td><td>4.7</td><td>3.2</td><td>1.3</td><td>0.2</td><td>SET</td><td>0</td><td>0.26</td></tr><tr><td>3</td><td>4.6</td><td>3.1</td><td>1.5</td><td>0.2</td><td>SET</td><td>0</td><td>0.30</td></tr><tr><td>4</td><td>5.0</td><td>3.6</td><td>1.4</td><td>0.2</td><td>SET</td><td>0</td><td>0.28</td></tr><tr><td>5</td><td>5.4</td><td>3.9</td><td>1.7</td><td>0.4</td><td>SET</td><td>0</td><td>0.68</td></tr><tr><td>6</td><td>4.6</td><td>3.4</td><td>1.4</td><td>0.3</td><td>SET</td><td>0</td><td>0.42</td></tr><tr><td>7</td><td>5.0</td><td>3.4</td><td>1.5</td><td>0.2</td><td>SET</td><td>0</td><td>0.30</td></tr></tbody></table>		sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)	species	wide petal	petal area	0	5.1	3.5	1.4	0.2	SET	0	0.28	1	4.9	3.0	1.4	0.2	SET	0	0.28	2	4.7	3.2	1.3	0.2	SET	0	0.26	3	4.6	3.1	1.5	0.2	SET	0	0.30	4	5.0	3.6	1.4	0.2	SET	0	0.28	5	5.4	3.9	1.7	0.4	SET	0	0.68	6	4.6	3.4	1.4	0.3	SET	0	0.42	7	5.0	3.4	1.5	0.2	SET	0	0.30
	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)	species	wide petal	petal area																																																																		
0	5.1	3.5	1.4	0.2	SET	0	0.28																																																																		
1	4.9	3.0	1.4	0.2	SET	0	0.28																																																																		
2	4.7	3.2	1.3	0.2	SET	0	0.26																																																																		
3	4.6	3.1	1.5	0.2	SET	0	0.30																																																																		
4	5.0	3.6	1.4	0.2	SET	0	0.28																																																																		
5	5.4	3.9	1.7	0.4	SET	0	0.68																																																																		
6	4.6	3.4	1.4	0.3	SET	0	0.42																																																																		
7	5.0	3.4	1.5	0.2	SET	0	0.30																																																																		

Creating a new feature

Notice that we called `apply` not on a series here, but on the entire DataFrame, and because `apply` was called on the entire DataFrame, we passed in `axis=1` in order to tell pandas that we want to apply the function row-wise. If we passed in `axis=0`, then the function would operate column-wise. Here, each column is processed sequentially, and we choose to multiply the values from the petal length (cm) and petal width (cm) columns. The resultant series then becomes the petal area column in our DataFrame. This type of power and flexibility is what makes pandas an indispensable tool for data manipulation.

applymap

We've looked at manipulating columns and explained how to work with rows, but suppose you'd like to perform a function across all data cells in your DataFrame. This is where `applymap` is the correct tool. Let's take a look at an example:

In [36]:	df.applymap(lambda x: np.log(x) if isinstance(x, float) else x)																																																																								
Out[36]:	<table border="1"> <thead> <tr> <th></th><th>sepal length (cm)</th><th>sepal width (cm)</th><th>petal length (cm)</th><th>petal width (cm)</th><th>species</th><th>wide petal</th><th>petal area</th></tr> </thead> <tbody> <tr><td>0</td><td>1.629241</td><td>1.252763</td><td>0.336472</td><td>-1.609438</td><td>SET</td><td>0</td><td>-1.272966</td></tr> <tr><td>1</td><td>1.589235</td><td>1.098612</td><td>0.336472</td><td>-1.609438</td><td>SET</td><td>0</td><td>-1.272966</td></tr> <tr><td>2</td><td>1.547563</td><td>1.163151</td><td>0.262364</td><td>-1.609438</td><td>SET</td><td>0</td><td>-1.347074</td></tr> <tr><td>3</td><td>1.526056</td><td>1.131402</td><td>0.405465</td><td>-1.609438</td><td>SET</td><td>0</td><td>-1.203973</td></tr> <tr><td>4</td><td>1.609438</td><td>1.280934</td><td>0.336472</td><td>-1.609438</td><td>SET</td><td>0</td><td>-1.272966</td></tr> <tr><td>5</td><td>1.686399</td><td>1.360977</td><td>0.530628</td><td>-0.916291</td><td>SET</td><td>0</td><td>-0.385662</td></tr> <tr><td>6</td><td>1.526056</td><td>1.223775</td><td>0.336472</td><td>-1.203973</td><td>SET</td><td>0</td><td>-0.867501</td></tr> <tr><td>7</td><td>1.609438</td><td>1.223775</td><td>0.405465</td><td>-1.609438</td><td>SET</td><td>0</td><td>-1.203973</td></tr> </tbody> </table>		sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)	species	wide petal	petal area	0	1.629241	1.252763	0.336472	-1.609438	SET	0	-1.272966	1	1.589235	1.098612	0.336472	-1.609438	SET	0	-1.272966	2	1.547563	1.163151	0.262364	-1.609438	SET	0	-1.347074	3	1.526056	1.131402	0.405465	-1.609438	SET	0	-1.203973	4	1.609438	1.280934	0.336472	-1.609438	SET	0	-1.272966	5	1.686399	1.360977	0.530628	-0.916291	SET	0	-0.385662	6	1.526056	1.223775	0.336472	-1.203973	SET	0	-0.867501	7	1.609438	1.223775	0.405465	-1.609438	SET	0	-1.203973
	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)	species	wide petal	petal area																																																																		
0	1.629241	1.252763	0.336472	-1.609438	SET	0	-1.272966																																																																		
1	1.589235	1.098612	0.336472	-1.609438	SET	0	-1.272966																																																																		
2	1.547563	1.163151	0.262364	-1.609438	SET	0	-1.347074																																																																		
3	1.526056	1.131402	0.405465	-1.609438	SET	0	-1.203973																																																																		
4	1.609438	1.280934	0.336472	-1.609438	SET	0	-1.272966																																																																		
5	1.686399	1.360977	0.530628	-0.916291	SET	0	-0.385662																																																																		
6	1.526056	1.223775	0.336472	-1.203973	SET	0	-0.867501																																																																		
7	1.609438	1.223775	0.405465	-1.609438	SET	0	-1.203973																																																																		

Using applymap function

Here, we called `applymap` on our DataFrame in order to get the log of every value (`np.log()` utilizes the NumPy library to return this value), if that value is of the float type. This type checking prevents returning an error or a float for the `species` or `wide petal` columns, which are string and integer values respectively. Common uses of `applymap` include transforming or formatting each cell based on meeting a number of conditional criteria.

groupby

Let's now look at an operation that is highly useful, but often difficult for new pandas users to get their heads around: the `.groupby()` function. We'll walk through a number of examples step by step in order to illustrate the most important functionality.

The `groupby` operation does exactly what it says: it groups data based on some class or classes you choose. Let's take a look at a simple example using our `iris` dataset. We'll go back and reimport our original `iris` dataset, and run our first `groupby` operation:

In [37]:	df.groupby('species').mean()																												
Out[37]:	<table border="1"> <thead> <tr> <th>species</th><th>sepal length (cm)</th><th>sepal width (cm)</th><th>petal length (cm)</th><th>petal width (cm)</th><th>wide petal</th><th>petal area</th></tr> </thead> <tbody> <tr><td>SET</td><td>5.006</td><td>3.418</td><td>1.464</td><td>0.244</td><td>0.0</td><td>0.3628</td></tr> <tr><td>VER</td><td>5.936</td><td>2.770</td><td>4.260</td><td>1.326</td><td>0.7</td><td>5.7204</td></tr> <tr><td>VIR</td><td>6.588</td><td>2.974</td><td>5.552</td><td>2.026</td><td>1.0</td><td>11.2962</td></tr> </tbody> </table>	species	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)	wide petal	petal area	SET	5.006	3.418	1.464	0.244	0.0	0.3628	VER	5.936	2.770	4.260	1.326	0.7	5.7204	VIR	6.588	2.974	5.552	2.026	1.0	11.2962
species	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)	wide petal	petal area																							
SET	5.006	3.418	1.464	0.244	0.0	0.3628																							
VER	5.936	2.770	4.260	1.326	0.7	5.7204																							
VIR	6.588	2.974	5.552	2.026	1.0	11.2962																							

Here, data for each species is partitioned and the mean for each feature is provided. Let's take it a step further now and get full descriptive statistics for each species:

In [38]:	df.groupby('species').describe()																																																																																																															
Out[38]:	<table border="1"> <thead> <tr> <th rowspan="2">species</th><th colspan="5">petal area</th><th colspan="5">petal length (cm)</th><th colspan="5">sepal width (cm)</th><th colspan="3">wide petal</th></tr> <tr> <th>count</th><th>mean</th><th>std</th><th>min</th><th>25%</th><th>50%</th><th>75%</th><th>max</th><th>count</th><th>mean</th><th>...</th><th>75%</th><th>max</th><th>count</th><th>mean</th><th>std</th><th></th><th></th></tr> </thead> <tbody> <tr> <td>SET</td><td>50.0</td><td>0.3628</td><td>0.183248</td><td>0.11</td><td>0.2650</td><td>0.300</td><td>0.42</td><td>0.96</td><td>50.0</td><td>1.464</td><td>...</td><td>3.675</td><td>4.4</td><td>50.0</td><td>0.0</td><td>0.00000</td><td></td><td></td></tr> <tr> <td>VER</td><td>50.0</td><td>5.7204</td><td>1.368403</td><td>3.30</td><td>4.8600</td><td>5.615</td><td>6.75</td><td>8.64</td><td>50.0</td><td>4.260</td><td>...</td><td>3.000</td><td>3.4</td><td>50.0</td><td>0.7</td><td>0.46291</td><td></td><td></td></tr> <tr> <td>VIR</td><td>50.0</td><td>11.2962</td><td>2.157412</td><td>7.50</td><td>9.7175</td><td>11.445</td><td>12.79</td><td>15.87</td><td>50.0</td><td>5.552</td><td>...</td><td>3.175</td><td>3.8</td><td>50.0</td><td>1.0</td><td>0.00000</td><td></td><td></td></tr> </tbody> </table>																		species	petal area					petal length (cm)					sepal width (cm)					wide petal			count	mean	std	min	25%	50%	75%	max	count	mean	...	75%	max	count	mean	std			SET	50.0	0.3628	0.183248	0.11	0.2650	0.300	0.42	0.96	50.0	1.464	...	3.675	4.4	50.0	0.0	0.00000			VER	50.0	5.7204	1.368403	3.30	4.8600	5.615	6.75	8.64	50.0	4.260	...	3.000	3.4	50.0	0.7	0.46291			VIR	50.0	11.2962	2.157412	7.50	9.7175	11.445	12.79	15.87	50.0	5.552	...	3.175	3.8	50.0	1.0	0.00000		
species	petal area					petal length (cm)					sepal width (cm)					wide petal																																																																																																
	count	mean	std	min	25%	50%	75%	max	count	mean	...	75%	max	count	mean	std																																																																																																
SET	50.0	0.3628	0.183248	0.11	0.2650	0.300	0.42	0.96	50.0	1.464	...	3.675	4.4	50.0	0.0	0.00000																																																																																																
VER	50.0	5.7204	1.368403	3.30	4.8600	5.615	6.75	8.64	50.0	4.260	...	3.000	3.4	50.0	0.7	0.46291																																																																																																
VIR	50.0	11.2962	2.157412	7.50	9.7175	11.445	12.79	15.87	50.0	5.552	...	3.175	3.8	50.0	1.0	0.00000																																																																																																
SET	50.0	0.3628	0.183248	0.11	0.2650	0.300	0.42	0.96	50.0	1.464	...	3.675	4.4	50.0	0.0	0.00000																																																																																																
VER	50.0	5.7204	1.368403	3.30	4.8600	5.615	6.75	8.64	50.0	4.260	...	3.000	3.4	50.0	0.7	0.46291																																																																																																
VIR	50.0	11.2962	2.157412	7.50	9.7175	11.445	12.79	15.87	50.0	5.552	...	3.175	3.8	50.0	1.0	0.00000																																																																																																

Statistics for each species

And now, we can see the full breakdown bucketed by species. Let's now look at some other groupby operations we can perform. We saw previously that petal length and width had some relatively clear boundaries between species. Now, let's examine how we might use groupby to see that:

species	
petal width (cm)	
0.1	[SET]
0.2	[SET]
0.3	[SET]
0.4	[SET]
0.5	[SET]
0.6	[SET]
1.0	[VER]
1.1	[VER]
1.2	[VER]
1.3	[VER]
1.4	[VER, VIR]
1.5	[VER, VIR]

In this case, we have grouped each unique species by the petal width they were associated with. This is a manageable number of measurements to group by, but if it were to become much larger, we would likely need to partition the measurements into brackets. As we saw previously, that can be accomplished by means of the `apply` function.

Let's now take a look at a custom aggregation function:

```
In [51]: df.groupby('species')[ 'petal width (cm)' ].agg(lambda x: x.max() - x.min())
Out[51]: species
          SET    0.5
          VER    0.8
          VIR    1.1
Name: petal width (cm), dtype: float64
```

In this code, we grouped petal width by species using the `.max()` and `.min()` functions, and a `lambda` function that returns a maximum petal width less than the minimum petal width.



We've only just touched on the functionality of the `groupby` function; there is a lot more to learn, so I encourage you to read the documentation available at <http://pandas.pydata.org/pandas-docs/stable/>.

Hopefully, you now have a solid base-level understanding of how to manipulate and prepare data in preparation for our next step, which is modeling. We will now move on to discuss the primary libraries in the Python machine learning ecosystem.

Modeling and evaluation

In this section, we will go through different libraries such as `statsmodels` and `Scikit-learn` and also understand what is deployment.

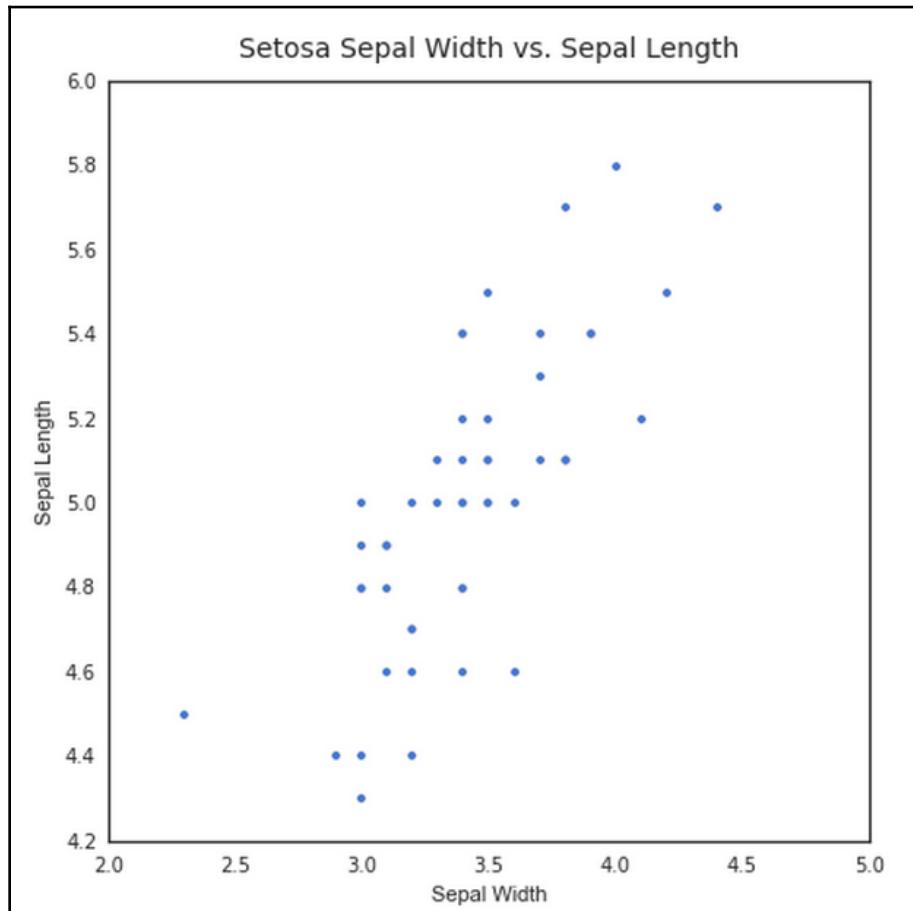
Statsmodels

The first library we'll cover is the `statsmodels` library (<http://statsmodels.sourceforge.net/>). `Statsmodels` is a Python package that is well documented and developed for exploring data, estimating models, and running statistical tests. Let's use it here to build a simple linear regression model of the relationship between sepal length and sepal width for the `setosa` species.

First, let's visually inspect the relationship with a scatterplot:

```
fig, ax = plt.subplots(figsize=(7, 7))
ax.scatter(df['sepal width (cm)'][:50], df['sepal length (cm)'][:50])
ax.set_ylabel('Sepal Length')
ax.set_xlabel('Sepal Width')
ax.set_title('Setosa Sepal Width vs. Sepal Length', fontsize=14, y=1.02)
```

The preceding code generates the following output:



So, we can see that there appears to be a positive linear relationship; that is, as the sepal width increases, the sepal length does as well. We'll next run a linear regression on the data using `statsmodels` to estimate the strength of that relationship:

```
import statsmodels.api as sm

y = df['sepal length'][:50]
x = df['sepal width'][:50]
X = sm.add_constant(x)

results = sm.OLS(y, X).fit()
print results.summary()
```

The preceding code generates the following output:

OLS Regression Results						
Dep. Variable:	sepal length	R-squared:	0.558			
Model:	OLS	Adj. R-squared:	0.548			
Method:	Least Squares	F-statistic:	60.52			
Date:	Sun, 11 Oct 2015	Prob (F-statistic):	4.75e-10			
Time:	18:14:39	Log-Likelihood:	2.0879			
No. Observations:	50	AIC:	-0.1759			
Df Residuals:	48	BIC:	3.648			
Df Model:	1					
	coef	std err	t	P> t	[95.0% Conf. Int.]	
const	2.6447	0.305	8.660	0.000	2.031	3.259
sepal width	0.6909	0.089	7.779	0.000	0.512	0.869
Omnibus:	0.252	Durbin-Watson:	2.517			
Prob(Omnibus):	0.882	Jarque-Bera (JB):	0.436			
Skew:	-0.110	Prob(JB):	0.804			
Kurtosis:	2.599	Cond. No.	34.0			

In the preceding diagram, we have the results of our simple regression model. Since this is a linear regression, the model takes the format of $Y = B_0 + B_1X$, where B_0 is the intercept and B_1 is the regression coefficient. Here, the formula would be $\text{Sepal Length} = 2.6447 + 0.6909 * \text{Sepal Width}$. We can also see that the R^2 for the model is a respectable 0.558, and the p -value, (Prob), is highly significant—at least for this species.

Let's now use the `results` object to plot our regression line:

```
fig, ax = plt.subplots(figsize=(7,7))
ax.plot(x, results.fittedvalues, label='regression line')
ax.scatter(x, y, label='data point', color='r')
ax.set_ylabel('Sepal Length')
ax.set_xlabel('Sepal Width')
ax.set_title('Setosa Sepal Width vs. Sepal Length', fontsize=14, y=1.02)
ax.legend(loc=2)
```

The preceding code generates the following output:



By plotting `results.fittedvalues`, we can get the resulting regression line from our regression.

There are a number of other statistical functions and tests in the `statsmodels` package, and I invite you to explore them. It is an exceptionally useful package for standard statistical modeling in Python. Let's now move on to the king of Python machine learning packages: scikit-learn.

Scikit-learn

Scikit-learn is an amazing Python library with unrivaled documentation, designed to provide a consistent API to dozens of algorithms. It is built upon, and is itself, a core component of the Python scientific stack, which includes NumPy, SciPy, pandas, and matplotlib. Here are some of the areas scikit-learn covers: classification, regression, clustering, dimensionality reduction, model selection, and preprocessing.

We'll look at a few examples. First, we will build a classifier using our `iris` data, and then we'll look at how we can evaluate our model using the tools of scikit-learn:

1. The first step to building a machine learning model in scikit-learn is understanding how the data must be structured.
2. The independent variables should be a numeric $n \times m$ matrix, X , and the dependent variable, y , an $n \times 1$ vector.
3. The y vector may be either a numeric continuous or categorical, or a string categorical.
4. These are then passed into the `.fit()` method on the chosen classifier.
5. This is the great benefit of using scikit-learn: each classifier utilizes the same methods to the extent possible. This makes swapping them in and out a breeze.

Let's see this in action in our first example:

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.cross_validation import train_test_split

clf = RandomForestClassifier(max_depth=5, n_estimators=10)

X = df.ix[:, :4]
y = df.ix[:, 4]

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=.3)

clf.fit(X_train, y_train)

y_pred = clf.predict(X_test)

rf = pd.DataFrame(zip(y_pred, y_test), columns=['predicted', 'actual'])
```

```
rf['correct'] = rf.apply(lambda r: 1 if r['predicted'] == r['actual'] else  
0, axis=1)  
rf
```

The preceding code generates the following output:

	predicted	actual	correct
0	VIR	VIR	1
1	SET	SET	1
2	VER	VER	1
3	SET	SET	1
4	VER	VER	1
5	VIR	VIR	1
6	VIR	VIR	1
7	VIR	VIR	1
8	VER	VER	1
9	VER	VER	1
10	VIR	VIR	1
11	VIR	VIR	1
12	SET	SET	1
13	VIR	VIR	1
14	SET	SET	1
15	VER	VER	1
16	SET	SET	1

Now, let's execute the following line of code:

```
rf['correct'].sum()/rf['correct'].count()
```

The preceding code generates the following output:

0.9555555555555556

In the preceding few lines of code, we built, trained, and tested a classifier that has a 95% accuracy level on our `iris` dataset. Let's unpack each of the steps. Up at the top, we made a couple of imports; the first two are from scikit-learn, which thankfully is shortened to `sklearn` in import statements. The first import is a random forest classifier, and the second is a module for splitting your data into training and testing cohorts. This data partitioning is critical in building machine learning applications for a number of reasons. We'll get into this in later chapters, but suffice to say at this point it is a must. This `train_test_split` module also shuffles your data, which again is important as the order can contain information that would bias your actual predictions.

The first curious-looking line after the imports instantiates our classifier, in this case a random forest classifier. We select a forest that uses 10 decision trees, and each tree is allowed a maximum split depth of five. This is put in place to avoid overfitting, something we will discuss in depth in later chapters.

The next two lines create our `X` matrix and `y` vector. If you remember our original `iris` DataFrame, it contained four features: petal width and length, and sepal width and length. These features are selected and become our independent feature matrix, `X`. The last column, the `iris` class names, then becomes our dependent `y` vector.

These are then passed into the `train_test_split` method, which shuffles and partitions our data into four subsets, `X_train`, `X_test`, `y_train`, and `y_test`. The `test_size` parameter is set to `.3`, which means 30% of our dataset will be allocated to the `X_test` and `y_test` partitions, while the rest will be allocated to the training partitions, `X_train` and `y_train`.

Next, our model is fitted using the training data. Having trained the model, we then call the `predict` method on our classifier using our test data. Remember, the test data is data the classifier has not seen. The return of this prediction is a list of prediction labels. We then create a DataFrame of the actual labels versus the predicted labels. We finally total the correct predictions and divide by the total number of instances, which we can see gave us a very accurate prediction. Let's now see which features gave us the most discriminative or predictive power:

```
f_importances = clf.feature_importances_
f_names = df.columns[:4]
f_std = np.std([tree.feature_importances_ for tree in clf.estimators_],
axis=0)

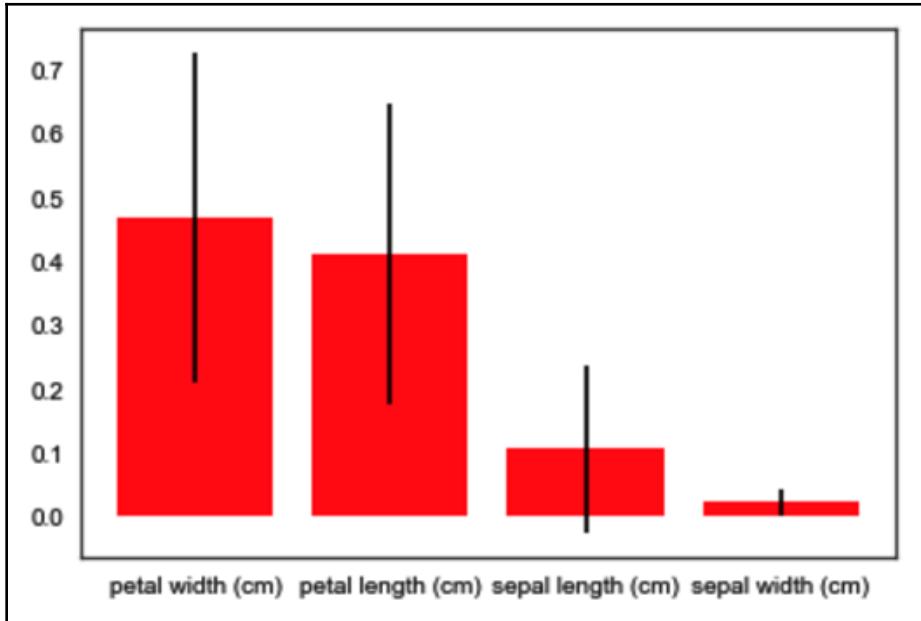
zz = zip(f_importances, f_names, f_std)
zzs = sorted(zz, key=lambda x: x[0], reverse=True)

imps = [x[0] for x in zzs]
```

```
labels = [x[1] for x in zzs]
errs = [x[2] for x in zzs]

plt.bar(range(len(f_importances)), imps, color="r", yerr=errs,
align="center")
plt.xticks(range(len(f_importances)), labels);
```

The preceding code generates the following output:



As we expected, based upon our earlier visual analysis, the petal length and width have more discriminative power when differentiating between the `iris` classes. Where exactly did these numbers come from though? The random forest has a method called `.feature_importances_` that returns the relative performance of the feature for splitting at the leaves. If a feature is able to consistently and cleanly split a group into distinct classes, it will have a high feature importance. This number will always total one. As you will notice here, we have included the standard deviation, which helps to illustrate how consistent each feature is. This is generated by taking the feature importance, for each of the features, for each ten trees, and calculating the standard deviation.

Let's now take a look at one more example using scikit-learn. We will now switch out our classifier and use a **support vector machine (SVM)**:

```
from sklearn.multiclass import OneVsRestClassifier
from sklearn.svm import SVC
from sklearn.cross_validation import train_test_split

clf = OneVsRestClassifier(SVC(kernel='linear'))

X = df.ix[:, :4]
y = np.array(df.ix[:, 4]).astype(str)

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=.3)

clf.fit(X_train, y_train)

y_pred = clf.predict(X_test)

rf = pd.DataFrame(zip(y_pred, y_test), columns=['predicted', 'actual'])
rf['correct'] = rf.apply(lambda r: 1 if r['predicted'] == r['actual'] else 0, axis=1)
rf
```

The preceding code generates the following output:

	predicted	actual	correct
0	SET	SET	1
1	VIR	VIR	1
2	VER	VER	1
3	VIR	VIR	1
4	SET	SET	1
5	VIR	VER	0
6	VIR	VIR	1
7	VER	VER	1
8	VIR	VIR	1
9	VER	VER	1
10	VIR	VER	0
11	VIR	VIR	1
12	VIR	VIR	1

Now, let's execute the following line of code:

```
rf['correct'].sum() / rf['correct'].count()
```

The preceding code generates the following output:

```
0.9777777777777775
```

Here, we have swapped in an SVM without changing virtually any of our code. The only changes were the ones related to the importing of the SVM instead of the random forest, and the line that instantiates the classifier. (I did have to make one small change to the format of the `y` labels, as the SVM wasn't able to interpret them as NumPy strings like the random forest classifier was. Sometimes, these data type conversions have to be made specific or it will result in an error, but it's a minor annoyance.)

This is only a small sample of the functionality of scikit-learn, but it should give you a hint of the power of this magnificent tool for machine learning applications. There are a number of additional machine learning libraries we won't have a chance to discuss here but will explore in later chapters, but I strongly suggest that if this is your first time utilizing a machine learning library, and you want a strong general-purpose tool, scikit-learn is your go-to choice.

Deployment

There are a number of options you can choose from when you decide to put your machine learning model into production. It depends substantially on the nature of the application. Deployment could include anything from a cron job run on your local machine to a full-scale implementation deployed on an Amazon EC2 instance.

We won't go into detail regarding specific implementations here, but we will have a chance to delve into different deployment examples throughout the book.

Setting up your machine learning environment

We've covered a number of libraries, and it could be somewhat of a chore to install if you were to do each individually—which you certainly can, since most can be installed with pip, Python's package manager, but I would strongly urge you to go with a prepacked solution such as the Anaconda Python distribution (<http://anaconda.org>). This allows you to download and install a single executable with all the packages and dependencies handled for you. And since the distribution is targeted to Python scientific stack users, it is essentially a one-and-done solution.

Anaconda also includes a package manager that makes updating your packages a simple task. Simply type `conda update <package_name>`, and you will be updated to the most recent stable release.

Summary

In this chapter, we learned about the data science/machine learning workflow. We learned how to take our data step by step through each stage of the pipeline, going from acquisition all the way through to deployment. We also learned key features of each of the most important libraries in the Python scientific stack. We will now take this knowledge and these lessons and begin to apply them to create unique and useful machine learning applications. Let's get started!

2

Build an App to Find Underpriced Apartments

In Chapter 1, *The Python Machine Learning Ecosystem*, we learned the essentials for working with data. We'll now apply that knowledge to build out our first machine learning application. We'll begin with a minimal, but highly-practical example: building an application to identify underpriced apartments.

If you've ever searched for an apartment, you will appreciate just how frustrating the process can be. Not only is it time-consuming, but even when you do find an apartment you like, how do you know whether it's the right one?

Most likely, you have a target budget and a target location. But, if you are anything like me, you are also willing to make a few trade-offs. For example, I live in New York City, and being near an amenity like the subway is a big plus. But how much is that worth? Should I trade being in a building with an elevator for being closer to the train? How many minutes of walking to the train is worth walking up a flight of stairs? When renting, there are dozens of questions like this to consider. So how can we use machine learning to help us make these types of decisions?

We'll spend the remainder of this chapter exploring just that. We won't be able to get answers to all the questions we have (for reasons that will become clear later), but by the end of the chapter, we'll have created an application that will make finding the right apartment just a little bit easier.

Here's what we'll cover in this chapter:

- Sourcing apartment listing data
- Inspecting and preparing the data
- Visualizing the data
- Regression modeling
- Forecasting

Sourcing apartment listing data

In the early 1970s, if you wanted to purchase a stock, you would need to engage a broker, who would charge you a fixed commission of nearly 1%. If you wanted to purchase an airline ticket, you would need to contact a travel agent, who would earn a commission of around 7%. And if you wanted to sell a home, you would contact a real estate agent, who would earn a commission of 6%. In 2018, you can do the first two essentially for free. The last one remains as it was in the 1970s.

Why is this the case and, more importantly, what does any of this have to do with machine learning? The reality is, it all comes down to data, and who has access to that data.

You might assume that you could easily access troves of real estate listing data quite easily through APIs or by **web scraping** real estate websites. You would be wrong. Well, wrong if you intend to follow the terms and conditions of those sites. Real estate data is tightly controlled by the **National Association of Realtors (NAR)**, who run the **Multiple Listing Service (MLS)**. This is a service that aggregates listing data, and is only available to brokers and agents at great expense. So, as you can imagine, they aren't too keen on letting just anyone download it *en masse*.

This is unfortunate, since opening up this data would undoubtedly lead to useful consumer applications. This seems especially important for a purchase decision that represents the largest portion of a family's budget.

With that said, not all hope is lost, as not every site explicitly bans scraping.

Pulling down listing data

We'll be using the RentHop site, <http://www.renthop.com>, to source our listing data. The following screenshot of the site shows the layout of the listings we'll be retrieving:

Upcoming Open Houses

Sun, Jul 15	1:00pm - 10:00pm	\$3,000	2BR at 161 west 80th street
Sun, Jul 15	1:00pm - 10:00pm	\$2,750	1BR at 11 west 82nd street
Sun, Jul 15	1:00pm - 10:00pm	\$2,150	1BR at 143 west 80th street

« Back | Page 1 of 3422 (68,428 Rentals) | Next » Sort: Quality | Price

11 West 82nd Street, Apt 12A
Upper West Side, Upper Manhattan, Manhattan
\$2,750 | 1 Bed | 1 Bath
Hillary Pitt, Upper West Side Expert
Open House: Sun, Jul 15, 1:00pm - 10:00pm
[Check Availability](#)
Exclusive | Elevator

78-11 87 Ave, Apt 2
Woodhaven, Southwestern Queens, Queens
\$2,300 | 3 Bed | 1 Bath
By Monika Bochen
[Check Availability](#)
Exclusive | 1,000 ft²

What we can see is that the listings have the address, the price, the number of bedrooms, and the number of bathrooms. We'll start by retrieving this information for each listing.

We are going to be using the Python Requests library for this task. Requests is dubbed *HTTP for humans*, and it makes it super easy to retrieve websites. If you want an overview on how to use Requests, the quick start guide is available at <http://docs.python-requests.org/en/master/user/quickstart/>. Follow these steps:

1. So, the first step is to prepare our Jupyter Notebook with the imports we'll be using for this task. We do that in the following code snippet:

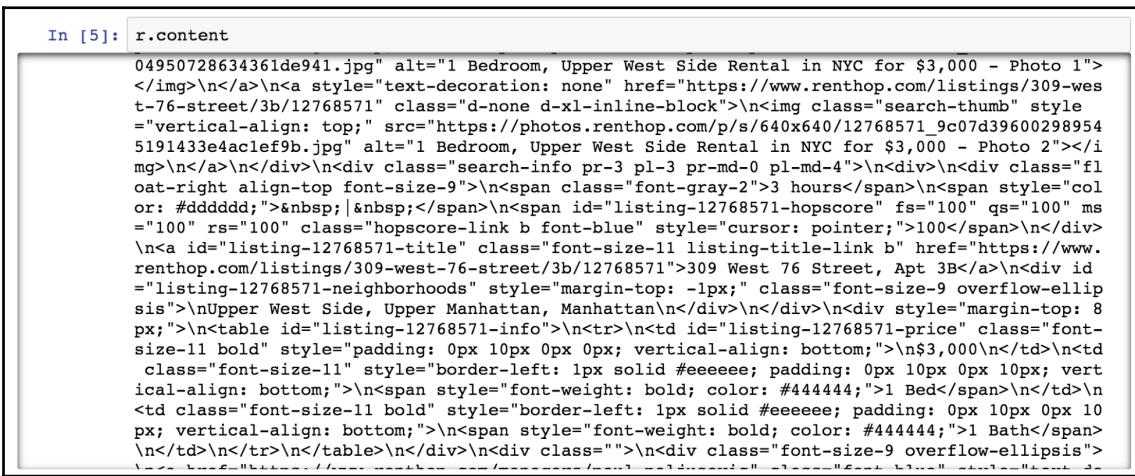
```
import numpy as np
import pandas as pd
import requests
import matplotlib.pyplot as plt
%matplotlib inline
```

We'll likely need to import more libraries later on, but for now this should get us started.

2. We are going to use NYC apartment data in our model. The URL for that data is <https://www.renthop.com/nyc/apartments-for-rent>. Let's run a quick test and make sure we can retrieve that page. We do that in the following code:

```
r = requests.get('https://www.renthop.com/nyc/apartments-for-rent')
r.content
```

3. This code makes a call to the site, and retrieves the information, storing it in the `r` object. There are a number of attributes we could retrieve from that `r` object, but for now, we just want the page content. We can see the output of that in the following screenshot:



The screenshot shows a Jupyter Notebook cell with the input code `In [5]: r.content` and the resulting output, which is a large block of raw HTML. The HTML contains details about an apartment listing, including an image URL, a title, address, price, and other descriptive text.

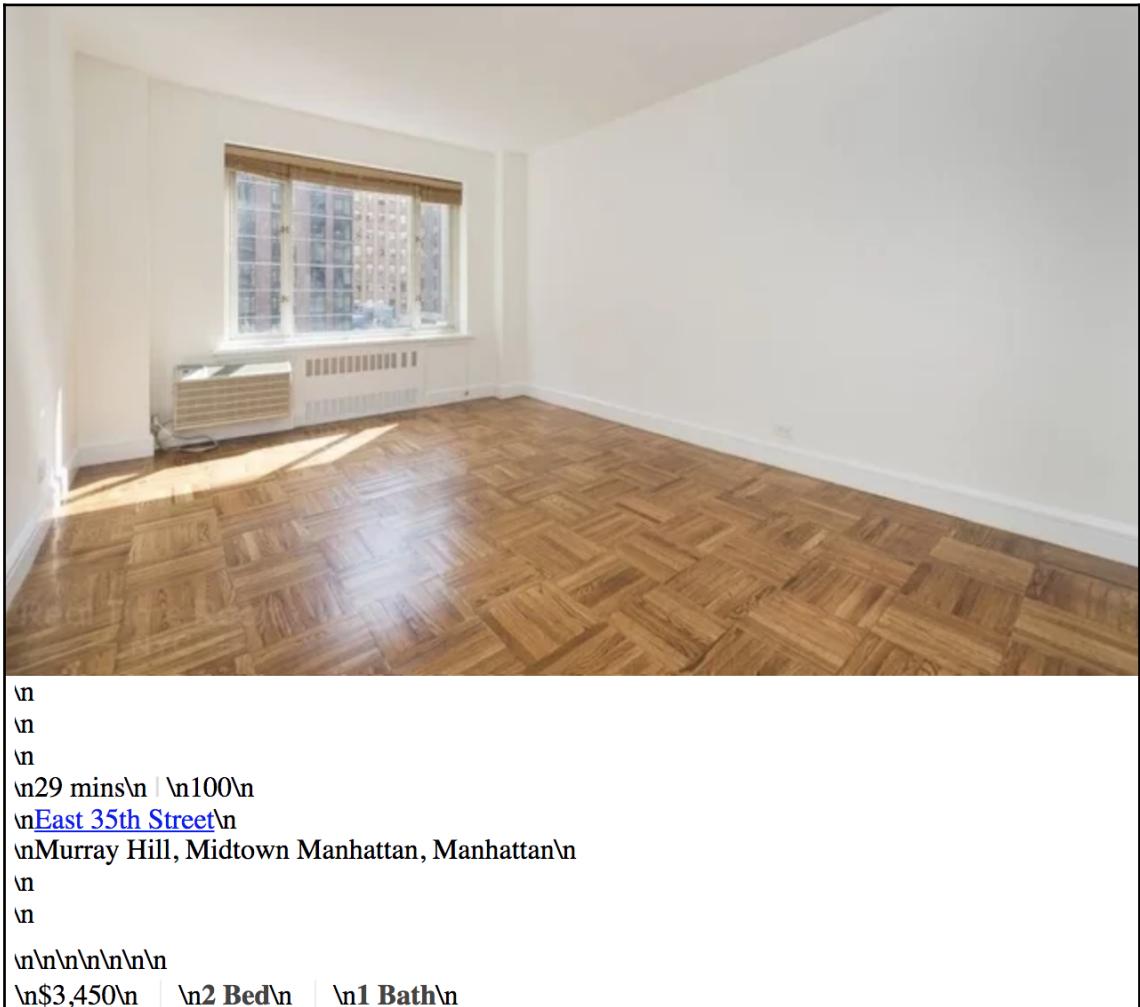
```
04950728634361de941.jpg" alt="1 Bedroom, Upper West Side Rental in NYC for $3,000 - Photo 1">
</img></a><a style="text-decoration: none" href="https://www.renthop.com/listings/309-west-street/3b/12768571" class="d-none d-xl-inline-block"></img></a></div><div class="search-info pr-3 pl-3 pr-md-0 pl-md-4"><div class="float-right align-top font-size-9"><span class="font-gray-2">3 hours</span><span style="color: #dddddd;">&ampnbsp|&ampnbsp</span><span id="listing-12768571-hopscore" fs="100" qs="100" ms="100" rs="100" class="hopscore-link b font-blue" style="cursor: pointer;">>100</span></div>
<a id="listing-12768571-title" class="font-size-11 listing-title-link b" href="https://www.renthop.com/listings/309-west-76-street/3b/12768571">309 West 76 Street, Apt 3B</a><div id="listing-12768571-neighborhoods" style="margin-top: -1px;" class="font-size-9 overflow-ellipsis">>Upper West Side, Upper Manhattan</div><div style="margin-top: 8px;"><table id="listing-12768571-info"><tr><td id="listing-12768571-price" class="font-size-11 bold" style="padding: 0px 10px 0px 0px; vertical-align: bottom;">$3,000</td><td class="font-size-11" style="border-left: 1px solid #eeeeee; padding: 0px 10px 0px 10px; vertical-align: bottom;"><span style="font-weight: bold; color: #444444;">1 Bed</span></td><td class="font-size-11 bold" style="border-left: 1px solid #eeeeee; padding: 0px 10px 0px 10px; vertical-align: bottom;"><span style="font-weight: bold; color: #444444;">1 Bath</span></td></tr></table></div><div class=""><div class="font-size-9 overflow-ellipsis">
```

4. Upon inspection, it looks like everything we want is contained in this. To verify that, let's copy all of the HTML and paste it into a text editor, and then open it in a browser. I'm going to do that using **Sublime Text**, a popular text editor available at <https://www.sublimetext.com/>.

5. In the following screenshot, you can see that I have pasted the copied HTML from the Jupyter output into Sublime Text and saved it as `test.html`:

HTML text

6. Next, we click on **Open in Browser**, and we can see output that resembles the following image:



Notice that although the text doesn't render cleanly (due to the lack of CSS), all the data we are targeting is there. Fortunately for us, that means the RentHop site doesn't use any advanced JavaScript rendering, so that should make our job much easier. If it did, we'd have to use a different tool like Selenium.

Let's now examine the page elements to see how we can parse the page data:

1. Open the RentHop site in Chrome and right-click anywhere on the page.
2. At the bottom of the context menu, you should see **Inspect**. Click on that. The page should now resemble the following image:

The screenshot shows a web browser with the RentHop website loaded. Two apartment listings are visible: one for West 150th Street and one for East 72nd Street. To the right of the listings is a map of New York City. The developer tools are open, specifically the Elements tab, which displays the HTML code for the current view. The code includes details like listing IDs, addresses, prices, and room counts. The CSS styles section on the right shows specific rules for the search-thumb class, including vertical alignment and media queries for different screen widths.

3. In the tool that just opened, in the upper left-hand corner, there is a square with an arrow in the corner. Click that, and then click on the data on the page. It should look like the following:

This screenshot shows a closer look at the developer tools in the Elements tab. A specific table row for an apartment listing is selected, indicated by a blue highlight. The table has three columns: the first column contains the price (\$2,750), the second column contains the number of bedrooms (1 Bed), and the third column contains the number of bathrooms (1 Bath). The CSS styles for these elements are also visible in the right panel, showing how the font size and boldness are applied.

We can see from this that each listing's data is in a table, and that the first `td` tag contains the price, the second contains the number of bedrooms, and the third contains the number of bathrooms. We will also want the address of the apartment that can be found in an anchor, or a tag.

Let's now begin building out our code to test our parsing of the data. To do our HTML parsing, we are going to use a library call **BeautifulSoup**. The documentation for it can be found at <https://www.crummy.com/software/BeautifulSoup/>. BeautifulSoup is a popular, easy-to-use Python HTML parsing library. It can be pip installed if you don't already have it. We are going to use it to pull out all of the individual specs for our apartment listings:

1. To get started, we simply need to pass our page content into the `BeautifulSoup` class. This can be seen in the following code:

```
from bs4 import BeautifulSoup  
  
soup = BeautifulSoup(r.content, "html5lib")
```

2. We now can use this `soup` object that we've created to begin parsing out our apartment data. The first thing we want to do is retrieve that `div` tag that contains our listing data on the page. We see that in the following code:

```
listing_divs = soup.select('div[class*=search-info]')  
listing_divs
```

What we've done in the preceding code is to select all `div`s that contain `search-info`. These are exactly the `div`s that have our data.

3. Next, we look at the output from this in the following screenshot:

```
In [6]: listing_divs = soup.select('div[class*=search-info]')
listing_divs

Out[6]: [<div class="search-info pr-3 pl-3 pr-md-0 pl-md-4">
<div>
<div class="float-right align-top font-size-9">
<span class="font-gray-2">30 mins</span>
<span style="color: #dddddd;"> | </span>
<span class="hopscore-link b font-blue" fs="100" id="listing-12890730-hopscore" ms="95" qs="100" rs="100" style="cursor: pointer;">100</span>
</div>
<a class="font-size-11 listing-title-link b" href="https://www.renthop.com/listings/2309-astoria-boulevard/2f/12890730" id="listing-12890730-title">2309 Astoria Boulevard, Apt 2F...</a>
<div class="font-size-9 overflow-ellipsis" id="listing-12890730-neighborhoods" style="margin-top: -1px;">
Astoria, Northwestern Queens, Queens
</div>
</div>
<div style="margin-top: 8px;">
<table id="listing-12890730-info">
<tbody><tr>
<td class="font-size-11 bold" id="listing-12890730-price" style="padding: 0px 10px 0px 0px; vertical-align: bottom;">
```

4. Notice that we have a Python list of all the `div` tags we were seeking. We know from looking at the page that there should be twenty of these. Let's confirm that:

```
len(listing_divs)
```

5. We then see the following output, which confirms that we have captured them all as we wanted:

```
len(listing_divs)
```

```
20
```

Pulling out the individual data points

Now that we have all the `divs` with our listing data for each apartment, we need to pull out the individual data points for each apartments.

These are the points in each that we want to target:

- URL of the listing
- Address of the apartment
- Neighborhood
- Number of bedrooms
- Number of bathrooms

Obviously, we love to have way more info—things such as square footage, for example, but we'll have to make do with what we have.

Let's begin by looking at the first listing:

```
listing_divs[0]
```

The preceding code results in the following output:

```
<div class="search-info pr-3 pl-3 pr-md-0 pl-md-4">
<div>
<div class="float-right align-top font-size-9">
<span class="font-gray-2">9 hours</span>
<span style="color: #dddddd; "> | </span>
<span class="hopscore-link b font-blue" fs="99" id="listing-12960379-hopscore" ms="90" qs="100" rs="100" style="cursor: pointer;">100</span>
</div>
<a class="font-size-11 listing-title-link b" href="https://www.renthop.com/listings/439-w-48th-st/3w/12960379" id="listing-12960379-title">439 W 48th St, Apt 3W</a>
<div class="font-size-9 overflow-ellipsis" id="listing-12960379-neighborhoods" style="margin-top: -1px;">
Hell's Kitchen, Midtown Manhattan, Manhattan
</div>
</div>
<div style="margin-top: 8px;">
<table id="listing-12960379-info">
<tbody><tr>
<td class="font-size-11 bold" id="listing-12960379-price" style="padding: 0px 10px 0px 0px; vertical-align: bottom;">
```

Notice that this first `div` contains all of the data points we were looking for. We just now need to begin our parse to target them each individually. Let's look at the first one we want to retrieve, the URL.

We can see that the URL for the page is with an anchor, or a tag. Let's parse that out now. We can do that with another `select` statement, as can be seen in the following code snippet:

```
listing_divs[0].select('a[id*=title]')[0]['href']
```

We see the output in the following screenshot:

```
'https://www.renthop.com/listings/439-w-48th-st/3w/12960379'
```

This is exactly what we were hoping for. We can now continue to retrieve the other data points for the listing. We do that in the following code:

```
href = current_listing.select('a[id*=title]')[0]['href']
addy = current_listing.select('a[id*=title]')[0].string
hood = current_listing.select('div[id*=hood]')[0]\
    .string.replace('\n', '')
```

Let's now verify this by printing out what we've captured. We do that in the following code:

```
print(href)
print(addy)
print(hood)
```

The preceding code results in the following output:

```
https://www.renthop.com/listings/439-w-48th-st/3w/12960379
439 W 48th St, Apt 3W
Hell's Kitchen, Midtown Manhattan, Manhattan
```

Based on this output, we are getting the data we need. Let's continue on with the last few items we need—the bedrooms, bathrooms, and the price.

Since these items have a slightly different presentation in that they are in a `table` tag in our `div` and then inside a table row, or `tr`, we will need to iterate over each point to capture our data. We do that in the following code:

```
listing_specs = listing_divs[0].select('table[id*=info] tr')
for spec in listing_specs:
    spec_data = spec.text.strip().replace(' ', '_').split()
    print(spec_data)
```

The preceding code results in the following output:

```
['$4,500', '3_Bed', '1_Bath']
```

Again, this is exactly what we were looking for. We now have all the data that we were seeking. Let's now pull it all together in a loop so that we can pull the data from each listing and save it into a list.

In the following code, we will pull out all the data points for each listing:

```
listing_list = []
for idx in range(len(listing_divs)):
    indv_listing = []
    current_listing = listing_divs[idx]
    href = current_listing.select('a[id*=title]')[0]['href']
    addy = current_listing.select('a[id*=title]')[0].string
    hood = current_listing.select('div[id*=hood]')[0]\
        .string.replace('\n', '')
    indv_listing.append(href)
    indv_listing.append(addy)
    indv_listing.append(hood)
    listing_specs = current_listing.select('table[id*=info] tr')
    for spec in listing_specs:
        try:
            indv_listing.extend(spec.text.strip()\
                .replace(' ', '_').split())
        except:
            indv_listing.extend(np.nan)
    listing_list.append(indv_listing)
```

Let's unpack a bit what we did in the preceding code. We know we have 20 divs that contain the apartment listing on the page, so we create a `for` loop that goes through each one and pulls out the data and adds it to `indv_listing`. When that is complete, all the data for the individual listing is then added to the `listing_list`, which contains all the final info for the 20 apartment listings. We verify that with the following code:

```
listing_list
```

The preceding code results in the following output:

```
[['https://www.renthop.com/listings/439-w-48th-st/3w/12960379',
  '439 W 48th St, Apt 3W',
  "Hell's Kitchen, Midtown Manhattan, Manhattan",
  '$4,500',
  '3_Bed',
  '1_Bath'],
 ['https://www.renthop.com/listings/rutgers-street/4hs/12959229',
  'Rutgers Street',
  'Two Bridges, Downtown Manhattan, Manhattan',
  '$3,895',
  '1_Bed',
  '1_Bath'],
 ['https://www.renthop.com/listings/485-hicks-street/1/11481619',
  '485 Hicks Street, Apt 1',
  'Cobble Hill, South Brooklyn, Brooklyn',
  '$1,999',
  '1_Bed',
  '1_Bath'],
 ['https://www.renthop.com/listings/east-35th-street/17g/12930185',
  'East 35th Street',
```

Again, we appear to be getting the results we expect, so we will continue on. A check of the number of items in `listing_list` also confirms we have all 20 apartments on the page.

So far, we have successfully retrieved one page of data. While that is great, we are going to need far more apartments if we want to build any kind of meaningful model. To do this, we will need to iterate over a number of pages. To that end, we'll need to use the appropriate URLs. We can see that at the bottom of the listings, there is a button that says **Next**. If you right-click on that button, and click **Copy Link Address**, you see it looks like the following URL: https://www.renthop.com/search/nyc?max_price=50000min_price=0page=2sort=hopscoreq=search=0.

Parsing data

A basic analysis of the URL tells us that we are passing in parameters that include min price and max price, but most importantly, the page number. We can use this in our code, and just dynamically change that page number to pull additional pages using a loop.

Let's try this with some sample code:

```
url_prefix =
"https://www.renthop.com/search/nyc?max_price=50000&min_price=0&page="
page_no = 1
```

```
url_suffix = "&sort=hopscore&q=&search=0"
for i in range(3):
    target_page = url_prefix + str(page_no) + url_suffix
    print(target_page)
    page_no += 1
```

The preceding code results in the following output:

```
https://www.renthop.com/search/nyc?max_price=50000&min_price=0&page=1&sort=hopscore&q=&search=0
https://www.renthop.com/search/nyc?max_price=50000&min_price=0&page=2&sort=hopscore&q=&search=0
https://www.renthop.com/search/nyc?max_price=50000&min_price=0&page=3&sort=hopscore&q=&search=0
```

This looks like a success. Now we need to just put it all together. We'll start by turning our parsing loop into a proper function that we can call for each of the pages. We do that in the following code:

```
def parse_data(listing_divs):
    listing_list = []
    for idx in range(len(listing_divs)):
        indv_listing = []
        current_listing = listing_divs[idx]
        href = current_listing.select('a[id*=title]')[0]['href']
        addy = current_listing.select('a[id*=title]')[0].string
        hood = current_listing.select('div[id*=hood]')[0]\
            .string.replace('\n', '')
        indv_listing.append(href)
        indv_listing.append(addy)
        indv_listing.append(hood)

        listing_specs = current_listing.select('table[id*=info] tr')
        for spec in listing_specs:
            try:
                values = spec.text.strip().replace(' ', '_').split()
                clean_values = [x for x in values if x != '_']
                indv_listing.extend(clean_values)
            except:
                indv_listing.extend(np.nan)
        listing_list.append(indv_listing)
    return listing_list
```

This function will take in a page full of `listing_divs` and return the data payload for each. We can then keep adding the data to our master list of apartment data. Notice that there is some additional code in there to validate and remove some erroneous '`_`' values that get added in the `listing_spec` loop. This was to avoid some bad parsing that added an additional column when there shouldn't have been one.

Next, we will build the main loop that will retrieve each page, get the `listing_divs`, parse out the data points, and finally add all of the info to our final Python list of all data points for each listing. We do that in the following code:

```
all_pages_parsed = []
for i in range(100):
    target_page = url_prefix + str(page_no) + url_suffix
    print(target_page)
    r = requests.get(target_page)
    soup = BeautifulSoup(r.content, 'html5lib')
    listing_divs = soup.select('div[class*=search-info]')
    one_page_parsed = parse_data(listing_divs)
    all_pages_parsed.extend(one_page_parsed)
    page_no += 1
```

Before trying this on 100 pages, you should confirm that it works on a much smaller number, like 3.

You should have noticed the page being printed out as the code ran. If you used 30 pages, you should see that there are 2,000 listings in your `all_pages_parsed` list.

Let's now move our data into a pandas DataFrame, so that we can work with it more easily. We do that in the following code:

```
df = pd.DataFrame(all_pages_parsed, columns=['url', 'address',
    'neighborhood', 'rent', 'beds', 'baths'])

df
```

The preceding code results in the following output:

		url	address	neighborhood	rent	beds	baths
0		https://www.renthop.com/listings/east-75th-street-apartments-new-york-city-nyc-rent/	East 75th Street	Upper East Side, Upper Manhattan, Manhattan	\$2,900	2_Bed	1_Bath
1		https://www.renthop.com/listings/15-w-103rd-street-apartments-new-york-city-nyc-rent/	15 W 103rd St., Apt 1D	Manhattan Valley, Upper West Side, Upper Manhattan	\$4,595	Studio	1.5_Bath
2		https://www.renthop.com/listings/private-balconies-new-york-city-nyc-rent/	871 Bergen Street, Apt 4	Crown Heights, Central Brooklyn, Brooklyn	\$4,150	3_Bed	1_Bath
3		https://www.renthop.com/listings/1-west-street-apartments-new-york-city-nyc-rent/	1 West Street, Apt 2201	Financial District, Downtown Manhattan, Manhattan	\$2,800	1_Bed	1_Bath
4		https://www.renthop.com/listings/stanton-street-apartments-new-york-city-nyc-rent/	Stanton Street	Lower East Side, Downtown Manhattan, Manhattan	\$3,900	2_Bed	1_Bath
5		https://www.renthop.com/listings/long-island-city-apartments-new-york-city-nyc-rent/	Long Island City, Hunters Point	Long Island City, Northwestern Queens, Queens	\$3,100	1_Bed	1_Bath
6		https://www.renthop.com/listings/west-end-avenue-apartments-new-york-city-nyc-rent/	West End Ave.	Lincoln Square, Upper West Side, Upper Manhattan	\$6,100	3_Bed	2_Bath

Now that we have all our data pulled down, parsed, and incorporated in a DataFrame, let's move on to cleansing and verifying our data.

Inspecting and preparing the data

Let's begin by inspecting the data points for each of our columns. We want to look for odd and outlier values in our data. We will start by looking at the bedroom and bathroom columns:

1. In the following code, we look at the unique values for bedrooms:

```
df['beds'].unique()
```

The preceding code results in the following output:

```
array(['1_Bed', '4_Bed', '2_Bed', 'Studio', '3_Bed', '5_Bed', '_Studio',
       'Loft', '_1_Bed', '6_Bed', '_3_Bed', '_2_Bed'], dtype=object)
```

2. Now, let's look at bathrooms. We do that in the following code:

```
df['baths'].unique()
```

The preceding code results in the following output:

```
array(['1_Bath', '1.5_Bath', '2_Bath', '2.5_Bath', '_1_Bath', '3_Bath',
      '3.5_Bath', '_2.5_Bath', '_2_Bath', '4_Bath', '_3_Bath', '5_Bath',
      '4.5_Bath'], dtype=object)
```

3. Based on the output from the two preceding queries, we see that we need to correct some items that have a leading underscore. Let's do that now:

```
df['beds'] = df['beds'].map(lambda x: x[1:] if x.startswith('_')
                           else x)
df['baths'] = df['baths'].map(lambda x: x[1:] if x.startswith('_')
                           else x)
```

4. In the preceding code, we ran a pandas `map` function with a `lambda` function that essentially checks whether the element begins with an underscore and, if so, removes it. A quick check of the unique values for beds and baths should reveal that our erroneous starting underscores have been removed:

```
df['beds'].unique()
```

The preceding code results in the following output:

```
array(['1_Bed', '4_Bed', '2_Bed', 'Studio', '3_Bed', '5_Bed', 'Loft',
      '6_Bed'], dtype=object)
```

Let's execute the following line of code and look at the results:

```
df['baths'].unique()
```

The preceding code results in the following output:

```
array(['1_Bath', '1.5_Bath', '2_Bath', '2.5_Bath', '3_Bath', '3.5_Bath',
      '4_Bath', '5_Bath', '4.5_Bath'], dtype=object)
```

5. Next, we want to look at some descriptive statistics to better understand our data. One way to do that is with the `describe` method. Let's try that in the following code:

```
df.describe()
```

The preceding code results in the following output:

	url	address	neighborhood	rent	beds	baths
count	2000	1998		2000	2000	2000
unique	1773	1292		125	647	8
top	https://www.renthop.com/listings/106-ainslie-s...	York Ave.	Hell's Kitchen, Midtown Manhattan, Manhattan	\$3,200	1_Bed	1_Bath
freq	3	15		171	38	654
						1462

While we were hoping to get metrics such as the average number of beds and baths, and things like the max rent, what we instead received was much less than that. The problem is that the data is not the correct data type for these operations. Pandas can't perform those types of operation on what are string objects. We will need to clean up our data further and set it to the correct data types. We will do that in the following code:

```
df['rent'] = df['rent'].map(lambda x:
str(x).replace('$','').replace(',','')).astype('int')
df['beds'] = df['beds'].map(lambda x: x.replace('_Bed', ''))
df['beds'] = df['beds'].map(lambda x: x.replace('Studio', '0'))
df['beds'] = df['beds'].map(lambda x: x.replace('Loft', '0')).astype('int')
df['baths'] = df['baths'].map(lambda x: x.replace('_Bath',
'')).astype('float')
```

What we have done in the preceding code is to remove anything that is non-numeric from each of the values. You can see that we removed `_Bed` and `_Bath` to leave just the number, and that we replaced words such as `Studio` and `Loft` with the actual number of bedrooms, which is zero.

Sneak-peek at the data types

Let's now look at our data types:

```
df.dtypes
```

The preceding code results in the following output:

url	object
address	object
neighborhood	object
rent	int64
beds	int64
baths	float64
dtype:	object

This is what we want to see. Notice that since we can have a half bath, we needed a float there rather than an integer.

Next, let's carry out an inspection. Let's get a count of the number of units in each neighborhood:

```
df.groupby('neighborhood')['rent'].count().to_frame('count')\n    .sort_values(by='count', ascending=False)
```

The preceding code generates the following output:

neighborhood	count
Hell's Kitchen, Midtown Manhattan, Manhattan	171
Upper East Side, Upper Manhattan, Manhattan	115
Financial District, Downtown Manhattan, Manhattan	108
Upper West Side, Upper Manhattan, Manhattan	79
Yorkville, Upper East Side, Upper Manhattan, Manhattan	67
Williamsburg, Northern Brooklyn, Brooklyn	65
Murray Hill, Midtown Manhattan, Manhattan	64
Long Island City, Northwestern Queens, Queens	61
Rose Hill, Kips Bay, Midtown Manhattan, Manhattan	55
Bedford-Stuyvesant, Northern Brooklyn, Brooklyn	53

It looks like most of the units are in Manhattan, which is what we might expect. Let's make sure that our neighborhood strings are clean. We can do that by doing a number of groupby operations:

```
df[df['neighborhood'].str.contains('Upper East Side')]['neighborhood'].value_counts()
```

The preceding code generates the following output:

Upper East Side, Upper Manhattan, Manhattan	115
Yorkville, Upper East Side, Upper Manhattan, Manhattan	67
Lenox Hill, Upper East Side, Upper Manhattan, Manhattan	9
Carnegie Hill, Upper East Side, Upper Manhattan, Manhattan	5
Upper East Side, Upper Manhattan, Manhattan	1
Name: neighborhood, dtype: int64	

It looks like we have some issues with leading and possibly trailing spaces. Let's clean that up. We do so in the following code:

```
df['neighborhood'] = df['neighborhood'].map(lambda x: x.strip())
```

That should clear it up. Let's validate that:

```
df[df['neighborhood'].str.contains('Upper East Side')]['neighborhood'].value_counts()
```

The preceding code results in the following output:

Upper East Side, Upper Manhattan, Manhattan	116
Yorkville, Upper East Side, Upper Manhattan, Manhattan	67
Lenox Hill, Upper East Side, Upper Manhattan, Manhattan	9
Carnegie Hill, Upper East Side, Upper Manhattan, Manhattan	5
Name: neighborhood, dtype: int64	

Perfect. Exactly what we want to see. At this point, we can do a few more inspections. Let's just take a look at the mean rent by neighborhood:

```
df.groupby('neighborhood')['rent'].mean().to_frame('mean')\n    .sort_values(by='mean', ascending=False)
```

The preceding code results in the following output:

neighborhood	mean
Lincoln Square, Upper West Side, Upper Manhattan, Manhattan	7313.784314
SoHo, Downtown Manhattan, Manhattan	7177.272727
Tribeca, Downtown Manhattan, Manhattan	6571.400000
Governors Island, Manhattan	6000.000000
Midtown East, Midtown Manhattan, Manhattan	5884.200000
Flatiron District, Midtown Manhattan, Manhattan	5765.333333
North Slope, Park Slope, South Brooklyn, Brooklyn	5497.500000
Battery Park City, Downtown Manhattan, Manhattan	5388.578947
DUMBO, Northwestern Brooklyn, Brooklyn	5273.600000
Little Italy, Downtown Manhattan, Manhattan	5268.000000

We see that the Lincoln Square area appears to have the highest rent on average. At this point, we could continue on querying the data for interesting patterns, but let's move on to visualizing the data.

Visualizing our data

When dealing with geographic data, as we are here, it is immensely valuable to be able to plot that information. One way of doing that is with something called a **choropleth** map. A choropleth is essentially a geographic heat map. We are going to build a choropleth to create a heat map of average rental price by ZIP code.

The first thing we will need to do this is the ZIP code. Unfortunately for us, our dataset does not contain ZIP code information. We do, however, have the address for the properties. With a little help from the Google Maps API, we can retrieve this information.

Currently, the Google Maps API is a paid API. The rates are reasonable, 1,000 calls for \$5, but they also give you a credit of \$200 each month (at the time of writing). They also allow you to sign up for a free trial before they will start billing you, and they won't bill unless you explicitly give them the okay to do so. Since there really is no free alternative out there, we'll go ahead and sign up for an account. I'll walk you through the steps in the following:

1. The first step is to go to the Google Maps API page at <https://developers.google.com/maps/documentation/geocoding/intro>:

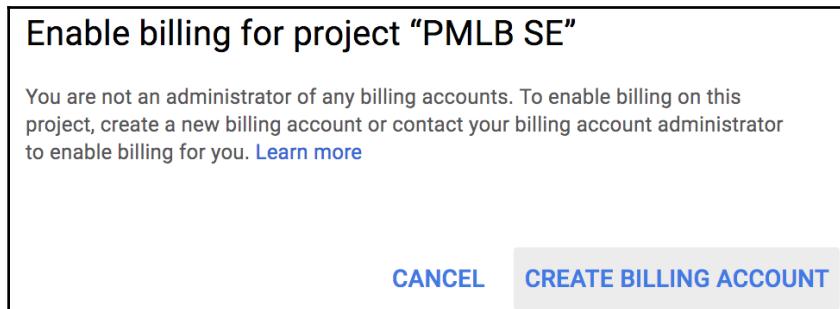
The screenshot shows the 'Geocoding API' section of the Google Maps Platform documentation. At the top, there's a navigation bar with links for Overview, Products, Pricing, Documentation (selected), Search, and ALL PRODUCTS. Below the navigation is a search bar and a 'GET STARTED' button. The main content area has a blue header bar with the text 'New pricing changes went into effect on July 16, 2018. For more information, check out the [Guide for Existing Users](#)'. To the left is a sidebar with links for Get Started, Developer Guide, Get API Key, Best Practices Geocoding Addresses, Forward Geocoder FAQ, Web Services, Best Practices, Client Libraries, Policies and Terms, Usage and Billing, Optimizing Quota Usage, Policies, and Terms of Service. The main content area features a 'Get Started' heading with a five-star rating. It explains what the Geocoding API does and provides links to client-side APIs and Java/Python/Go/Node.js clients. Below this, definitions for 'Geocoding' and 'Reverse geocoding' are provided, along with a note about finding addresses by place ID. On the right side, there's a sidebar with links for Contents, Sample request and response, Geocoding request and response (latitude/longitude lookup), Reverse geocoding request and response (address lookup), Start coding with our client libraries, Authentication, quotas, pricing, and policies, Activate the API and get an API key, Quotas and pricing, Policies, and Learn more.

2. Click on **GET STARTED** in the upper right-hand corner. You'll next be prompted to create a project. Give it any name you like:

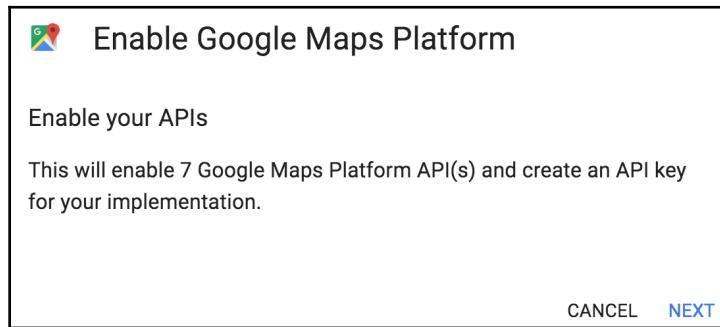


Creating a project

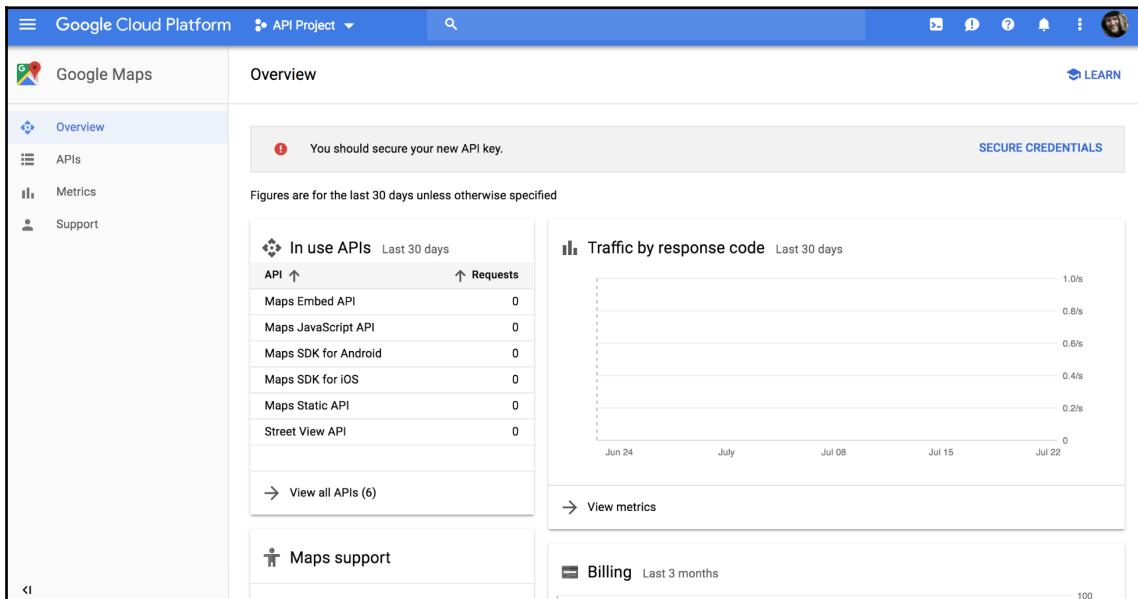
3. Then you will enable billing:



4. Next, you will enable your API keys:



5. Once this is completed and you have your API keys, head back to the front page to enable the Geolocation API. Click on **APIs** in the left-hand side pane:



6. And then, under **Unused APIs**, click **Geolocation API**:

Unused APIs

Select an API to view details in Marketplace.

API
Directions API
Distance Matrix API
Maps Elevation API
Geocoding API
Geolocation API
Roads API
Time Zone API
Places SDK for Android
Places SDK for iOS
Places API for Web

Once all of this is complete, and you have your API keys, pip install Google Maps. That can be done from your command line with `pip install -U googlemaps`.

Let's continue on now with this API in our Jupyter Notebook. We'll import our new mapping API and test it out:

```
import googlemaps

gmaps = googlemaps.Client(key='YOUR_API_KEY_Goes_Here')

ta = df.loc[3,['address']].values[0] + ' \' \
+ df.loc[3,['neighborhood']].values[0].split(', ')[-1]

ta
```

The preceding code results in the following output:

```
'355 South End Avenue, Apt 4J Manhattan'
```

Okay, so essentially, all we did in the final bit of code was to import and initialize our `googlemaps` client, as well as use piece together from one of our apartments as usable address. Let's now pass in that address to the Google Maps API:

```
geocode_result = gmaps.geocode(ta)

geocode_result
```

The preceding code generates the following output:

```
{'long_name': 'New York County',
 'short_name': 'New York County',
 'types': ['administrative_area_level_2', 'political']},
 {'long_name': 'New York',
 'short_name': 'NY',
 'types': ['administrative_area_level_1', 'political']},
 {'long_name': 'United States',
 'short_name': 'US',
 'types': ['country', 'political']},
 {'long_name': '10280', 'short_name': '10280', 'types': ['postal_code']},
 'formatted_address': '355 South End Ave #4J, New York, NY 10280, USA',
 'geometry': {'location': {'lat': 40.7109037, 'lng': -74.01662329999999},
 'location_type': 'ROOFTOP',
 'viewport': {'northeast': {'lat': 40.7122526802915,
 'lng': -74.01527431970848},
 'southwest': {'lat': 40.7095547197085, 'lng': -74.01797228029149}}},
 'place_id': 'Ei40SiwgMzU1IFNvdXRoIEVuZCBBdmUsIE5ldyBZb3JrLCBOWSAxMDI4MCwgVVNBih4aHAoWChQKEg
m1M--QGlrciRHPztxnTZeHLhICNeo',
 'types': ['subpremise']}
```

Remember, we are looking to extract just the ZIP code here. The ZIP code is embedded in the JSON, but it will take a bit of work to extract due to the formatting of this response JSON object. Let's do that now:

```
for piece in geocode_result[0]['address_components']:
    if 'postal_code' in piece['types'] :
        print(piece['short_name'])
```

The preceding code results in the following output:

10280

It looks like we're getting the information we want. There is one caveat, however. Looking deeper into the address column, we can see that occasionally, a full address is not given. This will result in no ZIP code coming back. We'll just have to deal with that later. For now, let's build a function to retrieve the ZIP codes that we can do as follows:

```
import re
def get_zip(row):
    try:
        addy = row['address'] + ' ' + row['neighborhood'].split(', ')[-1]
        print(addy)
        if re.match('^\d+\s\w', addy):
            geocode_result = gmaps.geocode(addy)
            for piece in geocode_result[0]['address_components']:
                if 'postal_code' in piece['types']:
                    return piece['short_name']
```

```
        else:  
            pass  
        else:  
            return np.nan  
    except:  
        return np.nan  
  
df['zip'] = df.apply(get_zip, axis=1)
```

There's a fair bit of code in the preceding snippet, so let's talk about what's going on here.

First, at the bottom, you see that we are running an `apply` method on our DataFrame. Because we have set `axis=1`, each row of the `df` DataFrame will be passed into our function. Within the function, we are piecing together an address to call with the Google Maps Geolocation API. We are using regex to limit our calls to only those that start with a street number. We then iterate over the JSON response to parse out the ZIP code. If we find a ZIP code, we return it, otherwise we return a `np.nan`, or null value. Note that this function will take some time to run as we have to make many hundreds of calls and then parse out the response.

Once that completes, we will have a DataFrame that now has the ZIP code for those properties that had a proper address provided. Let's take a look and see how many that actually is:

```
df[df['zip'].notnull()].count()
```

The preceding code generated the following output:

url	555
address	555
neighborhood	555
rent	555
beds	555
baths	555
zip	555
dtype:	int64

So, we lost quite a bit of our data, but nevertheless, what we have now is more useful in many ways, so we will continue on.

First, since it takes so long to retrieve all the ZIP code data, let's now store what we have so that we can always retrieve it later if necessary, and not have to make all those API calls again. We do that with the following code:

```
df.to_csv('apts_with_zip.csv')
```

Let's also store just the data with the ZIP code information in a new DataFrame. We will call that one zdf:

```
zdf = df[df['zip'].notnull()].copy()
```

Finally, let's do an aggregation by ZIP code to see what the average rental price is by ZIP:

```
zdf_mean = zdf.groupby('zip')['rent'].mean().to_frame('avg_rent')\n.zsort_values(by='avg_rent', ascending=False).reset_index()\nzdf_mean
```

The preceding code generates the following output:

	zip	avg_rent
0	10069	7485.000000
1	10030	7347.500000
2	10014	6615.000000
3	10001	6000.000000
4	10013	5825.000000
5	10282	5377.500000
6	10012	5058.500000
7	10065	4956.333333
8	10011	4923.636364
9	10038	4882.611111
10	10280	4852.250000

We can see this jibes with our earlier finding that the Lincoln Center area had the highest mean rental prices, since 10069 is in the Lincoln Center region.

Let's now move on to visualizing this information.

Visualizing the data

Since this data is based on ZIP codes, the best way to visualize it is with a choropleth. If you're unfamiliar with a choropleth, it's simply a visualization that represents the data according to a color spectrum. Let's create one now using a Python mapping library called folium at <https://github.com/python-visualization/folium>. If you don't have folium installed, again, it can be done with pip install on the command line.

Now we'll go ahead and create our visualization:

```
import folium

m = folium.Map(location=[40.748817, -73.985428], zoom_start=13)

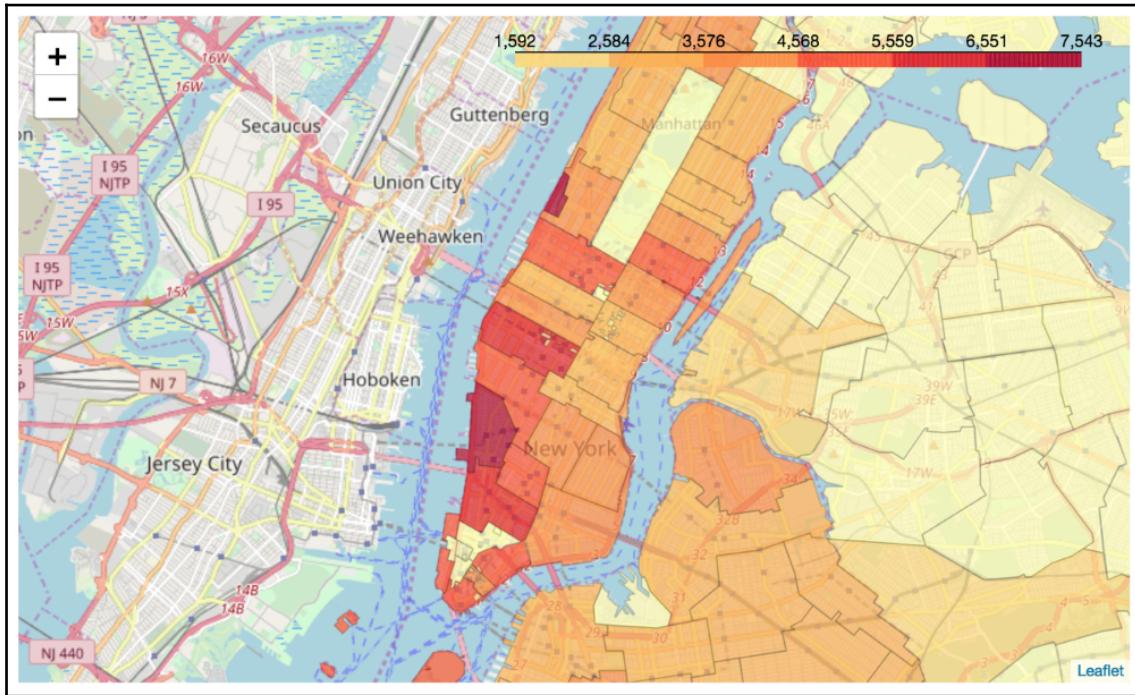
m.choropleth(
    geo_data=open('nyc.json').read(),
    data=zdf_mean,
    columns=['zip', 'avg_rent'],
    key_on='feature.properties.postalCode',
    fill_color='YlOrRd', fill_opacity=0.7, line_opacity=0.2,
)

m
```

There's a lot going on here, so let's take it step by step:

1. After importing `folium`, we create a `.Map()` object. We need to pass in coordinates and a zoom level to center the map. A Google search for the coordinates of the Empire State Building will give us the proper lat and long (flip the sign on the longitude to render it properly). Finally, adjust the zoom to get it centered appropriately for our data.
2. The next line requires something called a GeoJSON file. This is an open format for representing geographic attributes. This can be found by searching for NYC GeoJSON files—specifically, ones with ZIP code mappings. Once that is done, we reference the GeoJSON file by inputting its path.
3. Next, we reference our DataFrame in the `data` parameter. Here, we are using the mean rent by ZIP code we created previously. The `columns` parameter references those. The `key_on` parameter references the part of our JSON file that we are targeting, in this instance, the `postalCode`.
4. Finally, the other options determine the color palette and certain other parameters to adjust the legend and coloring.

When the cell is run, the map should render inline in your Jupyter Notebook, as can be seen in the following diagram:



With the heat map completed, you can begin to get a sense of which areas have higher or lower rents. This could help when targeting a particular area, but let's take our analysis deeper by using regression modeling.

Modeling the data

Let's begin modeling by using our dataset. We're going to examine the effect that the ZIP code and the number of bedrooms have on the rental price. We'll use two packages here: the first, `statsmodels`, we introduced in Chapter 1, *The Python Machine Learning Ecosystem*, but the second, `patsy`,

<https://patsy.readthedocs.org/en/latest/index.html>, is a package that makes working with `statsmodels` easier. Patsy allows you to use R-style formulas when running a regression. Let's do that now:

```
import patsy
import statsmodels.api as sm

f = 'rent ~ zip + beds'
y, X = patsy.dmatrices(f, zdf, return_type='dataframe')

results = sm.OLS(y, X).fit()
results.summary()
```

The preceding code generates the following output:

OLS Regression Results						
Dep. Variable:		rent	R-squared:		0.447	
Model:		OLS	Adj. R-squared:		0.367	
Method:		Least Squares	F-statistic:		5.587	
Date:		Sun, 22 Jul 2018	Prob (F-statistic):		3.50e-31	
Time:		23:22:57	Log-Likelihood:		-4770.2	
No. Observations:		555	AIC:		9682.	
Df Residuals:		484	BIC:		9989.	
Df Model:		70				
Covariance Type: nonrobust						
	coef	std err	t	P> t	[0.025	0.975]
Intercept	1798.5905	584.816	3.075	0.002	649.499	2947.682
zip[T.07310]	664.0111	739.535	0.898	0.370	-789.084	2117.106
zip[T.10001]	3416.5381	1513.657	2.257	0.024	442.388	6390.688
zip[T.10002]	1125.9781	652.024	1.727	0.085	-155.170	2407.126
zip[T.10003]	1182.1297	664.910	1.778	0.076	-124.337	2488.596

Note that the preceding output is truncated.

With those few lines of code, we have just run our first machine learning algorithm.



While most people don't tend to think of linear regression as machine learning, that's exactly what it is. Linear regression is a type of supervised machine learning. Supervised, in this context, simply means we provide the output values for our training set.

Let's now unpack what happened there. After our imports, we have two lines that relate to the `patsy` module. The first line is the formula we will be using. On the left-hand side (before the tilde) is our response, or dependent, variable, `rent`. On the right-hand side, we have our independent, or predictor, variables, `zip` and `beds`. This formula simply means we want to know how the ZIP code and the number of bedrooms will affect the rental price.

Our formula is then passed into `patsy.dmatrices()` along with our `DataFrame` containing corresponding column names. Patsy is then set to return a `DataFrame` with our `x` matrix of predictor variables and a `y` vector with our response variable. These are then passed into `sm.OLS()`, on which we also call `.fit()` to run our model. Finally, we print out the results of the model.

As you can see, there is a lot of information provided in the resulting output. Let's begin by looking at the topmost section. We see that the model included 555 observations, that it has an adjusted R^2 of .367, and that it is significant with an F-statistic probability of $3.50e-31$. What is the significance of this? It means that we have created a model that is able to explain about a third of the variance in price using just bedrooms and ZIP code. Is this a good result? In order to better answer that, let's now look at the center section of the output.

The center section provides us with information on each of the independent variables in our model. From left to right, we see the following: the variable, the variable's coefficient in the model, the standard error, the t -statistic, the p -value for the t -statistic, and a 95% confidence interval.

What does all of this tell us? If we look at the p -value column, we can determine whether our individual variables are statistically significant. Statistically significant in a regression model means that the relationship between an independent variable and a response variable is unlikely to have occurred by chance. Typically, statisticians use a p -value of .05 when determining this. A .05 p -value means that the results we see would occur by chance only 5% of the time. In terms of our output here, the number of bedrooms is clearly significant. What about the ZIP codes?

The first thing to notice here is that our intercept represents the 07302 ZIP code. When modeling a linear regression, an intercept is needed. The intercept is simply where the regression line meets the y axis. Statsmodels will automatically select one of the predictor variables to use as the intercept. Here it decided on Jersey City, 07302, since it organized the ZIP codes in ascending order. We can confirm this by examining the data as follows:

```
x
```

The preceding code generates the following output:

	Intercept	zip[T.07310]	zip[T.10001]	zip[T.10002]	zip[T.10003]	zip[T.10004]	zip[T.10005]	zip[T.10009]	zip[T.10010]
3	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
5	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
6	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
7	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
12	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
13	1.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0
15	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
22	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
24	1.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0
29	1.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0
30	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

Notice that they are in ascending order, and if we look at the sorted ZIP code values in our DataFrame, we see the same with the exception of the missing ZIP 07302, which is now our baseline against which all the others will be compared.

Looking at our results output again, we notice that some ZIP codes are highly significant and others are not. Let's look at our old friend, the Lincoln Center neighborhood, or 10069. If you remember, it was the area with the highest rents in our sample. We would expect that it would be significant and have a large positive coefficient when compared to the baseline of Jersey City, and, in fact, it does. The p -value is 0.000, and the coefficient is 4116. This means that you can expect the rent to be significantly higher near Lincoln Center, compared to an equivalent apartment in Jersey City—no surprise there.

Let's now use our model to make a number of forecasts.

Forecasting

Let's say we've decided from our prior analysis that we are interested in three particular ZIP codes: 10002, 10003, and 10009. How can we use our model to determine what we should pay for a given apartment? Let's now take a look.

First, we need to know what the inputs into the model looked like so that we know how to enter a new set of values. Let's take a look at our X matrix:

```
X.head()
```

The preceding code generates the following output:

	Intercept	zip[T.07310]	zip[T.10001]	zip[T.10002]	zip[T.10003]	zip[T.10004]	zip[T.10005]	zip[T.10009]	zip[T.10010]
3	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
5	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
6	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
7	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
12	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

What we see is that our input is coded with what are called **dummy variables**. To represent a ZIP code feature, since it is not numerical, dummy coding is used. If the apartment is in 10003, then that column will be coded as 1, while all other ZIP codes are coded as 0. Beds will be coded according to the actual number since they are numerical. So let's now create our own input row to predict:

```
to_pred_idx = X.iloc[0].index
to_pred_zeros = np.zeros(len(to_pred_idx))
tpdf = pd.DataFrame(to_pred_zeros, index=to_pred_idx, columns=['value'])

tpdf
```

The preceding code generates the following output:

	value
Intercept	0.0
zip[T.07310]	0.0
zip[T.10001]	0.0
zip[T.10002]	0.0
zip[T.10003]	0.0
zip[T.10004]	0.0
zip[T.10005]	0.0
zip[T.10009]	0.0
zip[T.10010]	0.0
zip[T.10011]	0.0
zip[T.10012]	0.0

We have just used the index from the `X` matrix and filled in the data with all zeros. Let's now fill in our values. We are going to price a one-bedroom apartment in the 10009 area code:

```
tpdf.loc['Intercept'] = 1  
tpdf.loc['beds'] = 1  
tpdf.loc['zip[T.10009]'] = 1  
  
tpdf
```



The intercept value for a linear regression must always be set to 1 for the model in order to return accurate statistical values.

The preceding code generates the following output:

	value
Intercept	1.0
zip[T.07310]	0.0
zip[T.10001]	0.0
zip[T.10002]	0.0
zip[T.10003]	0.0
zip[T.10004]	0.0
zip[T.10005]	0.0
zip[T.10009]	1.0
zip[T.10010]	0.0
zip[T.10011]	0.0
zip[T.10012]	0.0

We have set our features to the appropriate values, so let's now use our model to return a prediction. We'll need to convert it to a DataFrame and transpose it in order to get the correct format. We do this as follows:

```
results.predict(tpdf['value'].to_frame().T)
```

The preceding code generates the following output:

value	3203.660719
dtype:	float64

You will recall that `results` was the variable name we saved our model to. That model object has a `.predict()` method, which we call with our input values. And, as you can see, the model returns a predicted value.

What if we want to add another bedroom? We can do it as follows:

1. Let's change our inputs and see:

```
tpdf['value'] = 0  
tpdf.loc['Intercept'] = 1  
tpdf.loc['beds'] = 2  
tpdf.loc['zip[T.10009]'] = 1
```

2. Then we'll run the prediction again:

```
results.predict(tpdf['value'].to_frame().T)
```

The preceding code generates the following output:

value	3988.532144
dtype:	float64

3. It looks like that extra bedroom will cost us about \$800 more a month. But what if we choose 10069 instead? Lets change our input and see:

```
tpdf['value'] = 0  
tpdf.loc['Intercept'] = 1  
tpdf.loc['beds'] = 2  
tpdf.loc['zip[T.10069]'] = 1  
  
results.predict(tpdf['value'].to_frame().T)
```

The preceding code generates the following output:

value	7485.0
dtype:	float64

According to our model, two bedrooms in the Lincoln Center area is going to cost a pretty penny compared to the East Village.

Extending the model

At this point, we have only examined the relationship between the ZIP code, bedrooms, and rental price. And while our model had some explanatory benefit, we had a minimal dataset and far too few features to adequately examine the complex world of real estate valuation.

Fortunately, however, if we were to add more data and features to the model, we could use the exact same framework to expand our analysis.

Some possible future extensions to explore would be utilizing data for restaurants and bars available from APIs such as Foursquare or Yelp, or walkability and transportation-proximity measures from providers such as Walk Score.

There are a number of ways to extend the model, and I suggest if you do pursue working on a project such as this that you explore a variety of measures. More data is released every day and, with it, models can only improve.

Summary

In this chapter, we learned how to acquire data on real estate listings, how to utilize the functionality of pandas to manipulate and sanitize that data, how to inspect the data visually with choropleths, and finally, how to build and use regression modeling to price out an apartment.

At this point, we have just touched the surface of machine learning. In the chapters that follow, we'll go further into how to evaluate the quality of our model, and we'll also learn how to turn them into full-scale solutions.

3

Build an App to Find Cheap Airfares

Let's talk about mistakes. They're part of life; everyone makes them—even airlines.

In 2014, I happened to be reading my Twitter feed one afternoon when one of the accounts I follow tweeted that a major US airline had fares to Europe that were significantly below what would be expected. At the time, the cheapest fare from New York to Vienna was around \$800, but the advertised fares for a select number of dates were between \$350 and \$450. This seemed too good to be true. But it wasn't. I had lucked upon what's known in the industry as a *mistake fare*.

In the super-secretive society of travel hackers and mileage junkies, it's well-known that airlines occasionally—and accidentally—post fares that exclude fuel surcharges. And remarkably, this isn't the only type of mistake they make. You might expect advanced algorithms would be updating fares for each flight, taking into account an enormous number of factors. And for the most part, you'd be right. But due to legacy systems and the complexity of dealing with multiple carriers and multiple jurisdictions, mistakes do sometimes occur.

Here is a list of some of these more recent mistake fares:

- In 2007, United Airlines sold business fare tickets from San Francisco to New Zealand for \$1,500
- In 2013, Delta sold economy fare tickets from multiple US cities to Hawaii for \$6.90
- In 2015, American Airlines sold business class fares to China from Washington, DC for \$450

Now that you know these fares exist, how can you get in on them? Machine learning, of course! Since they typically last just a few hours before they disappear, we're going to build an application that continually monitors fare pricing, checking for anomalous prices that will generate an alert we can quickly act on.

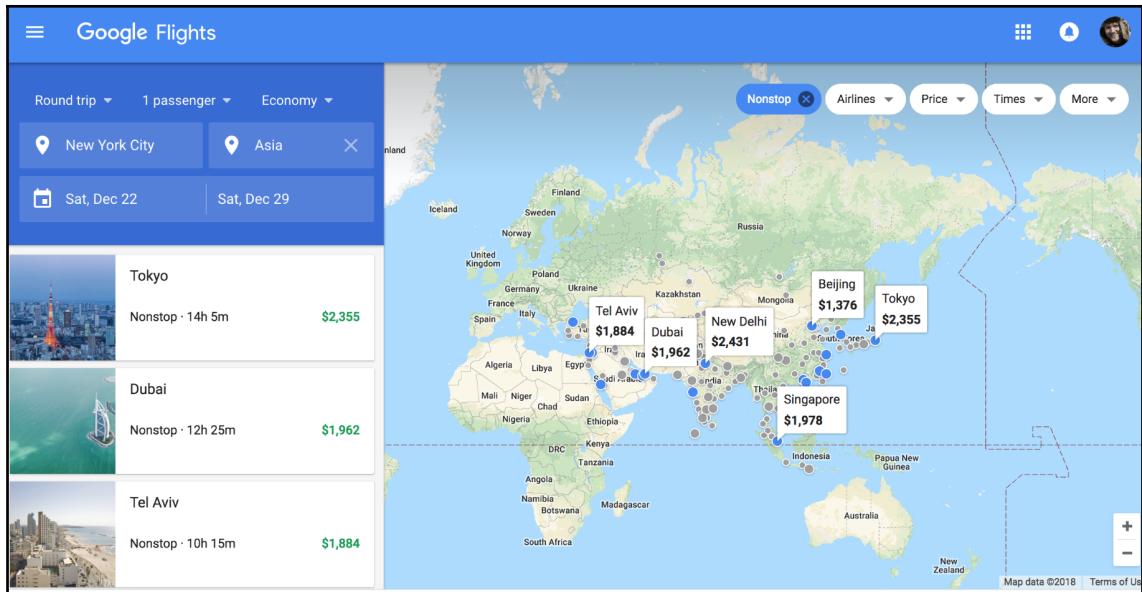
Here's what we'll cover in this chapter:

- Sourcing airfare pricing on the web
- Retrieving fare data with advanced web scraping techniques
- Parsing the DOM to extract prices
- Identifying outlier fares with anomaly detection techniques
- Sending real-time text alerts with IFTTT

Sourcing airfare pricing data

Fortunately, sourcing airfare data is somewhat easier than real estate data. There are a number of providers of this data, as well as paid and unpaid APIs. One challenging aspect of retrieving the data is that it requires a number of web requests. In the previous edition of this book, we outlined how to scrape data from Google's **Flight Explorer** page. This was the ideal presentation for seeing weeks of pricing data on one page for multiple cities.

Unfortunately, that page has now been removed, and Google now provides a more typical search interface that requires the user to input the departure city, the destination city, start date, and end date. One fortunate feature that remains is the ability to input an entire region rather than a specific city. We'll make use of this in our scraping. An example of this can be seen in the following screenshot:



As you can see, we have input New York as our departure city and simply put Asia as our destination. This returns pricing for all the top cities in Asia (as well as the Middle East, for some reason). This is great news as we want to capture as many prices in one web request as possible.

While the interface still has some features that make it easier to scrape this data, we'll need to use some more advanced techniques than we've used in the past. We'll discuss that next.

Retrieving fare data with advanced web scraping

In previous chapters, we've seen how to use the `Requests` library to retrieve web pages. As I've said before, it is a fantastic tool, but unfortunately, it won't work for us here. The page we want to scrape is entirely AJAX-based. **Asynchronous JavaScript (AJAX)** is a method for retrieving data from a server without having to reload the page. What this means for us is that we'll need to use a browser to retrieve the data. While that might sound like it would require an enormous amount of overhead, there are two libraries that, when used together, make it a lightweight task.

The two libraries are Selenium and ChromeDriver. Selenium is a powerful tool for automating web browsers, and ChromeDriver is a browser. Why use ChromeDriver rather than Firefox or Chrome itself? ChromeDriver is what's known as a **headless browser**. This means it has no user interface. This keeps it lean, making it ideal for what we're trying to do.



To install ChromeDriver, you can download the binaries or source from <https://sites.google.com/a/chromium.org/chromedriver/downloads>. As for Selenium, it can be pip installed.

We'll also need another library called `BeautifulSoup` to parse the data from the page. If you don't have that installed, you should `pip install` that now as well.

With that done, let's get started. We'll start out within the Jupyter Notebook. This works best for exploratory analysis. Later, when we've completed our exploration, we'll move on to working in a text editor for the code we want to deploy. This is done in following steps:

1. First, we import our routine libraries, as shown in the following code snippet:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
%matplotlib inline
```

2. Next, make sure you have installed `BeautifulSoup` and `Selenium`, and downloaded `ChromeDriver`, as mentioned previously. We'll import those now in a new cell:

```
from bs4 import BeautifulSoup
from selenium import webdriver

# replace this with the path of where you downloaded chromedriver
chromedriver_path = "/Users/alexcombs/Downloads/chromedriver"

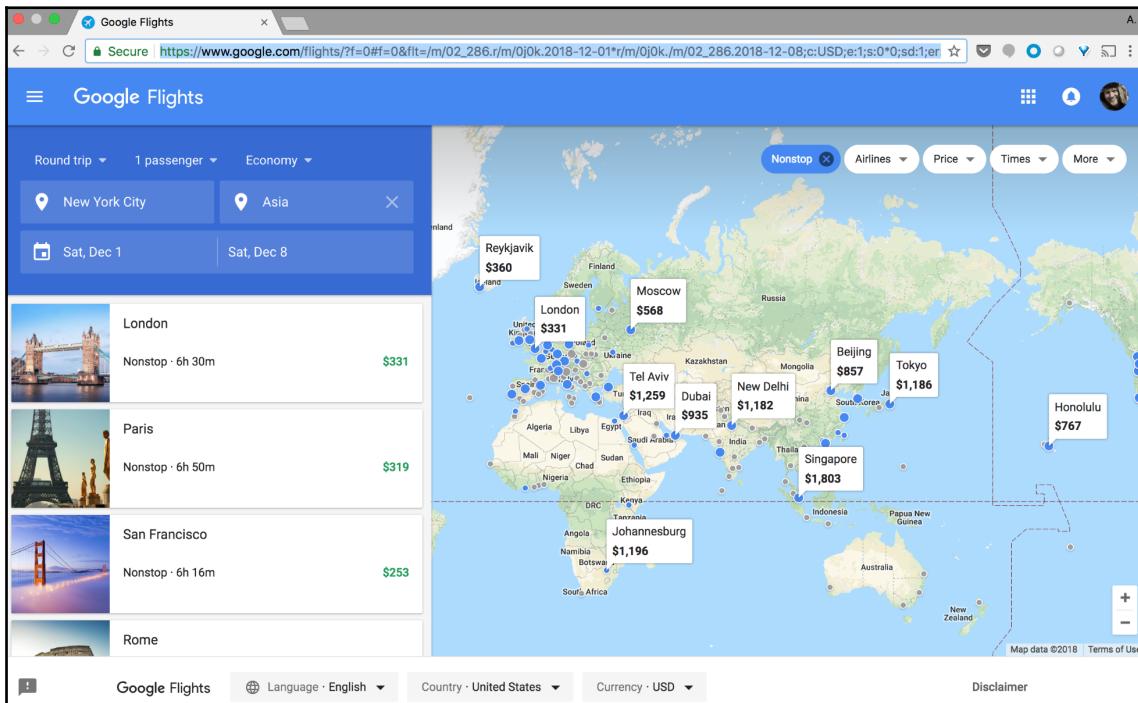
browser = webdriver.Chrome(chromedriver_path)
```

Notice that I have referenced the path on my machine where I have downloaded `ChromeDriver`. Note that you will have to replace that line with the path on your own machine.

Creating a link

Now, remarkably, we have everything we need to begin our airline fare scraping, with one exception: we need the URL. For this exercise, I'm going to focus on flights leaving from NYC and flying to Europe. Since we don't want to pull down massive quantities of data and risk being blocked, we are going to just pull data for non-stop flights that depart on Saturday and return on the following Saturday. You are, of course, free to change this to whatever fares you'd like to target, but we'll use this for our sample project.

The next step is to fill out the form in **Google Flights**. Make sure to choose a future date. Once you have input your data and hit **Search**, copy the URL string from your browser bar, as seen in the following screenshot:



The URL I copied is for flights that depart on 2018-12-01 and return on 2018-12-08. Those dates can be seen in the search string. If you choose different dates, you should see those reflected in the string you copy. Let's code this now:

1. Let's input that string and save it as the variable `sats`, as seen in the following block of code:

```
sats =  
'https://www.google.com/flights/f=0#f=0&flt=/m/02_286.r/m/02j9z.201  
8-12-01*r/m/02j9z./m/02_286.2018-12-08;c:USD;e:1;s:0*1;sd:1;t:e'
```

2. Next, we'll test that we can successfully retrieve the content that we see on the page. We'll test that with the following line of code, which utilizes `selenium`:

```
browser.get(sats)
```

3. That one line of code was all we needed to retrieve the page. We can validate that this was successful with a couple of additional lines of code.
4. First, let's check the title of the page:

```
browser.title
```

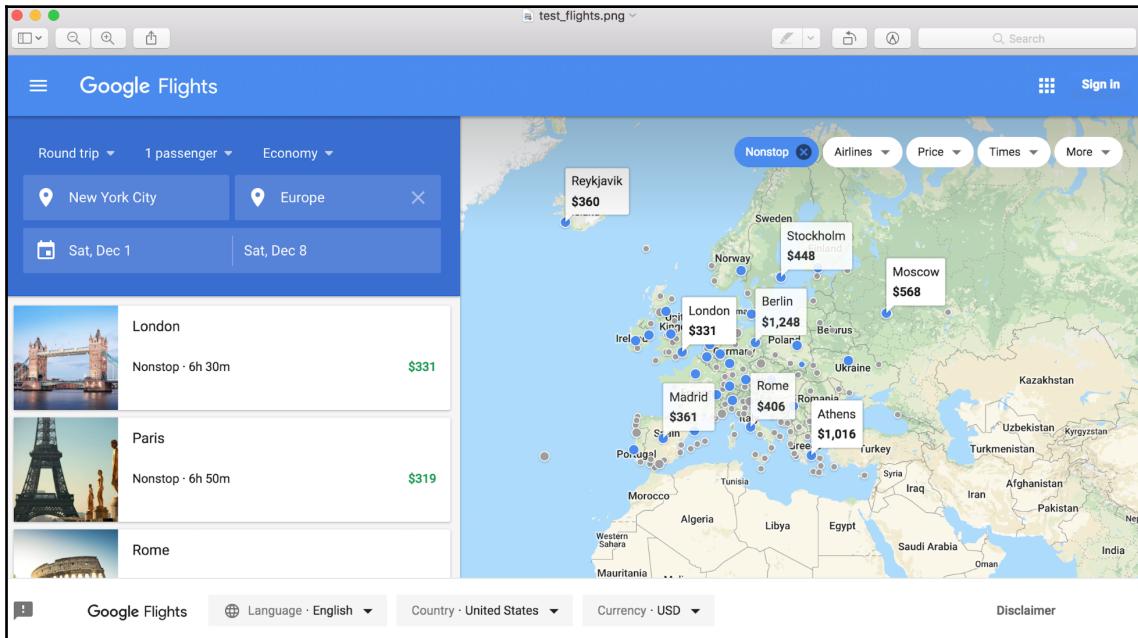
The resulting output can be seen as follows:

`'Google Flights'`

It looks like we were able to get the correct page. Let's now check to see whether we captured everything we were seeking. We can do that by taking a screenshot of the page. We do that with the following line of code:

```
browser.save_screenshot('/Users/alexcombs/Desktop/test_flights.png')
```

Again, the path I used to save the screenshot was based on my machine; you will need to reference a path on your own machine. As you should see based on the following output, we were able to successfully get all the content of the page:



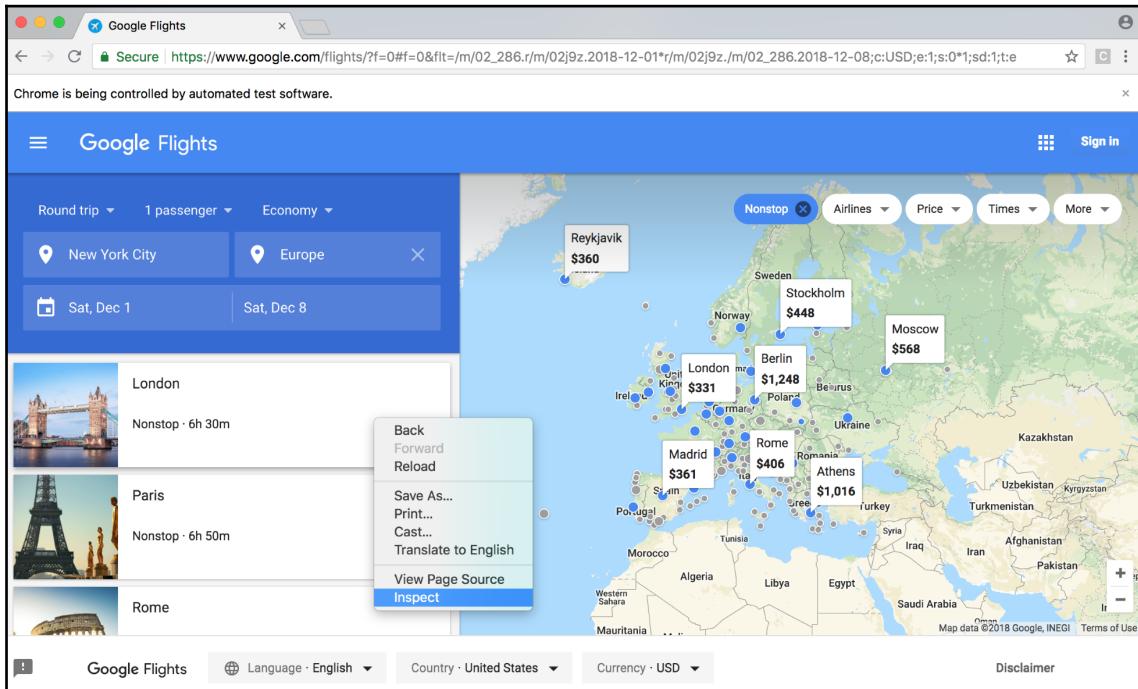
Since we appear to have all the page data we were seeking, we will now move on to how to pull individual data points from the page. To do that, first, we'll need to learn about the **Document Object Model (DOM)**.

Parsing the DOM to extract pricing data

The DOM is the collection of elements that comprise a web page. It includes HTML tags such as `body` and `div`, as well as the classes and IDs embedded within these tags.

Let's take a look at the DOM for our Google page:

1. To see it, right-click on the page and click on **Inspect**. This should be the same for Firefox or Chrome. This will open the developer tab that allows you to see the page source information, as demonstrated in the following screenshot:



- Once this is open, choose the element selector in the upper left-hand corner, and click on an element to jump to that element in the page source code:

Chrome is being controlled by automated test software.

Google Flights

Round trip ▾ 1 passenger ▾ Economy ▾

New York City ▾ Europe

Sat, Dec 1 Sat, Dec 8

 London Nonstop • 6h 30m \$331

Paris

Nonstop x Airlines ▾ Price ▾ Times ▾ More ▾

Copenhagen
Dublin \$483
London \$331
Amsterdam \$353
Berlin \$1,248
Lithuania
Warsaw \$940
Belarus
Copenhagen
Dublin \$483
London \$331
Amsterdam \$353
Berlin \$1,248
Lithuania
Warsaw \$940
Belarus

Dublin \$483
London \$331
Amsterdam \$353
Berlin \$1,248
Lithuania
Warsaw \$940
Belarus

Map data ©2018 Google, INEGI, ORION-ME Terms of use

3. The element that we are concerned with is the box that contains the flight information. This can be seen in the following screenshot:

The screenshot shows the Google Flights interface. On the left, there's a search bar with 'New York City' as the origin and 'Europe' as the destination. Below it, travel dates are set from 'Sat, Dec 1' to 'Sat, Dec 8'. A flight summary for a nonstop 6h 30m trip to London is displayed, showing a price of \$331. The main area is a map of Europe with flight prices overlaid on cities: London (\$331), Berlin (\$1,248), Madrid (\$361), Rome (\$406), Athens (\$1,016), Tel Aviv (\$1,075), and Baku (\$828). The right side of the map shows Russia and Central Asia. At the bottom, the developer tools' Elements tab is active, highlighting a specific `<div>` element with the class `sssAp3xR0A__info-container`. This element contains flight details for London. The computed styles for this class include `border-box`, `flex-direction: column`, `justify-content: space-between`, and `padding: 15px`.

If you look closely at the element, you will notice that it is an element called a `div`. This `div` has an attribute called `class`. The `class` is a long string of random numbers and letters in this `class`, but you will also notice that it contains the string `info-container`. We can use this information to retrieve all the `div` elements that have flight information for each city. We'll do that in a minute, but for now, let's discuss the parsing process.

Parsing

To begin our parsing, we will need to use the library we mentioned earlier called `BeautifulSoup`. We imported it earlier, so now we just need to pass the page source into `BeautifulSoup`. We do that by means of the following code:

```
soup = BeautifulSoup(browser.page_source, "html5lib")
```

Notice that the browser object contains a `page_source` attribute. That is all the HTML we retrieved with our `get` request earlier. The other parameter passed into `BeautifulSoup` is the parsing library it will use. Here, we will stick with `html5lib`.

Now, once the content of the page has been passed to `BeautifulSoup`, we want to start to extract the elements of interest. That's where the `div` elements with the `info-container` class come in. We are going to retrieve those. Each one corresponds to a single city.

Let's retrieve them, but we'll just look at the first one:

```
cards = soup.select('div[class*=info-container]')
cards[0]
```

The output for the preceding code is shown as follows:

```
<div class="sssApT3XrOA__info-container"><h3 class="sssApT3XrOA__title aria-punctuation-after" jstcache="5723">London</h3><div jstcache="5724" style="display:none"></div><destination-price-summary jstcache="5725"><div class="tD5kazVERD1__first-flight-row tD5kazVERD1__flight-row" jsan="7.tD5kazVERD1__first-flight-row,t-z-b82lC7-Tw,7.tD5kazVERD1__flight-row,21.jssc" jstcache="5732"><span class="gws-flights_ellipsis" jstcache="5736">Nonstop</span><span class="tD5kazVERD1__timing" jsan="7.tD5kazVERD1__timing,t-9OG0n4dH-E8" jstcache="5737" style=""><span aria-hidden="true" style="display:none"></span><span jstcache="5745">6h 30m</span></span><span jstcache="5738" style="display:none"></span></span><span class="tD5kazVERD1__price aria-punctuation-after tD5kazVERD1__cheapest-price" jsan="7.tD5kazVERD1__price,7.aria-punctuation-after,7.tD5kazVERD1__cheapest-price" jstcache="5739"><jsl jstcache="5740" style="">$331</jsl><jsl jstcache="5741" style="display:none"></jsl></span></div><div class="tD5kazVERD1__flight-row" jsan="t-z-b82lC7-Tw,7.tD5kazVERD1__flight-row,21.jssc" jstcache="5733"><span class="gws-flights_ellipsis" jstcache="5736">Connecting</span><span jstcache="5737" style="display:none"></span><span jstcache="5738" style="display:none"></span><span class="tD5kazVERD1__price aria-punctuation-after" jsan="7.tD5kazVERD1__price,7.aria-punctuation-after" jstcache="5739"><jsl jstcache="5740" style="">$440</jsl><jsl jstcache="5741" style="display:none"></jsl></span></div></destination-price-summary></div>
```

In the preceding code, we used the `select` method on our `soup` object. The `select` method allows us to use CSS selectors to reference the elements of interest. Here, we have specified that we want divs that have a `class` attribute that contains somewhere within the class name the string `info-container`. There is excellent documentation on `BeautifulSoup` that explains these CSS selectors and other methods, and is available at <https://www.crummy.com/software/BeautifulSoup/bs4/doc/#css-selectors>.

Looking at the preceding output, notice that buried deep within the markup, there is the name of the destination city (London) and the fare price (\$440). Since we just want the data and not all the surrounding markup, we'll need to create code to iterate over each `info`-container divs and pull out the city and the fare:

```
for card in cards:  
    print(card.select('h3')[0].text)  
    print(card.select('span[class*=price]')[0].text)  
    print('\n')
```

The preceding code results in the following output:

London	\$331
Paris	\$319
Rome	\$406
Reykjavik	\$360
Barcelona	\$370

Since it looks as if we were able to successfully retrieve the fares for each city, let's now move on to constructing a full scrape and parse for a large number of fares.

We are now going to attempt to retrieve the lowest cost, non-stop fares from NYC to Europe for a 26-week period. I'm using a start date of December 01, 2018, but obviously, if you are reading this after that date, make sure to adjust your dates accordingly.

The first thing we'll need is to bring in some additional imports. We do that in the following code:

```
from datetime import date, timedelta  
from time import sleep
```

Next, we'll construct the remainder of our scraping code:

```
start_sat = '2018-12-01'
end_sat = '2018-12-08'

start_sat_date = datetime.strptime(start_sat, '%Y-%m-%d')
end_sat_date = datetime.strptime(end_sat, '%Y-%m-%d')

fare_dict = {}

for i in range(26):
    sat_start = str(start_sat_date).split()[0]
    sat_end = str(end_sat_date).split()[0]
    fare_dict.update({sat_start: {}})
    sats =
"https://www.google.com/flights/?f=0#f=0&flt=/m/02_286.r/m/02j9z." + \
    sat_start + "*r/m/02j9z./m/02_286." + \
    sat_end + ";c:USD;e:1;s:0*1;sd:1;t:e"
    sleep(np.random.randint(3, 7))

    browser.get(sats)
    soup = BeautifulSoup(browser.page_source, "html5lib")
    cards = soup.select('div[class*=info-container]')
    for card in cards:
        city = card.select('h3')[0].text
        fare = card.select('span[class*=price]')[0].text
        fare_dict[sat_start] = {**fare_dict[sat_start], **{city: fare}}
    start_sat_date = start_sat_date + timedelta(days=7)
    end_sat_date = end_sat_date + timedelta(days=7)
```

That's a fair amount of code, so we'll unpack what is going on line by line. The first two lines just create our start and end dates that we'll use. The next two lines convert those date strings into `datetime` objects. This will be used later on when we want to add a week to each using `timedelta`. The last line before the `for` loop simply creates a dictionary that will hold our parsed data.

The next line begins a `for` loop. Inside this loop that will run 26 iterations, we convert our `datetime` object back into a string so that we can pass it into the URL that we will call with our `browser` object. Also, notice that on each iteration we populate our `fare` dictionary with the start date. We then create our URL using the date strings we created.

Next, we insert a random pause using the `numpy.random` function and the Python `sleep` function. This is simply to prevent us from appearing to be a bot and overtaxing the site.

We then retrieve the page with our browser object, pass it into BeautifulSoup for parsing, select the info-container divs, and then parse and update our fare dictionary. Finally, we add one week to our start and end dates so that the next iteration goes one week forward in time.

Now, let's look at the data in our fare dictionary:

```
fare_dict
```

The preceding code results in the following output:

```
{'2018-12-01': {'London': '$331',
 'Paris': '$319',
 'Rome': '$406',
 'Reykjavik': '$360',
 'Barcelona': '$370',
 'Amsterdam': '$353',
 'Madrid': '$361',
 'Dublin': '$483',
 'Athens': '$1,016',
 'Milan': '$557',
 'Lisbon': '$669',
 'Frankfurt': '$722',
 'Moscow': '$568',
 'Berlin': '$1,248',
 'Istanbul': '$889',
 'Copenhagen': '$575',
 'Zürich': '$520',
 'Munich': '$1,017',
 'Manchester': '$888',
```

As you can see, we have a dictionary with date as the primary key, and then sub dictionaries with city/fare pairings.

Now, let's dive into one city to examine the data. We'll begin with Berlin:

```
city_key = 'Berlin'  
for key in fare_dict:  
    print(key, fare_dict[key][city_key])
```

The preceding code results in the following output:

2018-12-01 \$1,248
2018-12-08 \$1,248
2018-12-15 \$1,248
2018-12-22 \$1,643
2018-12-29 \$1,408
2019-01-05 \$1,408
2019-01-12 \$1,323
2019-01-19 \$1,248
2019-01-26 \$1,248
2019-02-02 \$1,248
2019-02-09 \$1,248
2019-02-16 \$1,248
2019-02-23 \$1,248
2019-03-02 \$1,248
2019-03-09 \$1,248
2019-03-16 \$1,248
2019-03-23 \$1,248
2019-03-30 \$1,248
2019-04-06 \$1,308

One thing we notice right away is that we'll need to clean up the airfares so that we can work with them. We'll need to remove the dollar sign and the commas and convert them into integers. We do that in the following code:

```
city_dict = {}  
for k,v in fare_dict.items():  
    city_dict.update({k:int(v[city_key].replace(',', '').replace('$', ''))})
```

The preceding code results in the following output:

```
{'2018-12-01': 1248,
 '2018-12-08': 1248,
 '2018-12-15': 1248,
 '2018-12-22': 1643,
 '2018-12-29': 1408,
 '2019-01-05': 1408,
 '2019-01-12': 1323,
 '2019-01-19': 1248,
 '2019-01-26': 1248,
 '2019-02-02': 1248,
 '2019-02-09': 1248,
 '2019-02-16': 1248,
 '2019-02-23': 1248,
 '2019-03-02': 1248,
 '2019-03-09': 1248,
 '2019-03-16': 1248,
 '2019-03-23': 1248,
 '2019-03-30': 1248,
 '2019-04-06': 1308,
```

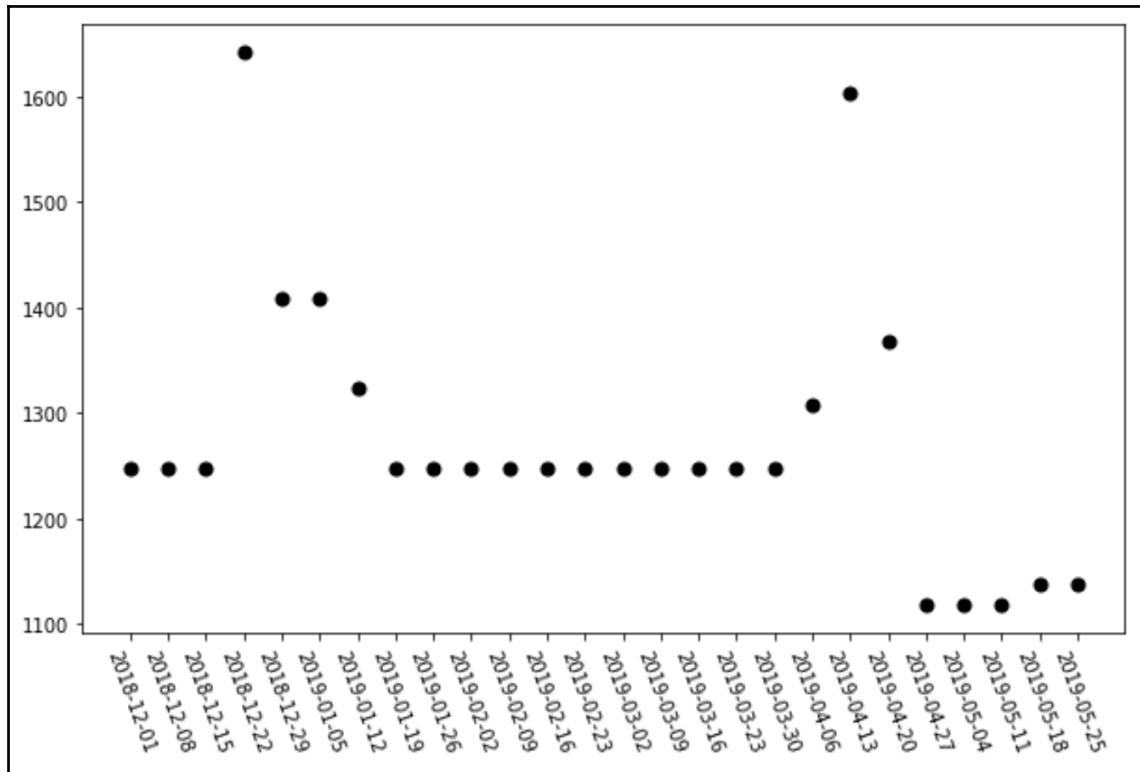
Remember, the output shown in the preceding code is only for Berlin, as we are just examining one city at the moment.

Now, let's plot that data:

```
prices = [int(x) for x in city_dict.values()]
dates = city_dict.keys()

fig,ax = plt.subplots(figsize=(10,6))
plt.scatter(dates, prices, color='black', s=50)
ax.set_xticklabels(dates, rotation=-70);
```

The preceding code generates the following output:



Notice that we have 26 consecutive weeks of data, in this case, for non-stop flights from NYC to Berlin leaving on Saturday and returning the following Saturday. There appears to be a fair amount of variation in these fares. Just eyeballing the data, it appears that there might be two outliers on the high end toward the beginning of the period and the end.

Now, let's take a look at another city. To do this, we simply need to return to our code and change the `city_key` variable. We can then rerun the cells below it. We'll do that in the following code:

```
city_key = 'Milan'  
for key in fare_dict:  
    print(key, fare_dict[key][city_key])
```

This results in the following output:

2018-12-01	\$557
2018-12-08	\$557
2018-12-15	\$557
2018-12-22	\$1,072
2018-12-29	\$1,221
2019-01-05	\$1,003
2019-01-12	\$1,003
2019-01-19	\$938
2019-01-26	\$938
2019-02-02	\$938
2019-02-09	\$938
2019-02-16	\$938
2019-02-23	\$938
2019-03-02	\$938
2019-03-09	\$938
2019-03-16	\$938
2019-03-23	\$938
2019-03-30	\$938

We'll need to remove the dollar sign and the commas and convert them into integers. We do that in the following code:

```
city_dict = {}
for k,v in fare_dict.items():
    city_dict.update({k:int(v[city_key].replace(',', '').split('$')[1])})

city_dict
```

The preceding code results in the following output:

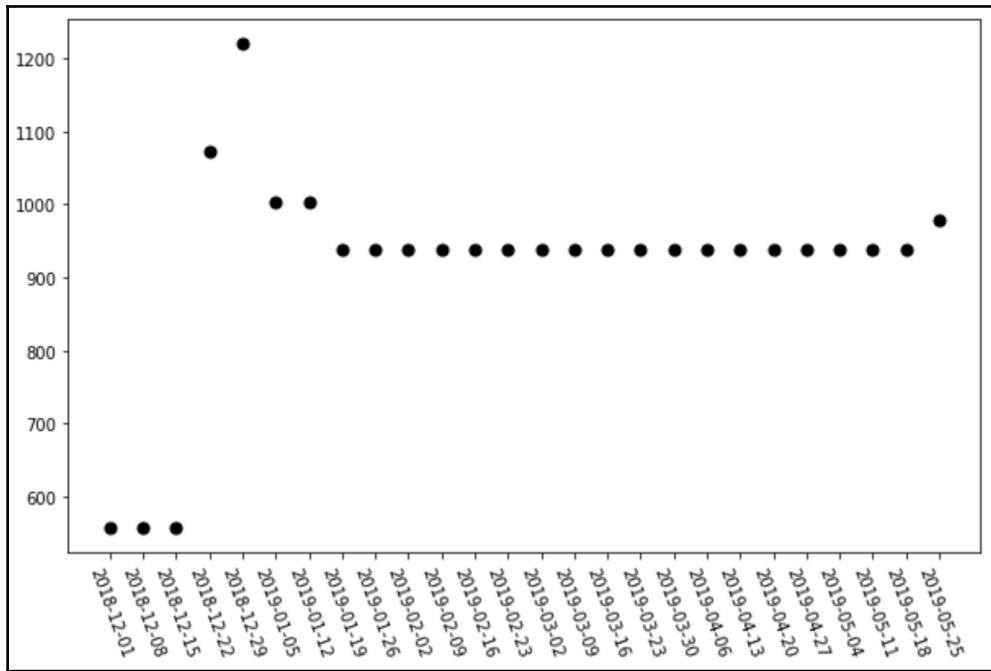
```
{'2018-12-01': 557,
 '2018-12-08': 557,
 '2018-12-15': 557,
 '2018-12-22': 1072,
 '2018-12-29': 1221,
 '2019-01-05': 1003,
 '2019-01-12': 1003,
 '2019-01-19': 938,
 '2019-01-26': 938,
 '2019-02-02': 938,
 '2019-02-09': 938,
 '2019-02-16': 938,
 '2019-02-23': 938,
 '2019-03-02': 938,
 '2019-03-09': 938,
 '2019-03-16': 938,
 '2019-03-23': 938,
 '2019-03-30': 938,
```

Now, let's plot that data:

```
prices = [int(x) for x in city_dict.values()]
dates = city_dict.keys()

fig,ax = plt.subplots(figsize=(10,6))
plt.scatter(dates, prices, color='black', s=50)
ax.set_xticklabels(dates, rotation=-70);
```

The preceding code results in the following output:



Here, we can see even wider variations, with fares ranging from under \$600 to over \$1,200. Those cheap fares on the left are exactly the type of fares we'd like to know about. We are going to want to create an outlier detection system that will tell us about these bargain fares. We'll move on and discuss that now.

Identifying outlier fares with anomaly detection techniques

There are various rigorous definitions of outliers, but for our purposes, an **outlier** is any extreme value that is far from the other observations in the dataset. There are numerous techniques, both parametric and non-parametric, that are used to identify outliers; example algorithms include **density-based spatial clustering of applications with noise (DBSCAN)**, isolation forests, and Grubbs' Test. Typically, the type of data determines the type of algorithm that is used. For example, some algorithms do better on multivariate data than univariate data. Here, we are dealing with univariate time-series data, so we'll want to choose an algorithm that handles that well.



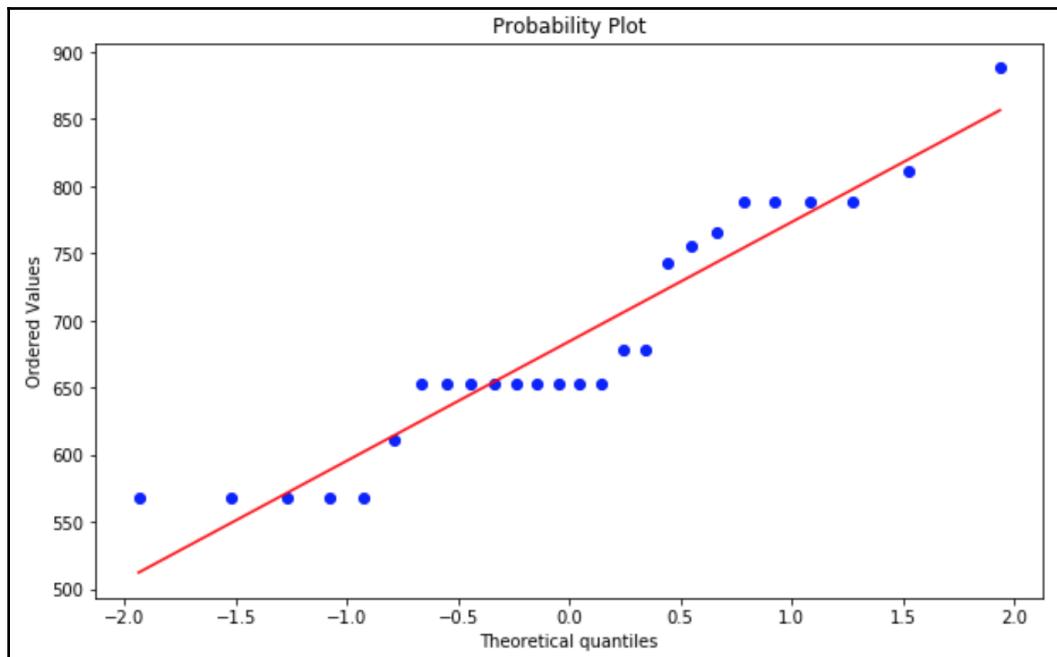
If you aren't familiar with the term *time series*, it simply means data that is recorded at regular intervals, such as the daily closing price of a stock.

The algorithm that we are going to use for our data is called **Generalized Extreme Studentized Deviate (Generalized ESD)** test for outliers. This algorithm is well suited for our data, since it is univariate and approximately normal.

There are several tests we can use to ensure that our data is approximately normally distributed, but we can also visually inspect our data for normality using a normal probability plot. We'll do that now for Moscow city data using some functionality from the SciPy library:

```
from scipy import stats
fig, ax = plt.subplots(figsize=(10, 6))
stats.probplot(list(city_dict.values()), plot=plt)
plt.show()
```

The preceding code generates the following output:



When assessing a **normal probability** or **quantile-quantile (Q-Q) plot**, we are looking for the data to be as close to the straight line as possible to reveal normality. Data that veers off in one direction or another, or with a strong S shape, argues against normal data. Here, we have a fairly low number of data points, and those that we do have are fairly balanced around the diagonal. If we had more data, it is likely that we would more closely approximate the diagonal. This should work well enough for our purposes.

We'll now move on to our outlier detection code. We are going to be utilizing another library for this called `PyAstronomy`. If you don't have it, it can easily be pip installed.

Let's look at the code:

```
from PyAstronomy import pyasl

r = pyasl.generalizedESD(prices, 3, 0.025, fullOutput=True)

print('Total Outliers:', r[0])

out_dates = {}
for i in sorted(r[1]):
    out_dates.update({list(dates)[i]: list(prices)[i]})

print('Outlier Dates', out_dates.keys(), '\n')
print('      R          Lambda')

for i in range(len(r[2])):
    print('%2d  %8.5f  %8.5f' % ((i+1), r[2][i], r[3][i]))

fig, ax = plt.subplots(figsize=(10, 6))
plt.scatter(dates, prices, color='black', s=50)
ax.set_xticklabels(dates, rotation=-70);

for i in range(r[0]):
    plt.plot(r[1][i], prices[r[1][i]], 'rp')
```

Let's discuss what the preceding code does. The first line is simply our import. Following that, we implement our generalized ESD algorithm. The parameters are our fare prices, then the maximum number of outliers (here, we chose 3), the significance level (alpha, at 0.025), and finally a Boolean to specify that we want the full output. With respect to the significance level, the lower the value, the less sensitive the algorithm and the fewer false positives will be generated.

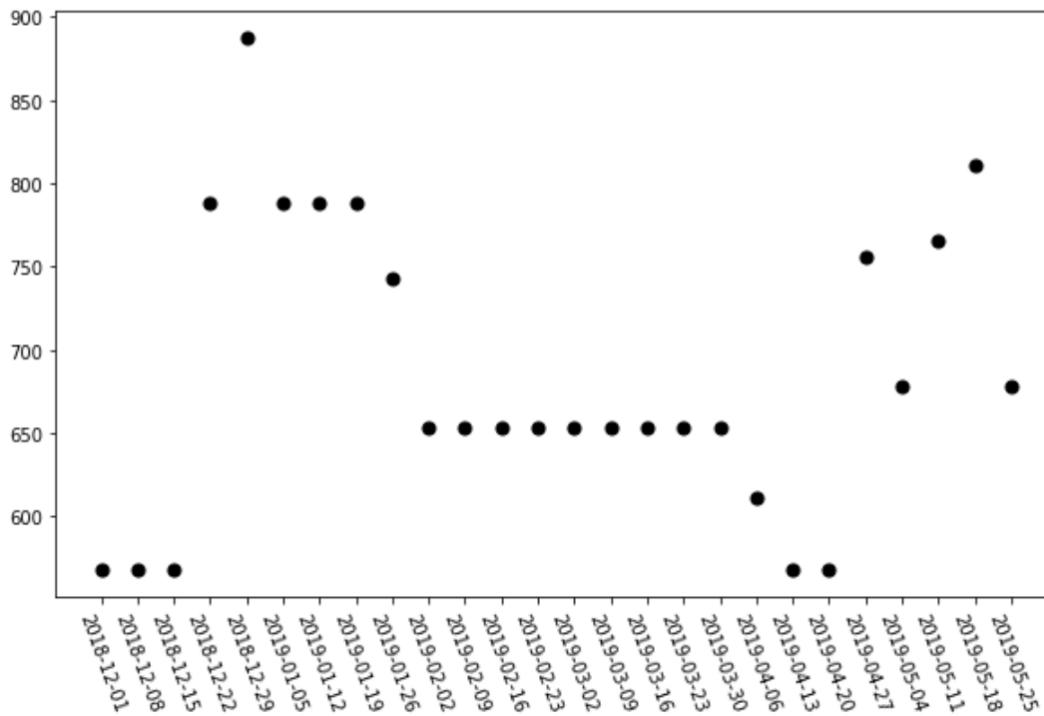
The next two lines simply print out data related to the `R` and `Lambda` values. These are utilized in the determination of whether a data point is an outlier.

Finally, the remainder of the code is simply for generating the scatter plot and coloring those fares that are outliers red.

The preceding code generates the following output:

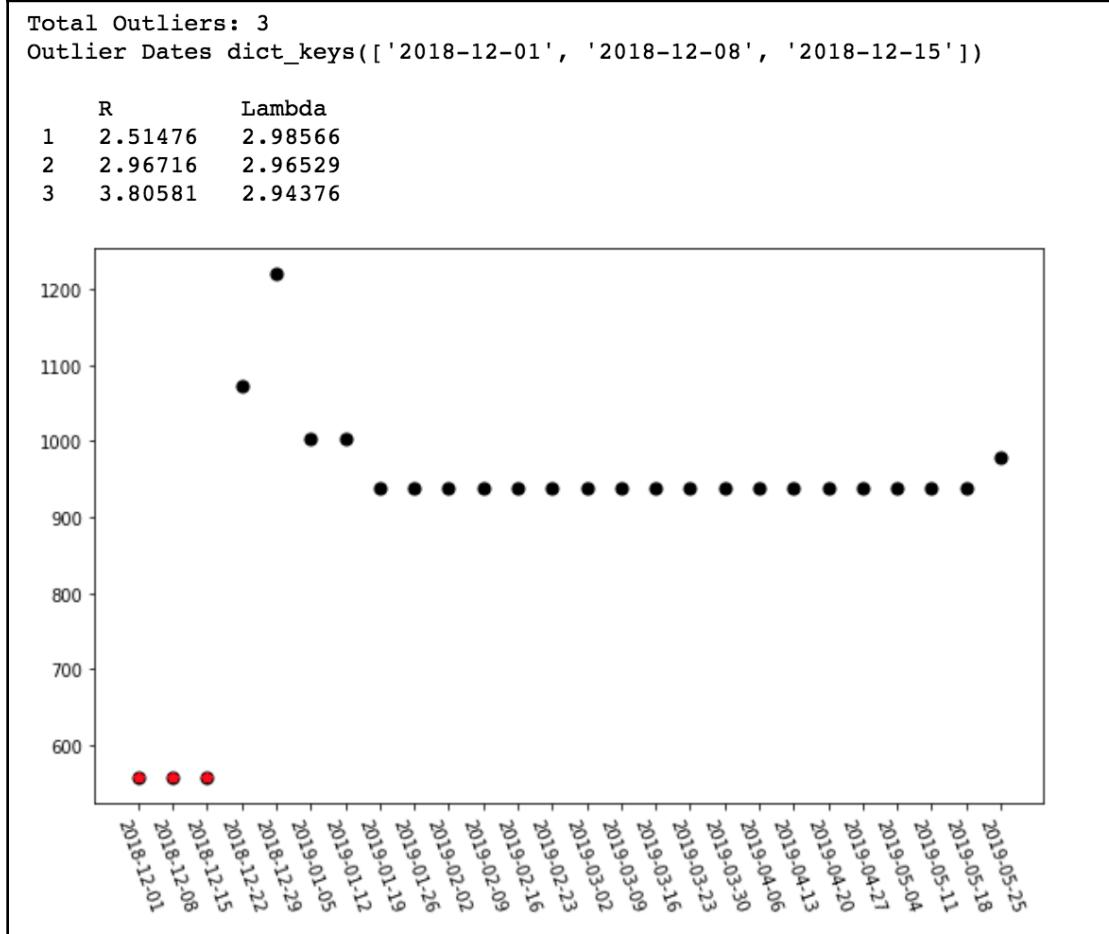
```
Total Outliers: 0
Outlier Dates dict_keys([])
```

	R	Lambda
1	2.33228	2.98566
2	1.71007	2.96529
3	1.55678	2.94376



Again, this data is for Moscow. Make sure you changed your `city_key` variable to reflect that to ensure you get that data. Notice that despite all the variations, there are no outliers in the data.

Now, let's run it for Milan as well. We'll go back up and change our `city_key` variable and run the cells below that to update everything, as demonstrated in the following diagram:

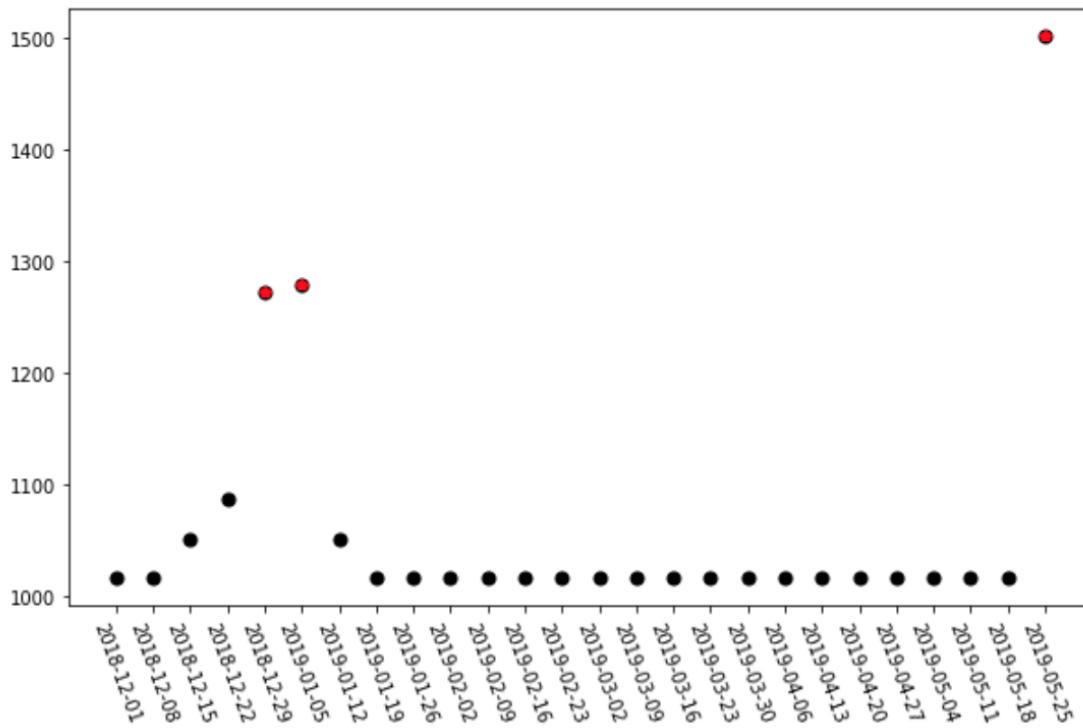


Notice that this time, we have three outliers, and these are fares that are under \$600 when the mean fare looks to be over \$900, so this looks like a win for us.

Let's try another city. This time, we'll look at Athens by updating the `city_key` variable and running the subsequent cells:

```
Total Outliers: 3
Outlier Dates dict_keys(['2018-12-29', '2019-01-05', '2019-05-25'])

      R      Lambda
1 3.93883  2.98566
2 3.35758  2.96529
3 4.55316  2.94376
```



Notice that again, we have three outliers, but that this time, they are extreme fares to the upside. Since we are only interested in getting alerts for cheap fares, we can build in a mechanism to only alert us when the fare outlier is less than the mean fare.

Now, we'll now create some code to handle this element:

```
city_mean = np.mean(list(city_dict.values()))  
  
for k,v in out_dates.items():  
    if v < city_mean:  
        print('Alert for', city_key + '!')  
        print('Fare: $' + str(v), 'on', k)  
        print('\n')
```

When we run the code for Athens, it will generate no output. When run for Milan, it generates the following output:

```
Alert for Milan!  
Fare: $557 on 2018-12-01  
  
Alert for Milan!  
Fare: $557 on 2018-12-08  
  
Alert for Milan!  
Fare: $557 on 2018-12-15
```

So, now, we have created a system to scrape the data, parse it, and identify the outliers. Let's move on and create a fully-fledged application that can alert us in real time.

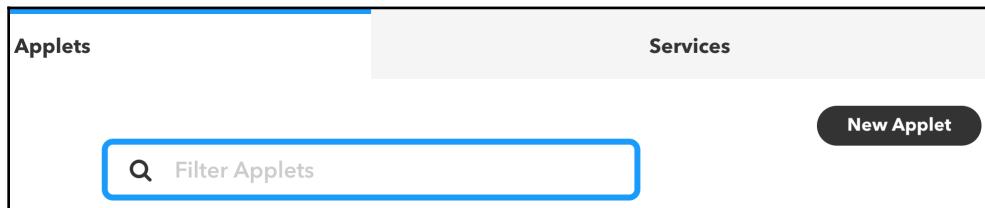


Keep in mind that we just did a very preliminary analysis on our outlier detection model. In the real world, it would likely take a much more thorough series of tests to identify whether we had selected workable parameters for our model.

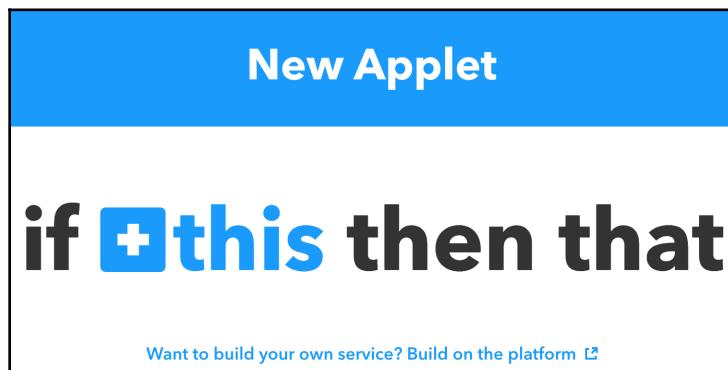
Sending real-time alerts using IFTTT

To have a chance at getting these cheap fares, we're going to need to know in nearly real time when they happen. To accomplish this, we'll use a service called **If This Then That** (IFTTT). This free service allows you to connect a huge number of services with a series of triggers and actions. Want to save as many pictures as you like on Instagram to your iPhone photos? Want to get an email every time a particular person tweets? Want your Facebook updates posted to Twitter? IFTTT can do all of this. Go through the following steps:

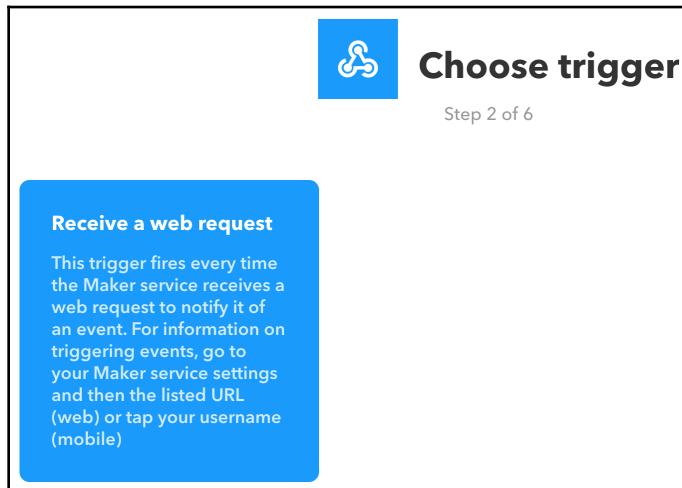
1. The first step is to sign up for an account at <http://www.ifttt.com>.
2. Once you've done that, you'll need to sign up for the Maker channel, https://ifttt.com/maker_webhooks, and the SMS channel, <https://ifttt.com/sms>.
3. Maker allows you to create IFTTT recipes by sending and receiving HTTP requests.
4. Once you've created an account and activated the Maker and SMS channels, click **My Applets** from the home page, and then click **New Applet**:



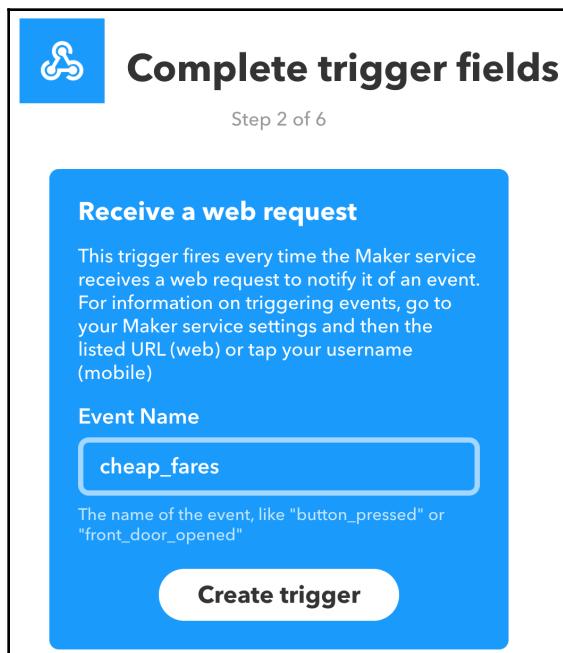
5. Then, click on **this**, as shown in the following screenshot:



6. Then, search for `webhooks` and click **Receive a web request**:



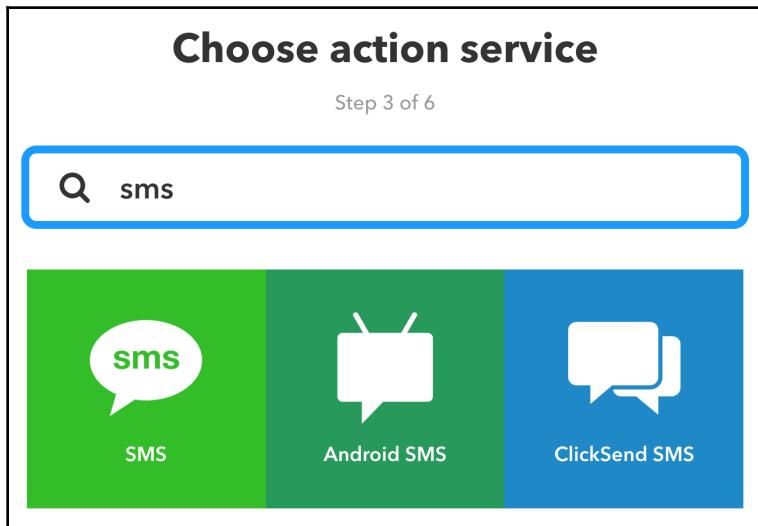
7. Then, we'll create an event called `cheap_fares`:



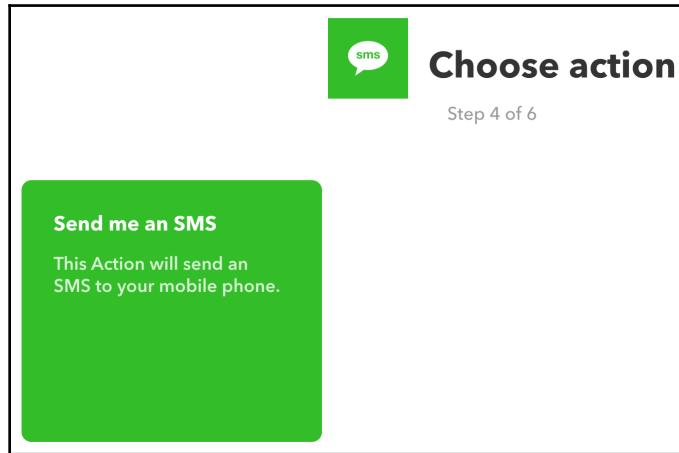
- Once you fill out the event name, click **Create trigger**. Next, we'll set up the **+that**:



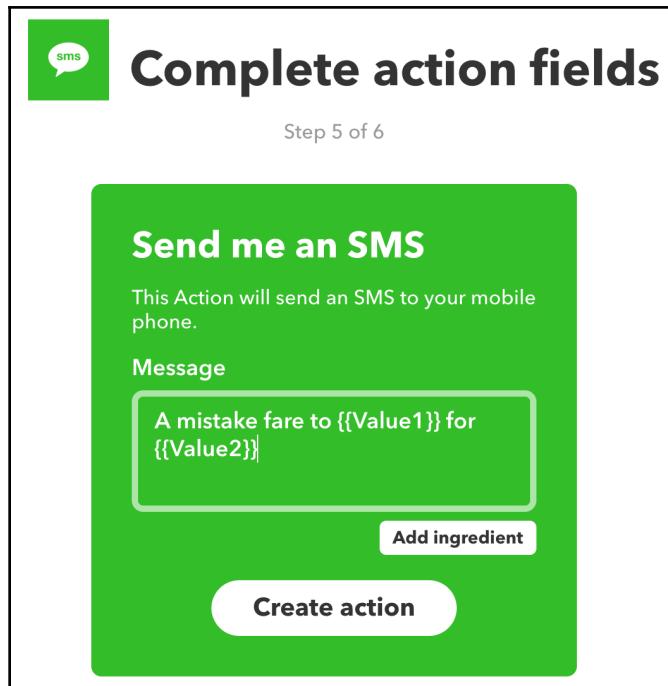
- Click **that**, and then search for **SMS** and select it:



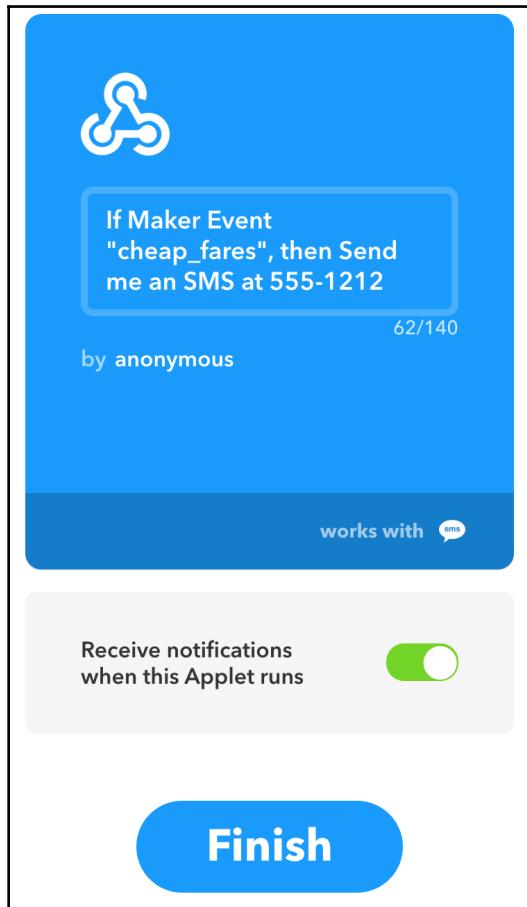
Then, choose **Send me an SMS**:



10. After that, we'll customize our message:



Once this is complete, click on **Finish** to wrap-up the setup:



To test the setup, go to http://www.ifttt.com/maker_webhooks and click on **Settings**. You should see your account information that includes a URL with a secret key. Copy and paste that URL into your browser. It should have a form with your secret key and a place to fill in the values that correspond to your city and price.

Fill in **cheap_fares** for event, and place the city and fare into **value1** and **value2**, respectively:

The screenshot shows a web interface for IFTTT. At the top, there's a blue logo icon. Below it, the text "Your key is: **MY_SECRET_KEY**". Underneath that, a link "Back to service". A section titled "To trigger an Event" contains instructions: "Make a POST or GET web request to:" followed by a URL template: `https://maker.ifttt.com/trigger/cheap_fares/with/key/MY_SECRET_KEY`. Below this, it says "With an optional JSON body of:" and shows a JSON template: `{ "value1" : "Milan", "value2" : "$500", "value3" : "[]" }`. A note at the bottom states: "The data is completely optional, and you can also pass value1, value2, and value3 as query parameters or form variables. This content will be passed on to the Action in your Recipe."

Finally, click **Test It**, and you should receive a text message in just a few seconds.

Now that we have all the pieces in place, it's time to pull it all together into a single script that will monitor fares 24/7.

Putting it all together

Up until this point, we've worked within the Jupyter Notebook, but now, in order to deploy our app, we'll move on to working in a text editor. The notebook is excellent for exploratory analysis and visualization, but running a background job is best done within a simple .py file. So, let's get started.

We'll begin with our imports. You may need to `pip install` a few of these if you don't already have them installed:

```
import sys
import sys
import numpy as np
from bs4 import BeautifulSoup
from selenium import webdriver
import requests
import scipy
from PyAstronomy import pyasl
```

```
from datetime import date, timedelta, datetime
import time
from time import sleep
import schedule
```

Next, we'll create a function that pulls down the data and runs our algorithm:

```
def check_flights():
    # replace this with the path of where you downloaded chromedriver
    chromedriver_path = "/Users/alexcombs/Downloads/chromedriver"

    browser = webdriver.Chrome(chromedriver_path)

    start_sat = sys.argv[2]
    end_sat = sys.argv[3]

    start_sat_date = datetime.strptime(start_sat, '%Y-%m-%d')
    end_sat_date = datetime.strptime(end_sat, '%Y-%m-%d')

    fare_dict = {}

    for i in range(26):
        sat_start = str(start_sat_date).split()[0]
        sat_end = str(end_sat_date).split()[0]
        fare_dict.update({sat_start: {}})
        sats =
"https://www.google.com/flights/?f=0#f=0&flt=/m/02_286.r/m/02j9z." + \
        sat_start + "*r/m/02j9z./m/02_286." + \
        sat_end + ";c:USD;e:1;s:0*1;sd:1;t:e"
        sleep(np.random.randint(10, 15))
        browser.get(sats)

        soup = BeautifulSoup(browser.page_source, "html5lib")
        cards = soup.select('div[class*=info-container]')
        for card in cards:
            city = card.select('h3')[0].text
            fare = card.select('span[class*=price]')[0].text
            fare_dict[sat_start] = {**fare_dict[sat_start], **{city: fare}}
        start_sat_date = start_sat_date + timedelta(days=7)
        end_sat_date = end_sat_date + timedelta(days=7)

    city_key = sys.argv[1]

    city_dict = {}
    for k,v in fare_dict.items():
        city_dict.update({k:int(v[city_key].replace(',', '').split('$')[1])})
```

```
prices = [int(x) for x in city_dict.values()]
dates = city_dict.keys()

r = pyasl.generalizedESD(prices, 3, 0.025, fullOutput=True)

print('Total Outliers:', r[0])

out_dates = {}
for i in sorted(r[1]):
    out_dates.update({list(dates)[i]: list(prices)[i]})

city_mean = np.mean(list(city_dict.values()))

for k,v in out_dates.items():
    if v < city_mean:
        requests.post('https://maker.ifttt.com/trigger/cheap_fares/with/key/bNHFwiZ
x0wMS7EnD425n3T', \
                      data={ "value1" : str(city_key), "value2" : str(v), "value3" :
"" })
        print('Alert for', city_key + '!')
        print('Fare: $' + str(v), 'on', k)
        print('\n')
    else:
        print(str(v) + ' is greater than ' + str(city_mean))
```

Finally, we'll include a scheduler. This will run our code every 60 minutes:

```
# set up the scheduler to run our code every 60 min
schedule.every(60).minutes.do(check_flights)

while 1:
    schedule.run_pending()
    time.sleep(1)
```

And that should do it. We can now save this as `fare_alerter.py`, and run it from the command line. You will need to pass in three arguments. The first is the city, the second is the start date, and the final one is the end date. An example is as follows:

```
python fare_alerter.py 'Milan' '2018-12-01' '2018-12-08'
```

The script will continue running and check fares every 60 minutes. If a mistake fare occurs, we'll be one of the first to know!

Summary

We've covered a lot of ground in this chapter. We've learned how to find the best airfare data on the web, how to work with the DOM to find the elements we want to parse, how to identify outliers, and finally, how to send text alerts from our code using web requests through IFTTT. While what we've covered here is for airfares, nearly everything we've done could be reused for any type of pricing you'd like to be alerted to.

If you do decide to use it for airfares though, I hope it provides you with many happy travels!

4

Forecast the IPO Market Using Logistic Regression

In the late 1990s, getting in on the right **Initial Public Offering (IPO)** was like winning the lottery. First-day returns for some technology companies were many times their initial offering price, and if you were lucky enough to get in on an allocation, you were in for a windfall. Here are a few of the top first-day performers from the period:

- VA Linux up 697%, 12/09/99
- Globe.com up 606%, 11/13/98
- Foundry Networks up 525%, 9/28/99

While the days of dotcom mania are far behind us, IPOs can still have outsized first-day returns. Here are just a few that rose substantially on their first day of trading in the past year:

- Bloom Energy up 67%
- Pinduoduo up 32%
- Tenable up 32%

As you can see, this is still a market worth paying attention to. In this chapter, we'll take a closer look at the IPO market. We'll see how we can use machine learning to help us decide which IPOs are worth a closer look and which ones we may want to take a pass on.

Here's what we'll cover in this chapter:

- The IPO market
- Data cleansing and feature engineering
- Binary classification with logistic regression
- Model evaluation
- Feature importance

The IPO market

Before we jump in and begin modeling, let's first discuss what an IPO, or initial public offering, is, and what research tells us about this market. After that, we'll discuss a number of strategies that we can apply.

What is an IPO?

An IPO is the process whereby a private company becomes a public company. Public offerings raise capital for the company and give the general public an opportunity to invest in the company by buying its shares.

Though there are variations in how this occurs, in a typical offering, a company enlists the help of one or more investment banks to underwrite their offering. This means that the banks make a guarantee to the company that they will purchase all of the shares being offered at the IPO price on the day of the IPO. The underwriters, of course, do not intend to keep all of the shares themselves. With the help of the offering company, they go on what's called a **roadshow** to drum up interest from institutional clients. These clients put in a **subscription** for the shares, which indicates their interest in buying shares on the day of the IPO. This is a non-binding contract, as the price of the offering is not finalized until the day of the IPO. The underwriter will then set the offer price, given the level of interest expressed.

What is interesting from our perspective is that research has consistently shown a systematic underpricing of IPOs. There are a number of theories as to why this happens, and why this level of underpricing seems to vary over time, but studies have shown that billions of dollars are left on the table every year.

In an IPO, **money left on the table**, is the difference between the offering price of shares and the first day's closing price.

One other point that should be mentioned before we move on is the difference between the offering price and the opening price. While you can occasionally get in on the deal through your broker and receive the IPO at its offering price, in nearly all instances, you, as a member of the general public, will have to purchase the IPO at the (typically higher) opening price. We'll build our models under this assumption.

Recent IPO market performance

Let's now take a look at the performance of the IPO market. We are going to pull down data from [IPoScoop.com](https://www.iposcoop.com), which is a service that provides ratings for upcoming IPOs. Go to <https://www.iposcoop.com/scoop-track-record-from-2000-to-present/> and click on the button at the bottom of the page to download the spreadsheet. We'll load this into pandas and run a number of visualizations using our Jupyter notebook.

Unfortunately, the data is in a format that makes it impossible to just read into pandas with the normal `.read_csv()` method. What we'll need to do is use a library that lets us read Excel files into Python lists and then perform some preprocessing to filter out those rows that aren't of interest, primarily, header rows, and some extraneous information. Follow the steps to set up notebook:

1. Let's now begin in our notebook by setting up the libraries we'll need:

```
import numpy as np
import pandas as pd
import xlrd
import matplotlib.pyplot as plt
%matplotlib inline
```

The `xlrd` library is what we'll be using to work with the Excel spreadsheet we downloaded earlier. If you don't have it installed already, it can be added to your Python distribution at the command line with `pip install xlrd`.

2. The next step is to load the workbook, as seen in the following block of code:

```
wb = xlrd.open_workbook('SCOOP-Rating-Performance.xls')
```

3. Now that we have loaded the entire Excel workbook, let's target the sheet we'll be working with, in this instance, the first one:

```
ws = wb.sheet_by_index(0)
```

4. Let's now check that we have the data we are expecting:

```
ws.nrows
```

5. The preceding line of code generates the following output:

3286

6. That number looks about right from comparing it against the spreadsheet, so let's now move on to incorporating the data row by row:

```
ipo_list = []
for i in range(36, ws.nrows):
    if isinstance(ws.row(i)[0].value, float):
        ipo_list.append([x.value for x in ws.row(i)])
    else:
        print(i, ws.row(i))
```

The preceding code generates the following output:

Let's talk about what happened in that code block. First, we create an empty list to which we will add the rows. Then, we loop over each row in the spreadsheet, checking whether the first value (the left most cell) is a float. If it is, then we add all the cells' values to our list. This works because the `Date` column, when read in, appears as a float, and we are only interested in those rows in the sheet that start with a date. Note that we also start our loop on line 36 to skip over the summary data in the sheet.

7. Let's now again check that the number of rows we expect are in our list:

```
len(ipo_list)
```

The preceding code generates the following output:

3156

After eliminating the header and other rows we aren't interested in, this looks about right.

Working on the DataFrame

Let's now move on to getting our DataFrame ready to work with:

```
df = pd.DataFrame(ipo_list)

df.head()
```

The preceding code generates the following output:

	0	1	2	3	4	5	6	7	8	9	10	11
0	43110.0	Nebula Acquisition	NEBUU	Deutsche Bank Securities/ Goldman Sachs	10.0	10.03	10.02	0.002	0.03	0.02	1	
1	43112.0	Industrial Logistics Properties Trust	ILPT	UBS Investment Bank/ Citigroup/ RBC Capital Ma...	24.0	23.70	23.35	-0.0270833	-0.30	-0.65	1	
2	43112.0	Liberty Oilfield Services	LBRT	Morgan Stanley/ Goldman, Sachs/ Wells Fargo Se...	17.0	21.20	21.75	0.279412	4.20	4.75	2	
3	43112.0	Platinum Eagle Acquisition	EAGLU	Deutsche Bank Securities/ BofA Merrill Lynch	10.0	10.06	10.01	0.001	0.06	0.01	1	
4	43124.0	PagSeguro Digital Ltd.	PAGS	Goldman Sachs/ Morgan Stanley	21.5	28.20	29.20	0.35814	6.70	7.70	3	

The data looks good, so let's now add our columns:

```
df.columns = ['Date', 'Company', 'Ticker', 'Managers', \
              'Offer Price', 'Opening Price', '1st Day Close', \
              '1st Day % Chg', '$ Chg Open', '$ Chg Close', \
              'Star Rating', 'Performed']

df.head()
```

The preceding code generates the following output:

	Date	Company	Ticker	Managers	Offer Price	Opening Price	1st Day Close	1st Day % Chg	\$ Chg Open	\$ Chg Close	Star Rating	Performed
0	43110.0	Nebula Acquisition	NEBUU	Deutsche Bank Securities/ Goldman Sachs	10.0	10.03	10.02	0.002	0.03	0.02	1	
1	43112.0	Industrial Logistics Properties Trust	ILPT	UBS Investment Bank/ Citigroup/ RBC Capital Ma...	24.0	23.70	23.35	-0.0270833	-0.30	-0.65	1	
2	43112.0	Liberty Oilfield Services	LBRT	Morgan Stanley/ Goldman, Sachs/ Wells Fargo Se...	17.0	21.20	21.75	0.279412	4.20	4.75	2	
3	43112.0	Platinum Eagle Acquisition	EAGLU	Deutsche Bank Securities/ BofA Merrill Lynch	10.0	10.06	10.01	0.001	0.06	0.01	1	
4	43124.0	PagSeguro Digital Ltd.	PAGS	Goldman Sachs/ Morgan Stanley	21.5	28.20	29.20	0.35814	6.70	7.70	3	
5	43125.0	Eyenovia	EYEN	Ladenburg Thalmann/ Roth Capital Partners	10.0	10.00	9.92	-0.008	0.00	-0.08	1	
6	43125.0	Gates Industrial Corporation plc	GTES	Citigroup/ Morgan Stanley/ UBS Investment Bank	19.0	19.05	18.50	-0.0263158	0.05	-0.50	1	

Let's now convert that Date column from a float to a proper date. The `xlrd` library has some functionality that can help us with that. We'll use it in a function to get our dates in the proper format:

```
def to_date(x):
    return xlrd.xldate.xldate_as_datetime(x, wb.datemode)
df['Date'] = df['Date'].apply(to_date)
df
```

The preceding code generates the following output:

	Date	Company	Ticker	Managers	Offer Price	Opening Price	1st Day Close	1st Day % Chg	\$ Chg Open	\$ Chg Close	Star Rating	Performed
0	43110.0	Nebula Acquisition	NEBUU	Deutsche Bank Securities/ Goldman Sachs	10.0	10.03	10.02	0.002	0.03	0.02	1	
1	43112.0	Industrial Logistics Properties Trust	ILPT	UBS Investment Bank/ Citigroup/ RBC Capital Ma...	24.0	23.70	23.35	-0.0270833	-0.30	-0.65	1	
2	43112.0	Liberty Oilfield Services	LBRT	Morgan Stanley/ Goldman, Sachs/ Wells Fargo Se...	17.0	21.20	21.75	0.279412	4.20	4.75	2	
3	43112.0	Platinum Eagle Acquisition	EAGLU	Deutsche Bank Securities/ BofA Merrill Lynch	10.0	10.06	10.01	0.001	0.06	0.01	1	
4	43124.0	PagSeguro Digital Ltd.	PAGS	Goldman Sachs/ Morgan Stanley	21.5	28.20	29.20	0.35814	6.70	7.70	3	
5	43125.0	Eyenovia	EYEN	Ladenburg Thalmann/ Roth Capital Partners	10.0	10.00	9.92	-0.008	0.00	-0.08	1	
6	43125.0	Gates Industrial Corporation plc	GTES	Citigroup/ Morgan Stanley/ UBS Investment Bank	19.0	19.05	18.50	-0.0263158	0.05	-0.50	1	

Now that we have dates that we can work with, let's add some additional date-related columns that can help us work with the data better:

```
df['Year'], df['Month'], df['Day'], df['Day of Week'] = \
df['Date'].dt.year, df['Date'].dt.month, df['Date'].dt.day,
df['Date'].dt.weekday
df
```

The preceding code generates the following output:

	Date	Company	Ticker	Managers	Offer Price	Opening Price	1st Day Close	1st Day % Chg	\$ Chg Open	\$ Chg Close	Star Rating	Performed	Year	Month	Day
0	2018-01-10	Nebula Acquisition	NEBUU	Deutsche Bank Securities/ Goldman Sachs	10.0000	10.030000	10.020000	0.002	0.030000	0.020000	1	2018	1	10	
1	2018-01-12	Industrial Logistics Properties Trust	ILPT	UBS Investment Bank/ Citigroup/ RBC Capital Ma...	24.0000	23.700000	23.350000	-0.0270833	-0.300000	-0.650000	1	2018	1	12	
2	2018-01-12	Liberty Oilfield Services	LBRT	Morgan Stanley/ Goldman, Sachs/ Wells Fargo Se...	17.0000	21.200000	21.750000	0.279412	4.200000	4.750000	2	2018	1	12	
3	2018-01-12	Platinum Eagle Acquisition	EAGLU	Deutsche Bank Securities/ BofA Merrill Lynch	10.0000	10.060000	10.010000	0.001	0.060000	0.010000	1	2018	1	12	
4	2018-01-24	PagSeguro Digital Ltd.	PAGS	Goldman Sachs/ Morgan Stanley	21.5000	28.200000	29.200000	0.35814	6.700000	7.700000	3	2018	1	24	
5	2018-01-25	Eyenovia	EYEN	Ladenburg Thalmann/ Roth Capital Partners	10.0000	10.000000	9.920000	-0.008	0.000000	-0.080000	1	2018	1	25	

Now that we've completed those steps, let's check our data in the DataFrame against the data in the original spreadsheet:

```
by_year_cnt = df.groupby('Year')[['Ticker']].count()

by_year_cnt
```

The preceding code generates the following output:

Ticker	Year
2000	237
2001	94
2002	82
2003	80
2004	232
2005	226
2006	237
2007	256
2008	47
2009	61
2010	160
2011	143
2012	145
2013	230
2014	288
2015	173
2016	106
2017	181
2018	178

Comparing this to the same values in the spreadsheet shows us that we have nearly identical values, so we should be good to continue.

We'll take one additional step here to eliminate what are sometimes referred to as *penny stocks*, or particularly low-priced stocks. Then, we'll check the data types to ensure they look appropriate:

```
df.drop(df[df['Offer Price'] < 5].index, inplace=True)

df.reset_index(drop=True, inplace=True)

df.dtypes
```

The preceding code generates the following output:

Date	datetime64[ns]
Company	object
Ticker	object
Managers	object
Offer Price	float64
Opening Price	float64
1st Day Close	float64
1st Day % Chg	object
\$ Chg Open	float64
\$ Chg Close	float64
Star Rating	object
Performed	object
Year	int64
Month	int64
Day	int64
Day of Week	int64
dtype:	object

This looks to be in line with what we expect, with the exception of the 1st Day % Chg column. We'll correct that now by changing the data type to a float:

```
df['1st Day % Chg'] = df['1st Day % Chg'].astype(float)
df.dtypes
```

The preceding code generates the following output:

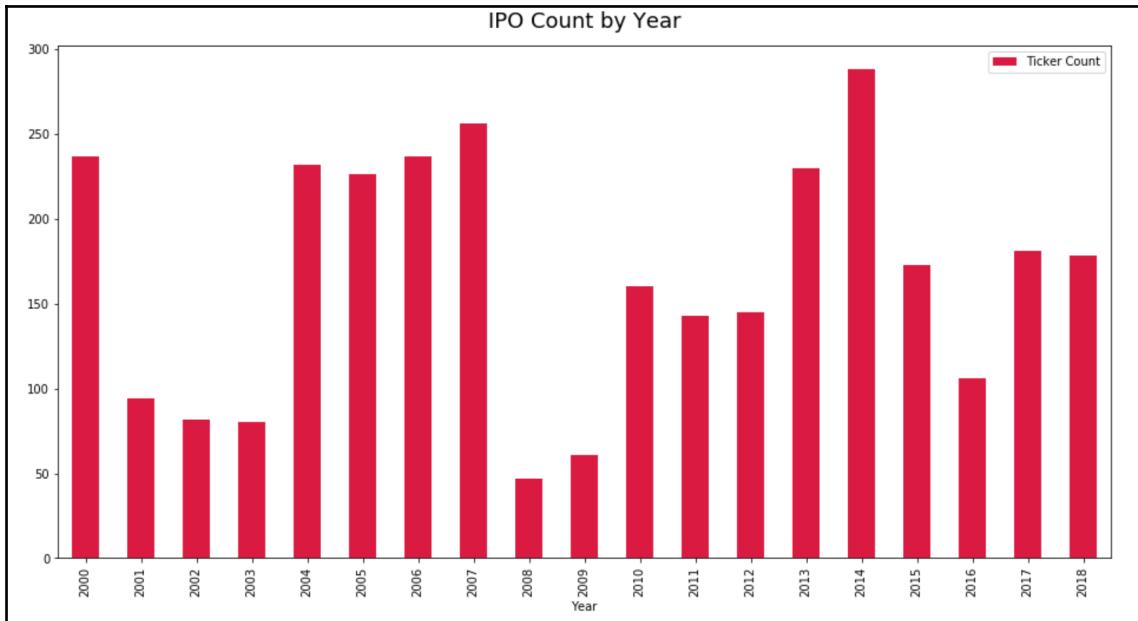
Date	datetime64[ns]
Company	object
Ticker	object
Managers	object
Offer Price	float64
Opening Price	float64
1st Day Close	float64
1st Day % Chg	float64
\$ Chg Open	float64
\$ Chg Close	float64
Star Rating	object
Performed	object
Year	int64
Month	int64
Day	int64
Day of Week	int64
1st Day Open to Close % Chg	float64
1st Day Open to Close \$ Chg	float64
Opening Gap % Chg	float64
Opening Gap \$ Chg	float64
Mgr Count	int64
Lead Mgr	object
dtype: object	

Analyzing the data

The data types all look good now, so we'll begin our exploratory analysis by graphing the number of IPOs since 2000:

```
fig, ax = plt.subplots(figsize=(16,8))
by_year_cnt.plot(kind='bar', ax=ax, color='crimson')
ax.legend(['Ticker Count'])
ax.set_title('IPO Count by Year', fontdict={'size': 18}, y=1.02);
```

The preceding code generates the following output:



From the chart, we can see that most years have over 100 IPOs, but that in the years after and including 2001 and 2008, there was a notable reduction, mostly likely due to the aftermath of 9/11 and the financial crisis.

Summarizing the performance of the stocks

We'll get a quick summary of the performance of the stocks over the past 18 years by executing the following code:

```
summary_by_year = df.groupby('Year')[['1st Day % Chg']].describe()

summary_by_year
```

The preceding code generates the following output:

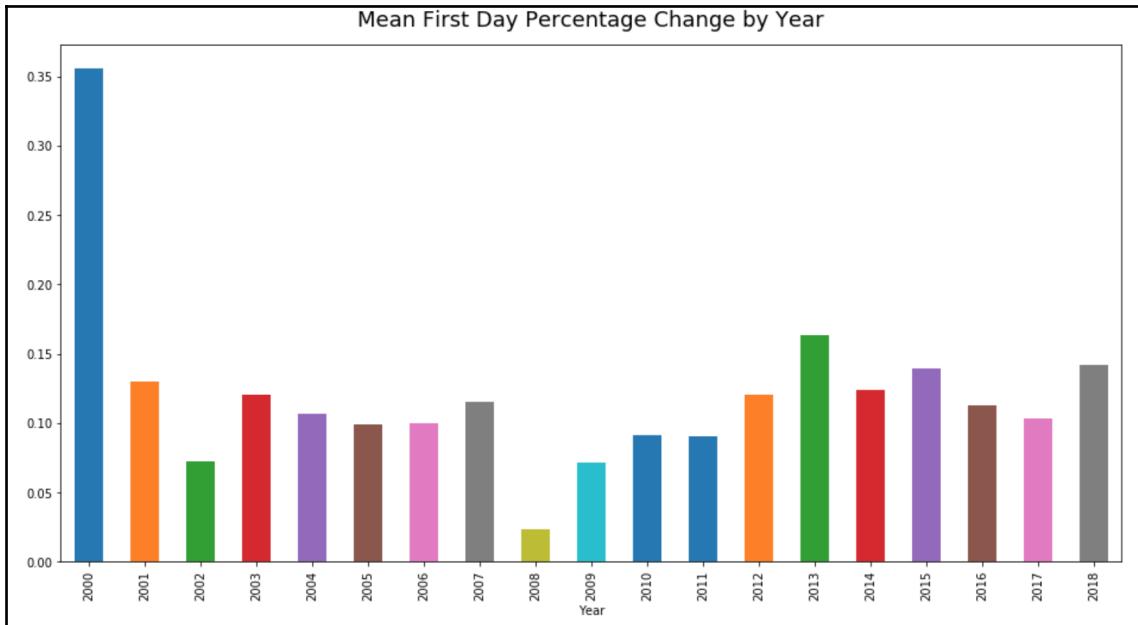
	count	mean	std	min	25%	50%	75%	max
Year								
2000	237.0	0.355621	0.557065	-0.269231	0.004464	0.162500	0.470052	2.775000
2001	93.0	0.130180	0.160606	-0.140625	0.011538	0.088542	0.229167	0.766667
2002	82.0	0.072315	0.141903	-0.330769	0.000000	0.045959	0.135866	0.666667
2003	79.0	0.120958	0.164613	-0.155455	0.002857	0.074286	0.230264	0.885556
2004	232.0	0.107032	0.165621	-0.142500	0.000000	0.051756	0.168558	1.036364
2005	226.0	0.099110	0.274405	-0.164000	0.000000	0.025357	0.143803	3.538519
2006	236.0	0.099694	0.176282	-0.295455	0.000000	0.041652	0.150852	1.254068
2007	256.0	0.115215	0.206049	-0.269091	0.000000	0.033571	0.179853	0.972222
2008	47.0	0.023182	0.134953	-0.199200	-0.034857	0.000000	0.028873	0.575000
2009	61.0	0.071752	0.155818	-0.167500	-0.017600	0.009000	0.131579	0.594500
2010	153.0	0.091201	0.215307	-0.260769	-0.016667	0.023529	0.136667	1.109375
2011	141.0	0.090664	0.218208	-0.215000	-0.006842	0.009167	0.152857	1.344828
2012	143.0	0.120579	0.215977	-0.187500	0.000000	0.053077	0.177124	1.087059
2013	229.0	0.163482	0.277182	-0.352222	0.000000	0.068718	0.245238	1.228333
2014	287.0	0.123636	0.269714	-0.348800	-0.007611	0.042857	0.191127	2.066667
2015	173.0	0.139472	0.299854	-0.274000	0.000000	0.048333	0.205714	1.855556
2016	104.0	0.113253	0.234423	-0.200000	-0.001013	0.038357	0.205474	1.312500
2017	180.0	0.103135	0.203852	-0.410833	0.000000	0.045000	0.159318	1.114000
2018	173.0	0.141809	0.254372	-0.289333	0.000000	0.016000	0.300000	1.281429

From the table, we can see the extraordinary average return of the IPO market in 2000. At over 35%, it is more than double any other year on the list. Also notable is the fact that every year has had a positive average return for first-day performance.

Let's plot first-day performance to get a better feel for it:

```
fig, ax = plt.subplots(figsize=(16, 8))
summary_by_year['mean'].plot(kind='bar', ax=ax)
ax.set_title('Mean First Day Percentage Change by Year', fontdict={'size': 18}, y=1.02);
```

The preceding code generates the following output:



The important point about these numbers is that they are not the first-day performance that the general investing public could expect to receive on that first-day. Only investors who got in on the offering could expect to see these numbers.

The first-day return that the general public could expect to receive would be the difference between the opening price and the closing price. This is entirely different, and much less lucrative. Let's now add a column of data to reflect that value and see the results:

```
df['1st Day Open to Close % Chg'] = ((df['1st Day Close'] - df['Opening Price'])/df['Opening Price'])

df['1st Day Open to Close % Chg'].describe()
```

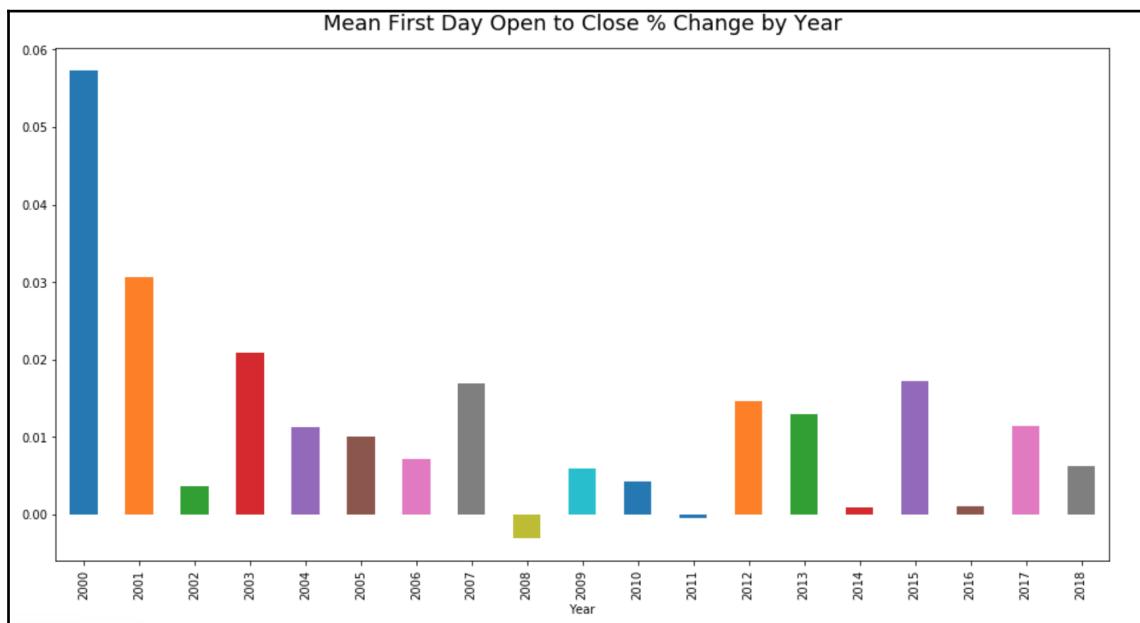
The preceding code generates the following output:

```
count      3132.000000
mean       0.013276
std        0.108117
min       -0.492812
25%       -0.028939
50%        0.000000
75%        0.040000
max        1.594175
Name: 1st Day Open to Close % Chg, dtype: float64
```

This shows returns that are markedly less exciting. Let's now plot them as before:

```
fig, ax = plt.subplots(figsize=(16, 8))
df.groupby('Year')['1st Day Open to Close % Chg'].mean().plot(kind='bar',
ax=ax)
ax.set_title('Mean First Day Open to Close % Change by Year',
fontdict={'size': 18}, y=1.02);
```

The preceding code generates the following output:



Comparing the preceding chart to the earlier one, it is clear that annual average returns on the first day are displayed in the range of their order of magnitude which is lower in many cases.

Baseline IPO strategy

Let's now suppose that we bought one share of every IPO at the exact opening tick and sold them at the precise closing price listed in these figures; what would our returns look like in terms of dollars earned?

To answer this question, let's look at the actual dollar price change from open to close:

```
df['1st Day Open to Close $ Chg'] = (df['1st Day Close'] - df['Opening Price'])

df[df['Year']==2018].sum()
```

The preceding code generates the following output:

Offer Price	2493.885000
Opening Price	2888.300000
1st Day Close	2916.510000
1st Day % Chg	24.532930
\$ Chg Open	394.415000
\$ Chg Close	436.125000
Year	349114.000000
Month	945.000000
Day	3257.000000
Day of Week	508.000000
1st Day Open to Close % Chg	1.080821
1st Day Open to Close \$ Chg	28.210000
Opening Gap % Chg	23.266162
Opening Gap \$ Chg	394.415000
Mgr Count	457.000000
dtype:	float64

From this, we see the first day open-to-close dollar total is just above \$28. That number is for over 173 IPOs so far in 2018:

```
df[df['Year']==2018]['1st Day Open to Close $ Chg'].describe()
```

The preceding code generates the following output:

```
count      173.000000
mean       0.163064
std        2.244747
min       -6.260000
25%      -0.840000
50%       0.000000
75%       0.500000
max        9.940000
Name: 1st Day Open to Close $ Chg, dtype: float64
```

That reflects a mean first-day gain of just over 16 cents per IPO. And remember, this is under ideal conditions where we ignore transaction costs and slippage.

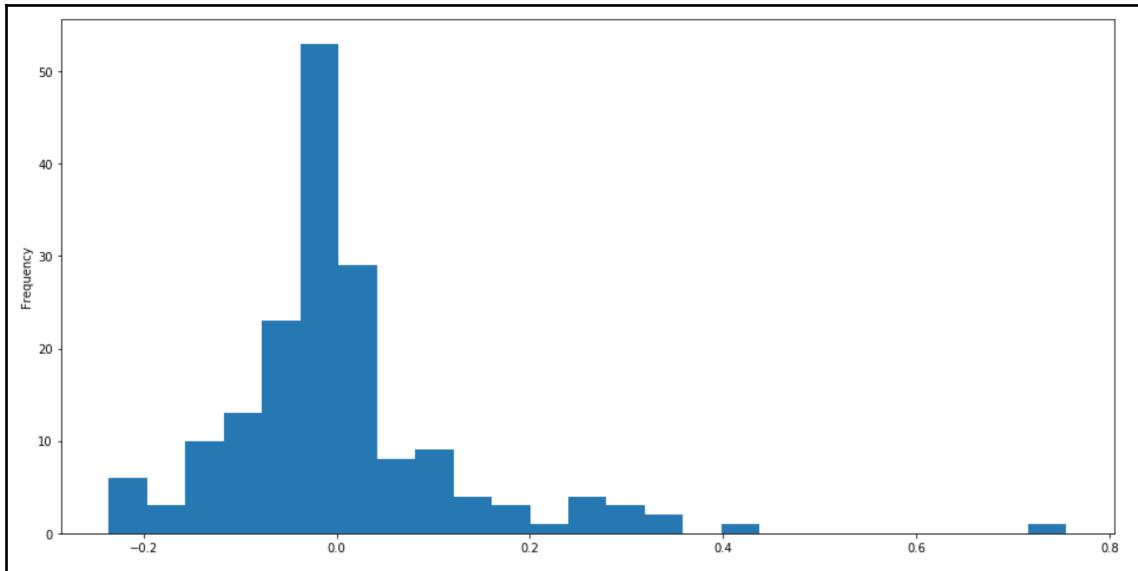


Slippage is the difference between your attempted entry or exit price for a target stock and the price at which your order is actually fulfilled.

Let's now take a look at what the distribution of returns for these IPOs look like. This may help us understand how to improve our returns over the baseline Naïve Bayes strategy of just buying every IPO:

```
fig, ax = plt.subplots(figsize=(16,8))
df[df['Year']==2018]['1st Day Open to Close % Chg'].plot(kind='hist',
bins=25, ax=ax)
```

The preceding code generates the following output:



We see that returns are centered around zero, but there is a long tail to the right where there are some exceptional returns. It would be quite financially rewarding if we were able to identify some of the commonalities that these exceptional IPOs have that we could capitalize on.

Let's see whether we can use machine learning to help improve our results from a Naïve Bayes approach. A reasonable strategy would seem to be targeting that long tail on the right, so we'll focus on feature engineering in the next section.

Data cleansing and feature engineering

What might impact the performance of an offering as it begins trading? Perhaps the performance of the market in general or the prestige of the underwriters could impact it? Perhaps the day of the week or the month that it trades is important? The consideration and inclusion of these factors in a model is called **feature engineering**, and modeling this is nearly as important as the data that you use to build the model. If your features aren't informative, your model simply won't have value.

Let's begin this process by adding a few features that we expect may influence the performance of an IPO.

Adding features to influence the performance of an IPO

One measure of demand that could be informative is the **opening gap**. This is the difference between the offer price and the opening price of the issue. Let's add that to our DataFrame:

```
df['Opening Gap % Chg'] = (df['Opening Price'] - df['Offer Price'])/df['Offer Price']
```

Next, let's get a count of the number of underwriters on the offering. Perhaps having more banks involved leads to better marketing of the issue? This is demonstrated in the following code block:

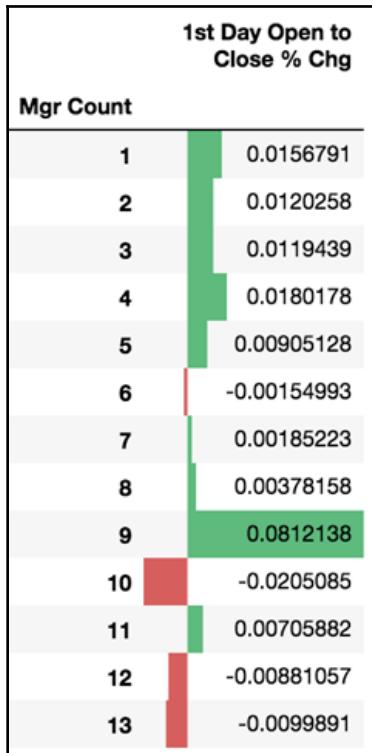
```
def get_mgr_count(x):
    return len(x.split('/'))

df['Mgr Count'] = df['Managers'].apply(get_mgr_count)
```

Let's quickly see whether there might be anything to this hypothesis by means of a visualization:

```
df.groupby('Mgr Count')['1st Day Open to Close % Chg'].mean().to_frame().style.bar(align='mid', color=['#d65f5f', '#5fba7d'])
```

The preceding code generates the following output:



It's not apparent what the relationship might be from this chart, but clearly nine bankers is the sweet spot!

Next, let's move on to extracting the first underwriter in the list. This would be the lead, and perhaps the prestige of this bank is important to the first-day gains:

```
df['Lead Mgr'] = df['Managers'].apply(lambda x: x.split('/')[0])
```

Next, let's take a quick peek at the data in the new column that we have created:

```
df['Lead Mgr'].unique()
```

The preceding code generates the following output:

```
array(['Deutsche Bank Securities', 'UBS Investment Bank',
       'Morgan Stanley', 'Goldman Sachs', 'Ladenburg Thalmann',
       'Citigroup', 'B. Riley FBR', 'Jefferies', 'Credit Suisse',
       'BofA Merrill Lynch', 'J.P. Morgan', 'EarlyBirdCapital',
       'Roth Capital Partners', 'Oppenheimer & Co.', '\u200bJefferies',
       'Cantor', 'Piper Jaffray', 'Barclays',
       'Sandler O'Neill & Partners', 'Goldman Sachs (Asia)',
       'Network 1 Financial Securities', 'Oppenheimer', 'Raymond James',
       'Stifel', 'Wells Fargo Securities', 'Baird', 'Stephens', 'Cowen',
       'Benchmark', 'Leerink Partners', 'Chardan', 'Evercore ISI',
       'Maxim Group ', '\u200bCredit Suisse',
       'Keefe, Bruyette & Woods (A Stifel Company)', 'Cowen and Company',
       'BMO Capital Markets', 'Joseph Gunnar & Co.', 'CICC',
       'Maxim Group', "Sandler O'Neill + Partners",
       'Cantor Fitzgerald & Co.', 'Joseph Gunnar',
       'Rodman & Renshaw (a Unit of H.C. Wainwright & Co.)',
       'CIBC Capital Markets', 'Goldman, Sachs', 'UBS Securities',
```

Even a cursory examination of the preceding shows us that we have some genuine issues with the data. Many names are replicated with different spellings and punctuation. We could, at this point, stop and attempt to clean up the data, and this would be the proper course of action if we were going to rely on our model for anything serious, but as this is just a toy project, we'll forge ahead and hope that the impact is minimal.

Binary classification with logistic regression

Instead of attempting to predict what the total first-day return will be, we are going to attempt to predict whether the IPO will be one we should buy for a trade or not. It is here that we should point out that this is not investment advice and is for illustrative purposes only. Please don't run out and start day trading IPOs with this model willy-nilly. It will end badly.

Now, to predict a binary outcome (that's a 1 or 0/yes or no), we will start with a model called **logistic regression**. Logistic regression is actually a binary classification model rather than regression. But it does utilize the typical form of a linear regression; it just does so within a logistic function.

A typical single variable regression model takes the following form:

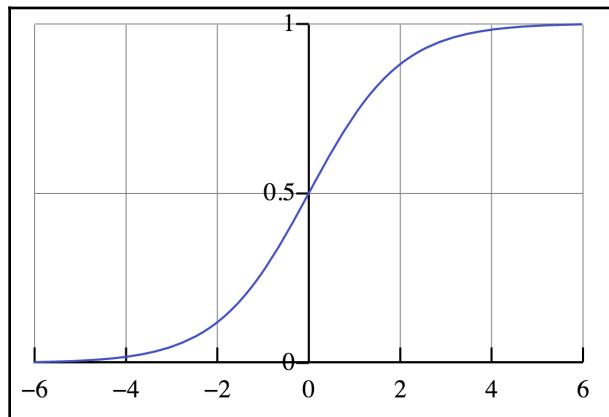
$$t = \beta_0 + \beta_1 x$$

Here, t is a linear function of a single explanatory variable, x . This can, of course, be expanded to be a linear combination of many variables. The problem with this form for a binary outcome variable is that t does not naturally fall between 1 and 0.

The logistic function seen in the following equation has some quite favorable mathematical properties, including the fact that it can take any number as an input (t here) and return a result that falls between 0 and 1:

$$\sigma(t) = \frac{e^t}{e^t + 1} = \frac{1}{1 + e^{-t}}$$

The graph is represented as below:



By replacing t with our regression function, we now have a model that is able to both give us information on the importance of each predictor (the beta coefficients) and provide a form that can be used to give us a binary prediction that represents the probability of success, or a *positive result*:

$$p(x) = \frac{1}{1 + e^{-(\beta_0 + \beta_1 x)}}$$

Before we can move on to modeling our data, we need to put it in a form that is appropriate for scikit-learn.

We'll start by importing a library that can help us with this task; it's called `patsy`. It can be pip-installed if necessary:

```
from patsy import dmatrix
```

Creating the target for our model

Now, we'll create the target for our model. This is the column that will tell our model whether each IPO should have been invested in or not. We are going to say that we should invest in any IPO that has a 2.5% or greater return on day one. Obviously, this is an arbitrary number, but it seems like a reasonable value for the investment to be worthy of our attention:

```
y = df['1st Day Open to Close % Chg'].apply(lambda x: 1 if x > .025 else 0)
```

Now that we have set our target column, we need to set up our predictor variables. We'll again use `patsy` for this:

```
X = dmatrix("Q('Opening Gap % Chg') + C(Q('Month'), Treatment) + C(Q('Day of Week'), Treatment)\\ + Q('Mgr Count') + Q('Lead Mgr') + Q('Offer Price') + C(Q('Star Rating'), Treatment)", df, return_type="dataframe")
```

Let's discuss what's going on in that line of code. `X` here is our design matrix, or the matrix that contains our predictor variables. We have included the things we discussed earlier that could have some impact on the performance: the size of the opening gap, the month and day of the IPO, the offering price, the lead manager, the number of managers, and finally, the star rating that `IPOScoop.com` provides in advance of the IPO's listing.

To give some explanation regarding the `Q`s and `C`s found in the lines of code, the `Q`s are simply used to provide quotes in the formula for columns that have white space in their names, and the `C`s are used to indicate that the referenced column should be treated as categorical features and dummy-coded.

Dummy coding

Dummy coding is a method where, if we had one column that had a student's favorite class as a predictor variable in one column, we would turn each class into its own column and then place a 1 in that column if it was the favorite class of the student, as seen in the following diagram:

Student	Favorite class	Dummy variables		
		Science	Math	Language
1	Science	1	0	0
2	Science	1	0	0
3	Language	0	0	1
4	Math	0	1	0
5	Language	0	0	1
6	Math	0	1	0

Source: <http://www.statisticssolutions.com/dummy-coding-the-how-and-why/>

Once that is done, then the next step is to actually drop one of those columns. The dropped column then becomes the **base case**. All the other cases are then compared to that case. In our IPO example using months as predictors, we will drop January, for example, and then all the other months will be judged against January's performance. The same goes for the days of the week or any other categorical predictor. This dropping of a column is to prevent multicollinearity, which would have a negative impact on the explanatory power of the model.

Let's take a look at what this coding looks like by running the following in a Jupyter cell:

```
x
```

The preceding code generates the following output:

Intercept	C(Q('Month'), Treatment) [T.2]	C(Q('Month'), Treatment) [T.3]	C(Q('Month'), Treatment) [T.4]	C(Q('Month'), Treatment) [T.5]	C(Q('Month'), Treatment) [T.6]	C(Q('Month'), Treatment) [T.7]	C(Q('Month'), Treatment) [T.8]	C(Q('Month'), Treatment) [T.9]	C(Q('Month'), Treatment) [T.10]	...
0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0 ...
1	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0 ...
2	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0 ...
3	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0 ...
4	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0 ...
5	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0 ...
6	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0 ...
7	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0 ...
8	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0 ...
9	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0 ...
10	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0 ...

Now that we have both our x and y , we are ready to fit our model. We are going to use a very basic train/test split and simply train our model on all but the last 200 IPOs:

```
from sklearn.linear_model import LogisticRegression

X_train = X[:-200]
y_train = y[:-200]

X_test = X[-200:]
y_test = y[-200:]

clf = LogisticRegression()
clf.fit(X_train, y_train)
```

And, with that, we have our model. Let's examine the performance of this very simple model.

Examining the model performance

We'll start by making predictions on our test data and then we'll examine whether our predictions were correct:

```
y_hat = clf.predict(X_test)
y_true = y_test

pdf = pd.DataFrame({'y_true': y_true, 'y_hat': y_hat})

pdf['correct'] = pdf.apply(lambda x: 1 if x['y_true'] == x['y_hat'] else 0,
axis=1)

pdf
```

The preceding code generates the following output:

	y_true	y_hat	correct
2932	1	1	1
2933	0	0	1
2934	0	0	1
2935	0	0	1
2936	0	0	1
2937	1	1	1
2938	1	0	0
2939	0	0	1
2940	1	0	0
2941	0	1	0
2942	0	1	0

Let's now look at what percentage of the 200 IPOs in our `test` dataset we should have invested in—remember, that means they rose over 2.5% from the open to the close:

```
pdf['y_true'].value_counts(normalize=True)
```

The preceding code generates the following output:

```
1    0.51
0    0.49
Name: y_true, dtype: float64
```

So, just north of half the IPOs rose over 2.5% from their opening tick to the closing tick. Let's see how accurate our model's calls were:

```
pdf['correct'].value_counts(normalize=True)
```

The preceding code generates the following output:

```
1    0.51
0    0.49
Name: correct, dtype: float64
```

Well, it looks like our model was about as accurate as a coin flip. That doesn't seem too promising. But with investing, what is important is not the accuracy but the expectancy. If we had a number of small losses, but a couple of huge wins, overall, the model could still be very profitable. Let's examine whether that's the case here. We'll join our results data with the first-day change data to explore this:

```
results = pd.merge(df[['1st Day Open to Close $ Chg']], pdf,  
left_index=True, right_index=True)  
  
results
```

The preceding code generates the following output:

	1st Day Open to Close \$ Chg	y_true	y_hat	correct
2932	9.125000	1	1	1
2933	-15.875000	0	0	1
2934	-1.343750	0	0	1
2935	-2.062500	0	0	1
2936	-1.234375	0	0	1
2937	20.609375	1	1	1
2938	2.375000	1	0	0
2939	-3.187500	0	0	1
2940	0.875000	1	0	0
2941	0.000000	0	1	0
2942	-2.000000	0	1	0

First, let's see what our results would have looked like for one share of every one of the 200 IPOs in our test data:

```
results['1st Day Open to Close $ Chg'].sum()
```

The preceding code generates the following output:

215.71875

From this, we see that we would have gained over \$215 in an ideal cost-free scenario. Now, let's examine some of the other statistics concerning these IPOs:

```
results['1st Day Open to Close $ Chg'].describe()
```

The preceding code generates the following output:

```
count      200.000000
mean       1.078594
std        4.136920
min       -15.875000
25%      -0.578125
50%       0.312500
75%       1.902344
max       20.609375
Name: 1st Day Open to Close $ Chg, dtype: float64
```

Based on the preceding, we see that the average gain was just over \$1, and the largest loss was 15 times that. How does our model fare against those numbers? First, we look at the trades our model said we should take and the resulting gains:

```
# ipo buys
results[results['y_hat']==1]['1st Day Open to Close $ Chg'].sum()
```

The preceding code generates the following output:

```
51.109375
```

Let's look at the other statistics as well:

```
# ipo buys
results[results['y_hat']==1]['1st Day Open to Close $ Chg'].describe()
```

The preceding code generates the following output:

```
count      34.000000
mean       1.503217
std        6.563061
min       -9.875000
25%      -1.968750
50%       0.507812
75%       4.542969
max       20.609375
Name: 1st Day Open to Close $ Chg, dtype: float64
```

Here, we see that our model suggested investing in only 34 IPOs, the mean gain rose to \$1.50, the largest loss was reduced to under \$10, and we still were able to capture the best performing IPO. It's not stellar, but we may be on to something. We'd need to explore further to really know whether we do have something worth expanding further.

Now, let's move on to examine the factors that seem to influence our model's performance.

Generating the importance of a feature from our model

One of the nice features of logistic regression is that it offers predictor coefficients that can tell us the relative importance of the predictor variables or features. For categorical features, a positive sign on a feature's coefficient tells us that, when present, this feature increases the probability of a positive outcome versus the baseline. For continuous features, a positive sign tells us that an increase in the value of a feature corresponds to an increase in the probability of a positive outcome. The size of the coefficient tells us the magnitude of the increase in probability.

Let's generate the importance of the feature from our model, and then we can examine the impact it has:

```
fv = pd.DataFrame(X_train.columns, clf.coef_.T).reset_index()
fv.columns = ['Coef', 'Feature']
fv.sort_values('Coef', ascending=0).reset_index(drop=True)
```

The preceding code generates the following output:

	Coef	Feature
0	1.480088	Q('Lead Mgr')[T.BMO Capital Markets]
1	1.296002	Q('Lead Mgr')[T.C.E. Unterberg, Towbin]
2	1.124586	Q('Lead Mgr')[T. J.P. Morgan]
3	1.042390	Q('Lead Mgr')[T.Joseph Gunnar]
4	0.980193	Q('Opening Gap % Chg')
5	0.967776	Q('Lead Mgr')[T.EarlyBridCapital]
6	0.907780	Q('Lead Mgr')[T.Deutsche Banc Alex Brown]
7	0.882732	Q('Lead Mgr')[T.Keefe, Bruyette & Woods]
8	0.880075	Q('Lead Mgr')[T.Sandler O'Neill & Partners]
9	0.848933	Q('Lead Mgr')[T.Deutsche Banc Alex. Brown]
10	0.848185	Q('Lead Mgr')[T.US Bancorp Piper Jaffray]

In the preceding screenshot, we see those features with the largest coefficients. Let's look at days of the week and their impact there:

```
fv[fv['Feature'].str.contains('Day')]
```

The preceding code generates the following output:

	Coef	Feature
12	-0.082384	C(Q('Day of Week'), Treatment)[T.1]
13	0.053059	C(Q('Day of Week'), Treatment)[T.2]
14	0.187880	C(Q('Day of Week'), Treatment)[T.3]
15	0.080080	C(Q('Day of Week'), Treatment)[T.4]
16	-0.200857	C(Q('Day of Week'), Treatment)[T.5]

Here, the first day of the week would be Monday and would be coded as T.0, or the base case. All the other days of the week would be compared to Monday. From the preceding screenshot, we see that Thursday appears to be the best day of the week. Saturday appears to be a terrible day of the week to have an IPO, most likely because the market is closed that day. (In all likelihood, those dates are just incorrectly recorded.)

Looking further at the features with the highest coefficients, we can now appreciate that extracting useful information for the predictive value of each is difficult, since many of those features are for things that no longer exist. For example, while Deutsche Bank still exists, it no longer underwrites as Deutsche Banc Alex. Brown, so that is actually conveying historical information rather than information that can be useful going forward.

Another issue is that features do not reflect how frequently they had an impact. The appearance of a bank that was only in business in 2000 and had 3 hugely successful IPOs would have a very large positive coefficient, but would be meaningless in our modeling efforts.

Random forest classifier method

Another method of modeling that tells us which features have an impact on our model is the feature importance that comes out of a random forest classifier. This more accurately reflects the true impact of a given feature.

Let's run our data through this type of model and examine the results:

```
from sklearn.ensemble import RandomForestClassifier
clf_rf = RandomForestClassifier(n_estimators=1000)
clf_rf.fit(X_train, y_train)

f_importances = clf_rf.feature_importances_

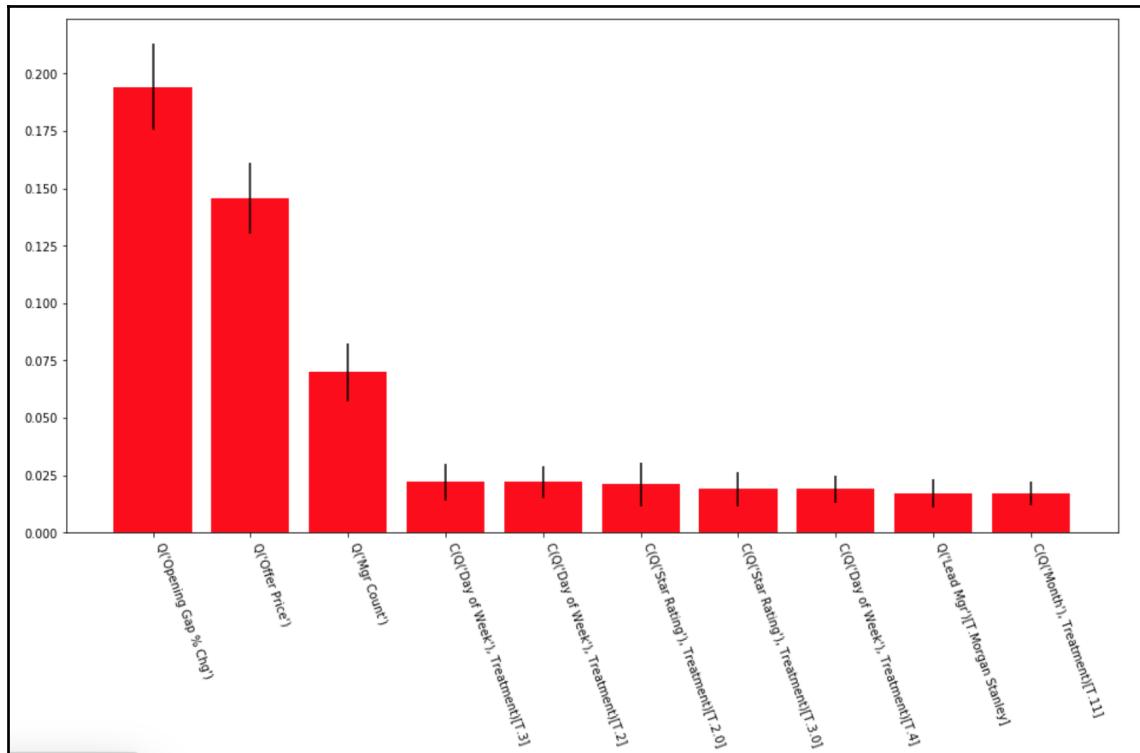
f_names = X_train.columns
f_std = np.std([tree.feature_importances_ for tree in clf_rf.estimators_],
axis=0)

zz = zip(f_importances, f_names, f_std)
zzs = sorted(zz, key=lambda x: x[0], reverse=True)

n_features = 10
imps = [x[0] for x in zzs[:n_features]]
labels = [x[1] for x in zzs[:n_features]]
errs = [x[2] for x in zzs[:n_features]]

fig, ax = plt.subplots(figsize=(16, 8))
ax.bar(range(n_features), imps, color="r", yerr=errs)
plt.xticks(range(n_features), labels)
plt.setp(ax.xaxis.get_majorticklabels(), rotation=-70, ha="left");
```

The preceding code generates the following output:



In the preceding code, we ran a random forest classifier, extracted and sorted the importance of the features, and then graphed those values with their error bars.

From this data, we see that what has the most impact on the model is the opening gap, the offer price, and the number of managers involved in the deal. These would all seem to make sense as having predictive values, since they indicate that there is strong demand for the deal.

Summary

We covered a lot of ground in this chapter, but we've only just scratched the surface in terms of how to build this type of model. Hopefully, you've gained a better understanding of the modeling process, from cleaning the data, to engineering the features, to testing. And hopefully, you'll use this information to extend the model on your own and improve upon it.

In the next chapter, we'll turn our attention to a very different domain, as we move from numeric data to text-based data.

5

Create a Custom Newsfeed

I read *a lot*. Some might even say compulsively. I've been known to consume more than a hundred articles on some days. But despite this, I frequently find myself searching for more to read. I suffer from this sneaking suspicion that I have missed something interesting, and will forever suffer a gap in my knowledge!

If you suffer from similar symptoms, fear not, because in this chapter, I'm going to reveal one simple trick to finding all the articles you want to read without having to dig through the dozens that you don't.

By the end of this chapter, you'll have learned how to build a system that understands your taste in news, and will send you a personally tailored newsletter each day.

Here's what we'll cover in this chapter:

- Creating a supervised training set with the Pocket app
- Leveraging the Pocket API to retrieve stories
- Using the Embedly API to extract story bodies
- Natural language processing basics
- Support vector machines
- IFTTT integration with RSS feeds and Google Sheets
- Setting up a daily personal newsletter

Creating a supervised training set with Pocket

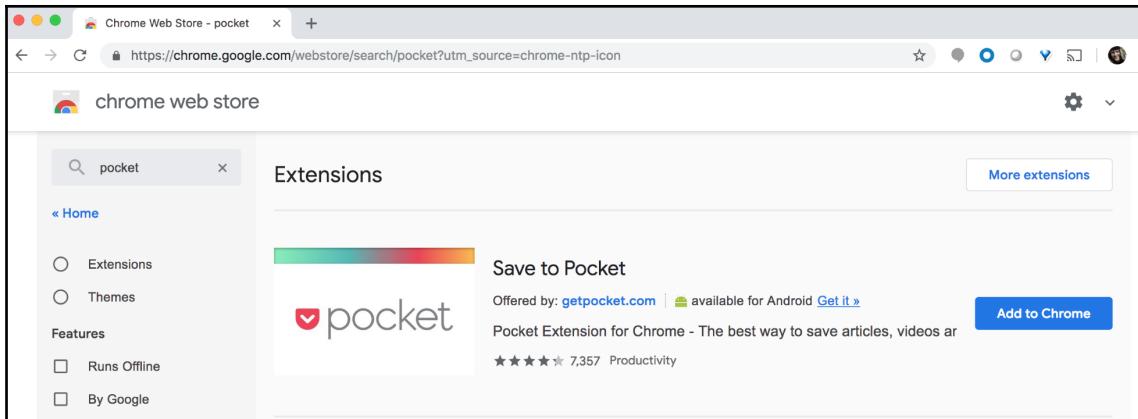
Before we can create a model of our taste in news articles, we need training data. This training data will be fed into our model in order to teach it to discriminate between the articles we'd be interested in and those we would not. To build this corpus, we will need to annotate a large number of articles to correspond to these interests. We'll label each article either *y* or *n*, indicating whether it is the type of article we would want to have sent to us in our daily digest or not.

To simplify this process, we'll use the Pocket app. Pocket is an application that allows you to save stories to read later. You simply install the browser extension, and then click on the Pocket icon in your browser's toolbar when you wish to save a story. The article is saved to your personal repository. One of the great features of Pocket for our purposes is the ability to save the article with a tag of your choosing. We'll use this to mark interesting articles as *y* and non-interesting articles as *n*.

Installing the Pocket Chrome Extension

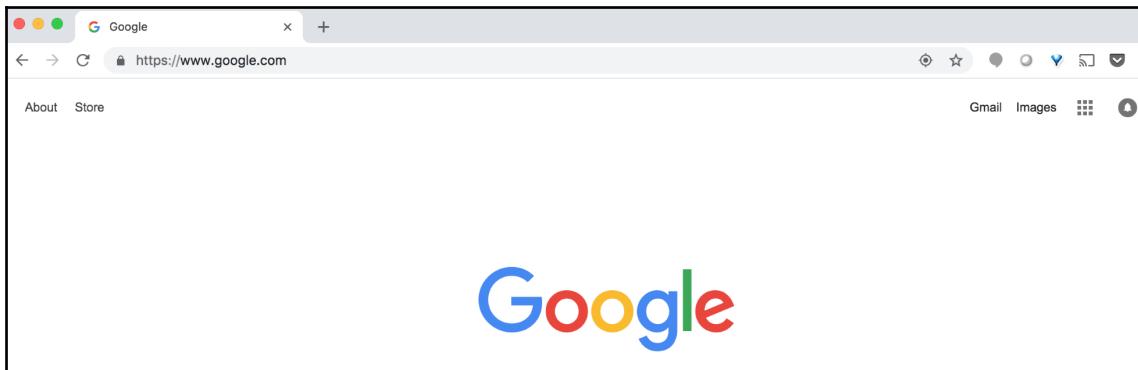
I'm using Google Chrome for this, but other browsers should work similarly. Follow the steps for installing the Pocket Chrome Extension:

1. For Chrome, go to the Google app store and look for the **Extensions** section:

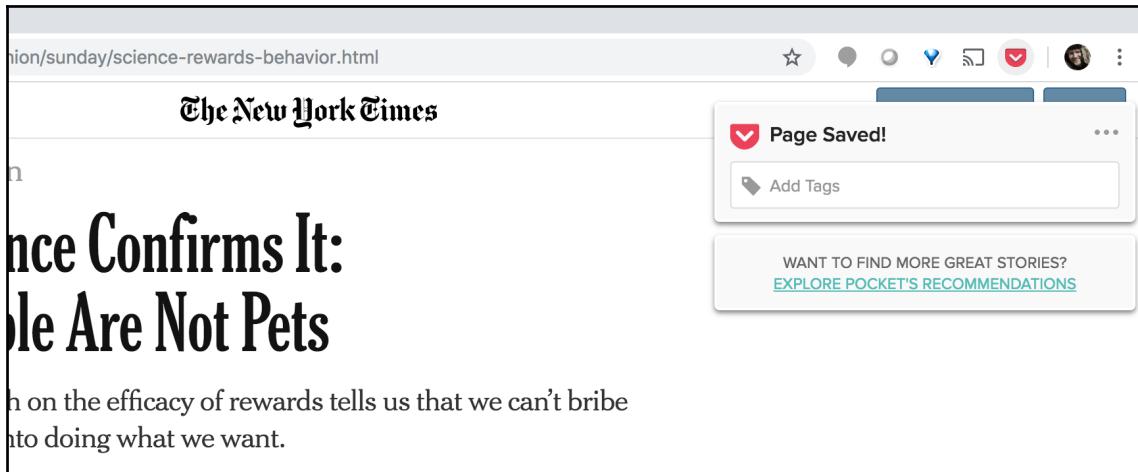


Pocket Chrome Extension

2. Click on **Add to Chrome**. If you already have an account, log in, and, if not, go ahead and sign up (it's free).
3. Once that is complete, you should see the Pocket icon in the upper-right corner of your browser.
4. It will be grayed out, but once there is an article you wish to save, you can click it. It will turn red once the article has been saved:



The saved page can be seen as below:



The New York Times saved page

Now the fun part! As you go through your day, start saving articles you'd like to read, as well as those you wouldn't. Tag the interesting ones with `y`, and the non-interesting ones with `n`. This is going to take some work. Your end results will only be as good as your training set, so you're going to need to do this for hundreds of articles. If you forget to tag an article when you save it, you can always go to the site, <http://www.get.pocket.com>, to tag it there.

Using the Pocket API to retrieve stories

Now that you've diligently saved your articles to Pocket, the next step is to retrieve them. To accomplish this, we'll use the Pocket API. You can sign up for an account at <https://getpocket.com/developer/apps/new>. Follow the steps to achieve that:

1. Click on **Create a New App** in the upper-left corner and fill in the details to get your API key.

2. Make sure to click all of the permissions so that you can add, change, and retrieve articles:

The screenshot shows the 'Create an Application' page on the pocket developer website. The left sidebar contains links for APPS (My Apps, Create a New App), DOCUMENTATION (Overview, Add, Modify, Retrieve, Authentication, Objective-C SDK, Article View API, Preferences API), GETTING STARTED (iOS/Mac, Android, Windows 8, Web, Other Mobile, Adding URLs, Existing Developers), and ADDITIONAL INFO (Rate Limits, Error Handling, Migrating Accounts to OAuth, Application Naming, Terms of Service, Security, Developer FAQ, Developer Support). The main form has fields for Application Name (with placeholder ' '), Application Description (with placeholder ' '), Permissions (checkboxes for Add, Modify, Retrieve), Platforms (checkboxes for iPhone, iPad, Mac, Android - Mobile, Android - Tablet, Extension, Windows - Mobile, Windows - Desktop, Web, Mobile (other), Desktop (other)), and a checkbox for accepting the Terms of Service. A 'CREATE APPLICATION' button is at the bottom.

How to Save Blog Support My List

APPS
My Apps
Create a New App

DOCUMENTATION
Overview
Add
Modify
Retrieve
Authentication
Objective-C SDK
Article View API
Preferences API

GETTING STARTED
iOS/Mac
Android
Windows 8
Web
Other Mobile
Adding URLs
Existing Developers

ADDITIONAL INFO
Rate Limits
Error Handling
Migrating Accounts to OAuth
Application Naming
Terms of Service
Security
Developer FAQ
Developer Support

Create an Application

Application Name:
This is the name of your application that will be displayed to users. 80 character max.
[]

Application Description:
This is the description of your application that will be displayed to users. 120 character max.
[]

Permissions:
Please select one or more permissions for your application below.
Important: You cannot change permissions once they are set without generating a new set of keys.

Add: Add items to a user's list.
 Modify: Modify items in a user's list.
 Retrieve: Retrieve items from a user's list.

Platforms:
We'll generate keys for the platforms you select below. You must select at least one platform to get started and you can always add or remove platforms later.

iPhone iPad Mac
 Android - Mobile Android - Tablet Extension
 Windows - Mobile Windows - Desktop Web
 Mobile (other) Desktop (other)

I accept the [Terms of Service](#).

CREATE APPLICATION

3. Once you have that filled in and submitted, you will receive your **consumer key**.
4. You can find that in the upper-left corner, under **My Apps**. It will look like the following screenshot, but obviously with a real key:

The screenshot shows the 'My Applications' section of the Pocket developer documentation. On the left sidebar, there are sections for 'APPS' (My Apps, Create a New App), 'DOCUMENTATION' (Overview, Add, Modify, Retrieve, Authentication, Objective-C SDK, Article View API, Preferences API), and 'GETTING STARTED' (iOS/Mac, Android, Windows 8, Web, Other Mobile, Adding URLs, Existing Developers). The main content area is titled 'My Applications' and contains a table for 'In Development'. The table has three columns: NAME, PLATFORM, and CONSUMER KEY. One row is listed: 'My Custom Feed' (Platform: Mac, Consumer Key: MY_SECRET_KEY). At the bottom of this section is a red button labeled 'CREATE AN APPLICATION'.

NAME	PLATFORM	CONSUMER KEY
My Custom Feed	Mac	MY_SECRET_KEY

5. Once that is set, you are ready to move on to the next step, which is to set up authorizations. We'll do that now.

6. It requires you to input your consumer key and a redirect URL. The redirect URL can be anything. Here, I have used my Twitter account:

```
import requests
import pandas as pd
import json
pd.set_option('display.max_colwidth', 200)

CONSUMER_KEY = 'enter_your_consumer_key_here'

auth_params = {'consumer_key': CONSUMER_KEY, 'redirect_uri':
'https://www.twitter.com/acombs'}

tkn = requests.post('https://getpocket.com/v3/oauth/request',
data=auth_params)

tkn.text
```

The preceding code results in the following output:

`'code=some_long_access_code'`

7. The output will have the code you'll need for the next step. Place the following in your browser bar:

`https://getpocket.com/auth/authorize?request_token=some_long_access_code&redirect_uri=https%3A//www.twitter.com/acombs`

8. If you change the redirect URL to one of your own, make sure to URL encode it (that's the %3A type stuff you see in the preceding URL).
9. At this point, you should be presented with an authorization screen. Go ahead and approve it, and then we can move on to the next step:

```
# below we parse out the access code from the tkn.text string
ACCESS_CODE = tkn.text.split('=')[1]

usr_params = {'consumer_key': CONSUMER_KEY, 'code': ACCESS_CODE}

usr = requests.post('https://getpocket.com/v3/oauth/authorize',
data=usr_params)

usr.text
```

The preceding code results in the following output:

```
'access_token=some_long_access_token&username=some_user@some_domain.com'
```

10. We'll use the output code here, to move on to retrieving the stories. First, we retrieve the stories tagged n:

```
# below we parse out the access token from the usr.text string
ACCESS_TOKEN = usr.text.split('=')[1].split('&')[0]

no_params = {'consumer_key': CONSUMER_KEY,
'access_token': ACCESS_TOKEN,
'tag': 'n'}

no_result = requests.post('https://getpocket.com/v3/get',
data=no_params)

no_result.text
```

The preceding code results in the following output:

```
{"status":1,"complete":1,"list":[{"item_id": "1167823383","resolved_id": "1167823383","given_url": "http://www.businessinsider.com/gates-dont-expect-the-nuclear-agreement-to-lead-to-a-more-moderate-iran-2016-1","given_title": "GATES: Nuclear agreement won't lead to moderate Iran - Business Insider","favorite": "0","status": "0","time_added": "1453255198","time_updated": "1453255217","time_read": "0","time_favorited": "0","sort_id": "0","resolved_title": "GATES: Don't expect the nuclear agreement to lead to a more moderate Iran","resolved_url": "http://www.businessinsider.com/gates-dont-expect-the-nuclear-agreement-to-lead-to-a-more-moderate-iran-2016-1","excerpt": "Former defense secretary Robert Gates isn't optimistic that the landmark July 2015 nuclear deal with Iran will lead the countr"}]
```

You'll notice that we have a long JSON string on all the articles that we tagged n. There are several keys in this, but we are really only interested in the URL at this point.

11. We'll go ahead and create a list of all the URLs from this:

```
no_jf = json.loads(no_result.text)
no_jd = no_jf['list']

no_urls=[]
for i in no_jd.values():
    no_urls.append(i.get('resolved_url'))

no_urls
```

The preceding code results in the following output:

```
[ 'http://www.slate.com/articles/double_x/doublex/2016/01/kermit_gosnell_s_atrocities_aren_t_an_argument_for_strict_abortion_laws.html',
  'http://bleacherreport.com/articles/2608872-australian-open-2016-results-winners-scores-stats-from-monday-singles-bracket',
  'http://www.slate.com/blogs/xx_factor/2016/01/14/rihanna_ahead_of_beyonce_in_the_celebrity_endorsement_game.html',
  'http://www.nzherald.co.nz/nz/news/article.cfm?c_id=1&objectid=11576760',
  'https://blogs.msdn.microsoft.com/oldnewthing/20160114-00/?p=92851',
  'https://www.washingtonpost.com/national/energy-environment/conservation-groups-demand-end-to-refuge-occupation/2016/01/19/bb83a94e-beff-11e5-98c8-7fab78677d51_story.html',
```

List of URLs

12. This list contains all the URLs of stories we aren't interested in. Let's now put that in a DataFrame and tag it as such:

```
no_uf = pd.DataFrame(no_urls, columns=['urls'])
no_uf = no_uf.assign(wanted = lambda x: 'n')

no_uf
```

The preceding code results in the following output:

	urls	wanted
0	http://www.businessinsider.com/gates-dont-expect-the-nuclear-agreement-to-lead-to-a-more-moderate-iran-2016-1	n
1	http://www.businessinsider.com/r-islamic-state-frees-270-of-400-people-it-kidnapped-from-syrias-deir-al-zor-2016-1	n
2	https://blogs.iyogeshjoshi.com/get-a-free-ssl-tls-certificate-for-your-website-and-setting-up-with-nodejs-server-b5189ac8e007?gi=316c7590f73e	n
3	https://www.usmagazine.com/entertainment/news/whoopi-slams-oscars-boycott-on-the-view-candace-cameron-bure-walks-out-w162078/	n
4	https://www.nbcnews.com/business/business-news/univision-s-hook-onion-play-coveted-millennials-n499906	n
5	http://www.playbill.com/news/article/12-things-you-should-know-about-hello-dolly-380518	n
6	http://www.people.com/article/celine-dion-husband-planned-funeral-to-ease-burden	n
7	https://www.usatoday.com/story/life/tv/2016/01/19/queen-victoria-series-take-downton-place-2017/79024610/	n
8	http://www.nydailynews.com/new-york/brooklyn/2m-tax-incentives-brooklyn-company-moving-article-1.2502387	n
9	https://www.nytimes.com/2016/01/24/travel/green-hotels-new-york-city.html	n
10	http://www.nydailynews.com/new-york/citizenship-now/undocumented-immigrants-qualify-n-y-nursing-licens-article-1.2501306	n

Tagging the URLs

13. Now we're all set with the unwanted stories. Let's do the same thing with those stories we are interested in:

```
yes_params = {'consumer_key': CONSUMER_KEY,
  'access_token': ACCESS_TOKEN,
  'tag': 'y'}
yes_result = requests.post('https://getpocket.com/v3/get',
  data=yes_params)

yes_jf = json.loads(yes_result.text)
```

```

yes_jd = yes_jf['list']

yes_urls=[]
for i in yes_jd.values():
    yes_urls.append(i.get('resolved_url'))

yes_uf = pd.DataFrame(yes_urls, columns=['urls'])
yes_uf = yes_uf.assign(wanted = lambda x: 'y')

yes_uf

```

The preceding code results in the following output:

		urls	wanted
0		https://lifehacker.com/the-akrasia-effect-why-we-make-plans-and-dont-follow-t-1752799091	y
1		http://www.businessinsider.com/blizzard-for-the-ages-could-hit-northeast-2016-1	y
2		http://www.businessinsider.com/making-a-murderer-teresa-halbach-keys-2016-1	y
3		https://qz.com/585918/theres-a-simple-scientific-reason-that-diets-almost-never-work/	y
4		https://blogs.wsj.com/economics/2016/01/12/why-an-economist-plays-powerball/	y
5		https://www.fastcompany.com/3055560/the-future-of-work/what-the-interview-process-is-like-at-google-apple-amazon-and-other-tech-	y
6		https://www.usatoday.com/story/tech/news/2016/01/19/amazon-automatic-ordering-dash-replenishment-brother-glucose-meter-ge-washer/78983730/	y
7		http://newyork.cbslocal.com/2016/01/19/east-village-slash-attack-surveillance-video/	y
8		http://www.slideshare.net/ChristopherMoody3/word2vec-lda-and-introducing-a-new-hybrid-algorithm-lda2vec-57135994	y
9		https://qz.com/588869/how-can-you-better-embarrass-yourself-the-big-questions-that-can-guide-your-lifes-purpose/	y
10		http://www.bloombergview.com/articles/2016-01-06/how-economics-went-from-theory-to-data	y

Tagging the URLs of stories we are interested in

- Now that we have both types of stories for our training data, let's join them together into a single DataFrame:

```

df = pd.concat([yes_uf, no_uf])

df.dropna(inplace=True)

df

```

The preceding code results in the following output:

24	https://www.fastcompany.com/3055071/the-future-of-work/new-gmail-plug-in-highlights-words-and-phrases-that-undermine-your-messag	y
25	https://www.fastcodesign.com/3055079/info-graphic-of-the-day/how-complex-systems-and-chaos-theory-work-according-to-emoji	y
26	http://thenewinquiry.com/blogs/marginal-utility/wizards-of-like/	y
27	http://mobile.nytimes.com/2016/01/08/fashion/mens-style/new-york-bachelors-yearn-for-more.html	y
28	https://www.fastcompany.com/3055019/how-to-be-a-success-at-everything/the-secret-to-making-anxiety-work-in-your-favor	y
29	https://fivethirtyeight.com/features/how-msg-got-a-bad-rap-flawed-science-and-xenophobia/	y
...
58	https://github.com/mateogianolio/nlapack	n
59	http://www.telecomitalia.com/content/dam/telecomitalia/documents/innovation/it/eventi/progetto_unesco/4.introduction_to_CPU_development_China.pdf	n
60	http://grodola.blogspot.com/2016/01/netbsd-support-for-psutil.html	n
61	https://github.com/arkverse/ell	n
62	https://akrzemi1.wordpress.com/2016/01/16/a-customizable-framework/	n

Joining the URLs- both interested and not interested

Now that we're set with all our URLs and their corresponding tags in a single frame, we'll move on to downloading the HTML for each article. We'll use another free service for this, called Embedly.

Using the Embedly API to download story bodies

We have all the URLs for our stories, but, unfortunately, this isn't enough to train on; we'll need the full article body. This in itself could become a huge challenge if we want to roll our own scraper, especially if we are going to be pulling stories from dozens of sites. We would need to write code to target the article body while carefully avoiding all the other site gunk that surrounds it. Fortunately, as far as we are concerned, there are a number of free services that will do this for us. I'm going to be using Embedly to do this, but there are a number of other services that you could use instead.

The first step is to sign up for Embedly API access. You can do that at <https://app.embed.ly/signup>. It is a straightforward process. Once you confirm your registration, you will receive an API key. That's really all you'll need. You'll just use that key in your HTTP request. Let's do that now:

```
import urllib

EMBEDLY_KEY = 'your_embedly_api_key_here'

def get_html(x):
```

```

try:
    qurl = urllib.parse.quote(x)
    rhtml = requests.get('https://api.embedly.com/1/extract?url=' +
qurl + '&key=' + EMBEDLY_KEY)
    ctn = json.loads(rhtml.text).get('content')
except:
    return None
return ctn

```

The preceding code results in the following output:

	urls wanted	html
0	https://lifehacker.com/the-akrasia-effect-why-...	y <div>\n<p>Humans are prolific procrastinators....
1	http://www.businessinsider.com/blizzard-for-th...	y <div>\n<p><figure>\n<p>Each day, we <a href="http://eab.sa...
5	https://www.fastcompany.com/3055560/the-future...	y <div>\n<p>It's kind of a perfect storm.</p>\n<
6	https://www.usatoday.com/story/tech/news/2016/...	y <div>\nCorrections & Clarification: An...
7	http://newyork.cbslocal.com/2016/01/19/east-vi...	y <div>\n<p>\n</p><p>NEW YORK (CBSNewYor...
9	https://qz.com/588869/how-can-you-better-embar...	y <div>\n<p>One day, when my brother was 18, he ...
11	https://qz.com/597214/what-you-eat-could-be-st...	y <div>\n<p>The world is getting fatter, and <a ...
12	http://time.com/3746047/diet-soda-bad-belly-fat/	y <div>\n<p>Drinking diet soda doesn't do your w...

HTTP requests

And with that, we have the HTML of each story.

Since the content is embedded in HTML markup, and we want to feed plain text into our model, we'll use a parser to strip out the markup tags:

```

from bs4 import BeautifulSoup

def get_text(x):
    soup = BeautifulSoup(x, 'html5lib')
    text = soup.get_text()
    return text

df.loc[:, 'text'] = df['html'].map(get_text)

df

```

The preceding code results in the following output:

	urls	wanted	html	text
0	https://lifehacker.com/the-akrasia-effect-why...	y	<div>\n<p>Humans are prolific procrastinators....	\nHumans are prolific procrastinators. It's ea...
1	http://www.businessinsider.com/blizzard-for-th...	y	<div>\n<p><figure>\n<p>Each day, we <a href="http://eab.sa...	\nEach day, we make hundreds of food decision...
5	https://www.fastcompany.com/3055560/the-future...	y	<div>\n<p>It's kind of a perfect storm.</p>\n<	\nIt's kind of a perfect storm.\nA record 78% ...
6	https://www.usatoday.com/story/tech/news/2016/...	y	<div>\nCorrections & Clarification: An...	\nCorrections & Clarification: An earlier vers...
7	http://newyork.cbslocal.com/2016/01/19/east-vi...	y	<div>\n<p>\n</p>NEW YORK (CBSNewYor...	\n\nNEW YORK (CBSNewYork) - Police have releas...
9	https://qz.com/588869/how-can-you-better-embar...	y	<div>\n<p>One day, when my brother was 18, he ...	\nOne day, when my brother was 18, he waltzed ...
11	https://qz.com/597214/what-you-eat-could-be-st...	y	<div>\n<p>The world is getting fatter, and <a ...	\nThe world is getting fatter, and some say it...
12	http://time.com/3746047/diet-soda-bad-belly-fat/	y	<div>\n<p>Drinking diet soda doesn't do your w...	\nDrinking diet soda doesn't do your waistline...

And with that, we have our training set ready. We can now move on to a discussion of how to transform our text into something that a model can work with.

Basics of Natural Language Processing

If machine learning models only operate on numerical data, how can we transform our text into a numerical representation? That is exactly the focus of **Natural Language Processing (NLP)**. Let's take a brief look at how this is done.

We'll begin with a small corpus of three sentences:

1. The new kitten played with the other kittens
2. She ate lunch
3. She loved her kitten

We'll first convert our corpus into a **bag-of-words (BOW)** representation. We'll skip preprocessing for now. Converting our corpus into a BOW representation involves taking each word and its count to create what's called a **term-document matrix**. In a term-document matrix, each unique word is assigned to a column, and each document is assigned to a row. At the intersection of the two is the count:

Sr. no.	the	new	kitten	played	with	other	kittens	she	ate	lunch	loved	her
1	1	1	1	1	1	1	1	0	0	0	0	0
2	0	0	0	0	0	0	0	1	1	1	0	0
3	0	0	1	0	0	0	0	1	0	0	1	1

Notice that, for these three short sentences, we already have 12 features. As you might imagine, if we were dealing with actual documents, such as news articles or even books, the number of features would explode into the hundreds of thousands. To mitigate this explosion, we can take a number of steps to remove features that add little to no informational value to our analysis.

The first step we can take is to remove **stop words**. These are words that are so common that they typically tell you nothing about the content of the document. Common examples of English stop words are *the*, *is*, *at*, *which*, and *on*. We'll remove those, and recompute our term-document matrix:

Sr. no.	new	kitten	played	kittens	ate	lunch	loved
1	1	1	1	1	0	0	0
2	0	0	0	0	1	1	0
3	0	1	0	0	0	0	1

As you can see, the number of features was reduced from 12 to 7. This is great, but we can take it even further. We can perform **stemming** or **lemmatization** to reduce the features further. Notice that in our matrix, we have both *kitten* and *kittens*. By using stemming or lemmatization, we can consolidate that into just *kitten*:

Sr. no.	new	kitten	play	eat	lunch	love
1	1	2	1	0	0	0
2	0	0	0	1	1	0
3	0	1	0	0	0	1

Our new matrix consolidated *kittens* and *kitten*, but something else happened as well. We lost the suffixes to *played* and *loved*, and *ate* was transformed to *eat*. Why? This is what lemmatization does. If you remember your grade school grammar classes, we've gone from the inflectional form to the base form of the word. If that is lemmatization, what is stemming? Stemming has the same goal, but uses a less sophisticated approach. This approach can sometimes produce pseudo-words rather than the actual base form. For example, in lemmatization, if you were to reduce *ponies*, you would get *pony*, but with stemming, you'd get *poni*.

Let's now go further to apply another transformation to our matrix. So far, we have used a simple count of each word, but we can apply an algorithm that will act like a filter on our data to enhance the words that are unique to each document. This algorithm is called **term frequency-inverse document frequency (tf-idf)**.

We calculate this tf-idf ratio for each term in our matrix. Let's calculate it for a couple of examples. For the word *new* in document one, the term frequency is just the count, which is 1. The inverse document frequency is calculated as the log of the number of documents in the corpus over the number of documents the term appears in. For *new*, this is $\log(3/1)$, or .4471. So, for the complete tf-idf value, we have $tf * idf$, or, here, it is $1 \times .4471$, or just .4471. For the word *kitten* in document one, the tf-idf is $2 * \log(3/2)$, or .3522.

Completing this for the remainder of the terms and documents, we have the following:

Sr. no.	new	kitten	play	eat	lunch	love
1	.4471	.3522	.4471	0	0	0
2	0	0	0	.4471	.4471	0
3	0	.1761	0	0	0	.4471

Why do all of this? Let's say, for example, we have a corpus of documents about many subjects (medicine, computing, food, animals, and so on) and we want to classify them into topics. Very few documents would contain the word *sphygmomanometer*, which is the device used to measure blood pressure; and all the documents that did would likely concern the topic of medicine. And obviously, the more times this word appears in a document, the more likely it is to be about medicine. So a term that occurs rarely across our entire corpus, but that is present many times in a document, makes it likely that this term is tied closely to the topic of that document. In this way, documents can be said to be represented by those terms with high tf-idf values.

With the help of this framework, we'll now convert our training set into a tf-idf matrix:

```
from sklearn.feature_extraction.text import TfidfVectorizer

vect = TfidfVectorizer(ngram_range=(1, 3), stop_words='english', min_df=3)

tv = vect.fit_transform(df['text'])
```

With those three lines, we have converted all our documents into a tf-idf vector. We passed in a number of parameters: `ngram_range`, `stop_words`, and `min_df`. Let's discuss each.

First, `ngram_range` is how the document is tokenized. In our previous examples, we used each word as a token, but here, we are using all one- to three-word sequences as tokens. Let's take our second sentence, *She ate lunch*. We'll ignore stop words for the moment. The n-grams for this sentence would be: *she*, *she ate*, *she ate lunch*, *ate*, *ate lunch*, and *lunch*.

Next, we have `stop_words`. We pass in `english` for this to remove all the English stop words. As discussed previously, this removes all terms that lack informational content.

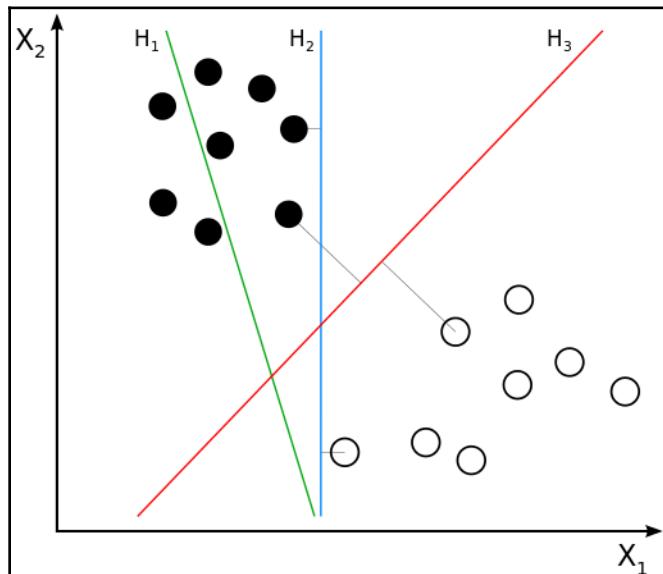
And finally, we have `min_df`. This removes all words from consideration that don't appear in at least three documents. Adding this removes very rare terms and reduces the size of our matrix.

Now that our article corpus is in a workable numerical format, we'll move on to feeding it to our classifier.

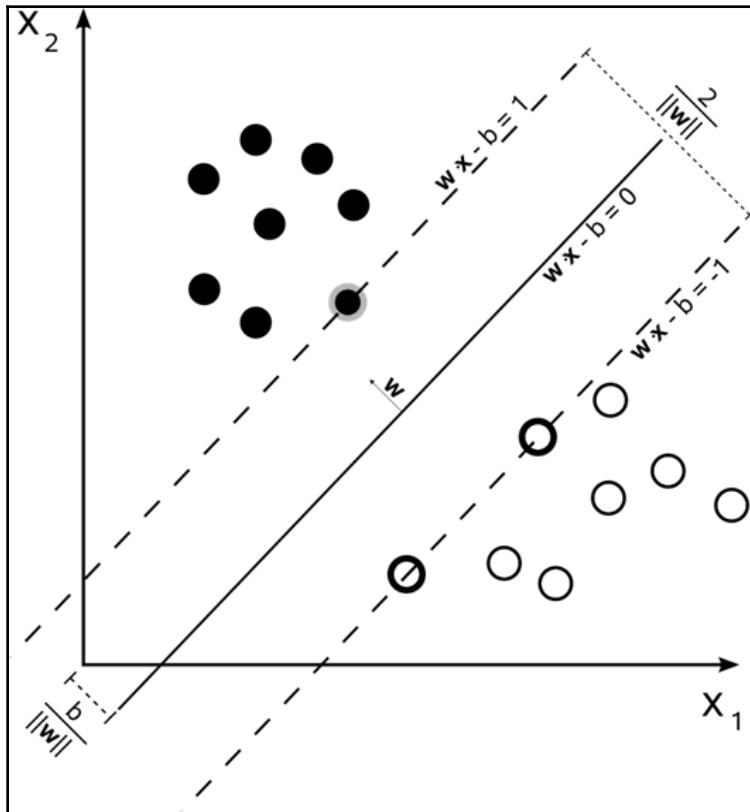
Support Vector Machines

We're going to be utilizing a new classifier in this chapter, a linear **Support Vector Machine (SVM)**. An SVM is an algorithm that attempts to linearly separate data points into classes using a **maximum-margin hyperplane**. That's a mouthful, so let's look at what it really means.

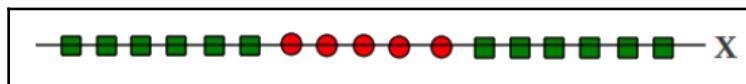
Suppose we have two classes of data, and we want to separate them with a line. (We'll just deal with two features, or dimensions, here.) What is the most effective way to place that line? Lets have a look at an illustration:



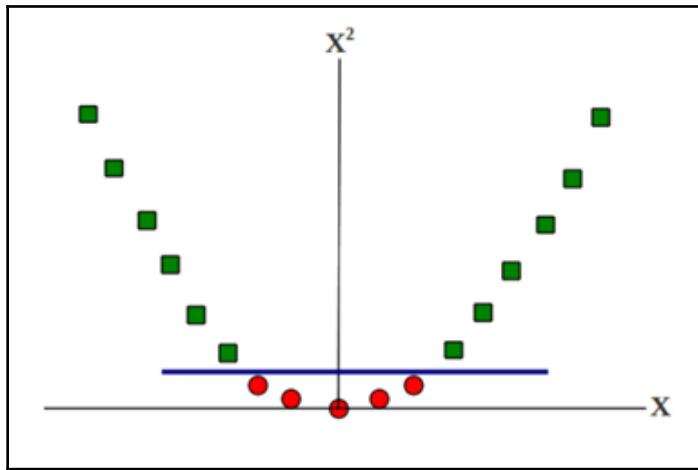
In the preceding diagram, line H_1 does not effectively discriminate between the two classes, so we can eliminate that one. Line H_2 is able to discriminate between them cleanly, but H_3 is the maximum-margin line. This means that the line is centered between the two nearest points of each class, which are known as the **support vectors**. These can be seen as the dotted lines in the following diagram:



What if the data isn't able to be separated into classes so neatly? What if there is an overlap between the points? In that situation, there are still options. One is to use what's called a **soft-margin SVM**. This formulation still maximizes the margin, but with the trade-off being a penalty for points that fall on the wrong side of the margin. The other option is to use what's called the **kernel trick**. This method transforms the data into a higher dimensional space where the data can be linearly separated. An example is provided here:



The two-dimensional representation is as follows:



We have taken a one-dimensional feature space and mapped it onto a two-dimensional feature space. The mapping simply takes each x value and maps it to x, x^2 . Doing so allows us to add a linear separating plane.

With that covered, let's now feed our tf-idf matrix into our SVM:

```
from sklearn.svm import LinearSVC  
  
clf = LinearSVC()  
model = clf.fit(tv, df['wanted'])
```

`tv` is our matrix, and `df['wanted']` is our list of labels. Remember this is either `y` or `n`, denoting whether we are interested in the article. Once that runs, our model is trained.

One thing we aren't doing in this chapter is formally evaluating our model. You should almost always have a hold-out set to evaluate your model against, but because we are going to be continuously updating our model, and evaluating it daily, we'll skip that step for this chapter. Just remember that this is generally a terrible idea.

Let's now move on to setting up our daily feed of news items.

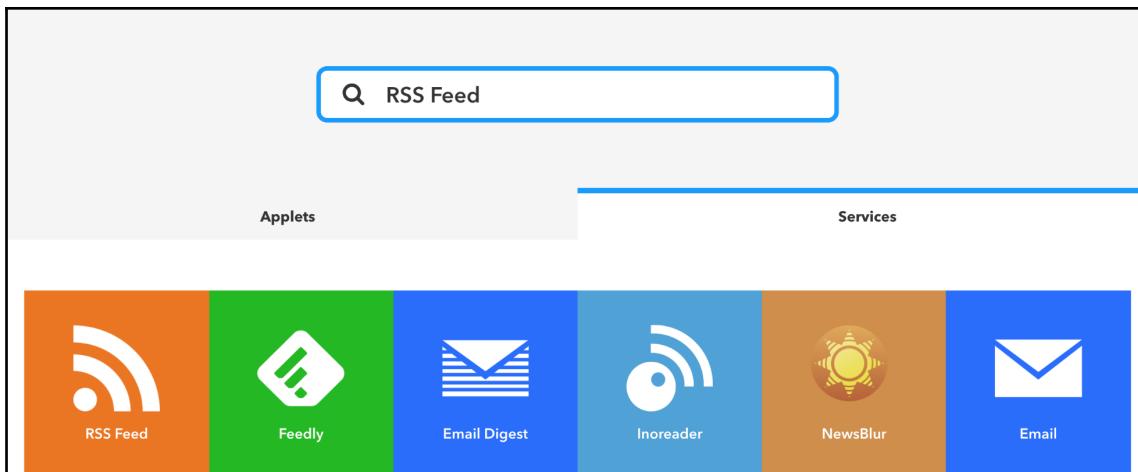
IFTTT integration with feeds, Google Sheets, and email

We used Pocket to build our training set, but now we need a streaming feed of articles to run our model against. To set this up, we'll use IFTTT once again, as well as Google Sheets, and a Python library that will allow us to work with Google Sheets.

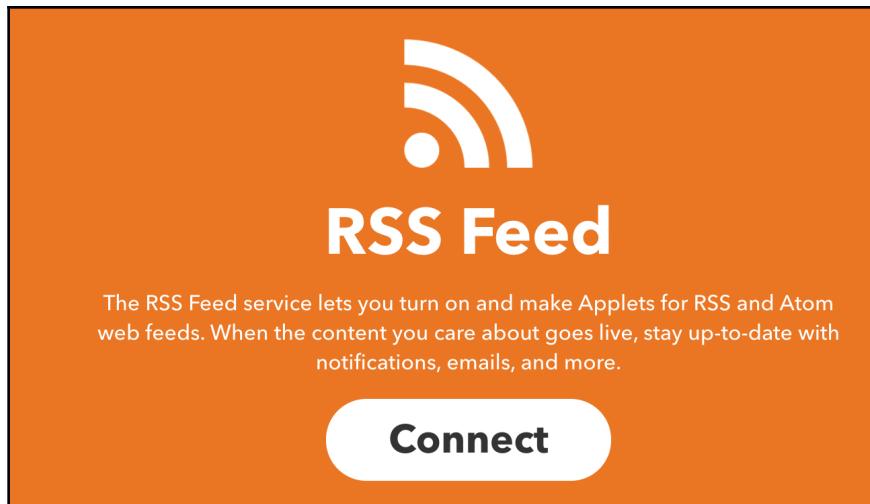
Setting up news feeds and Google Sheets through IFTTT

Hopefully, you have an IFTTT account set up at this point, but if not, go ahead and set that up now. Once that is done, you'll need to set up integration with feeds and with Google Sheets:

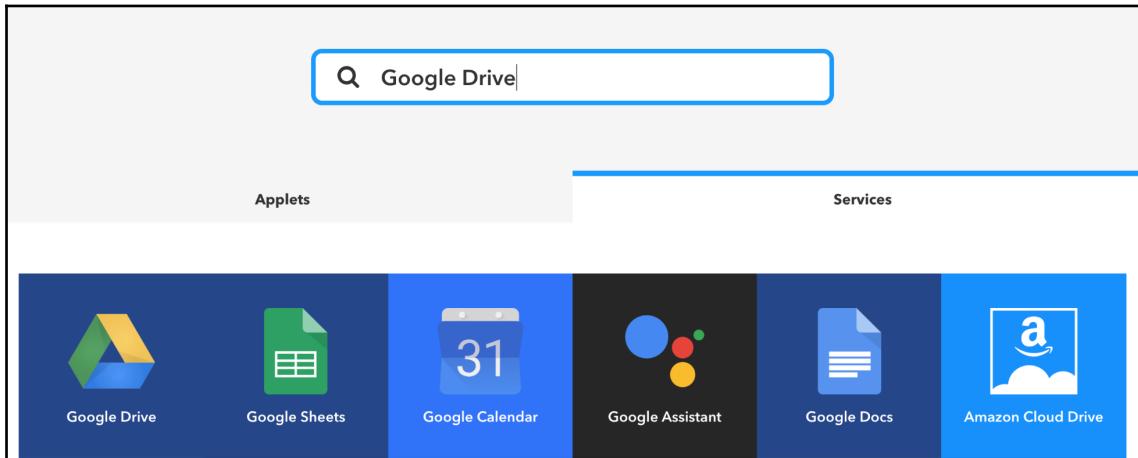
1. First, search for feeds in the search box on the home page, then click on **Services**, and click to set that up:



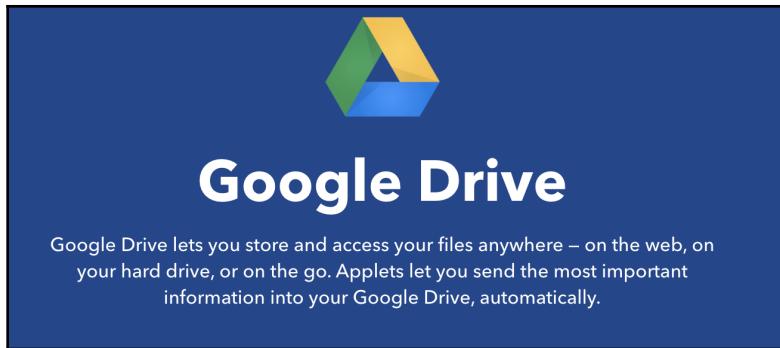
2. You'll just need to click **Connect**:



3. Next, search for Google Drive under **Services**:



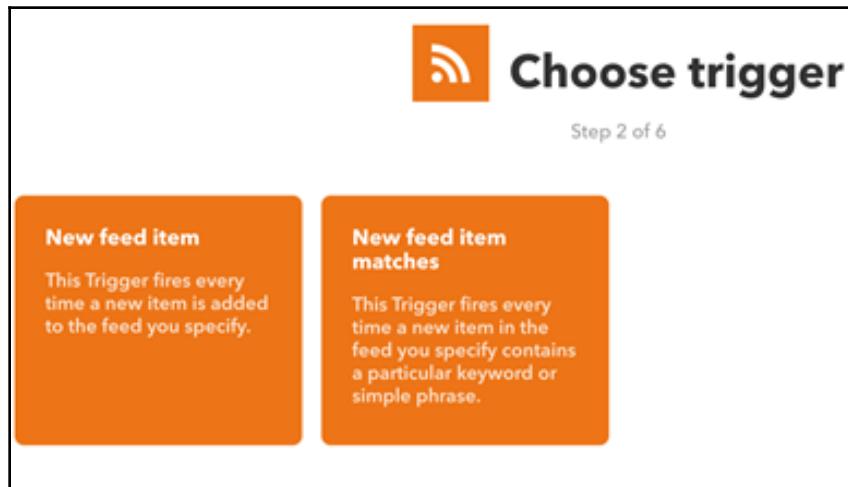
4. Click on that. It should take you to a page where you select the Google account you want to connect to. Choose the account and then click **Allow** to enable IFTTT to access your Google Drive account. Once that's done, you should see the following:



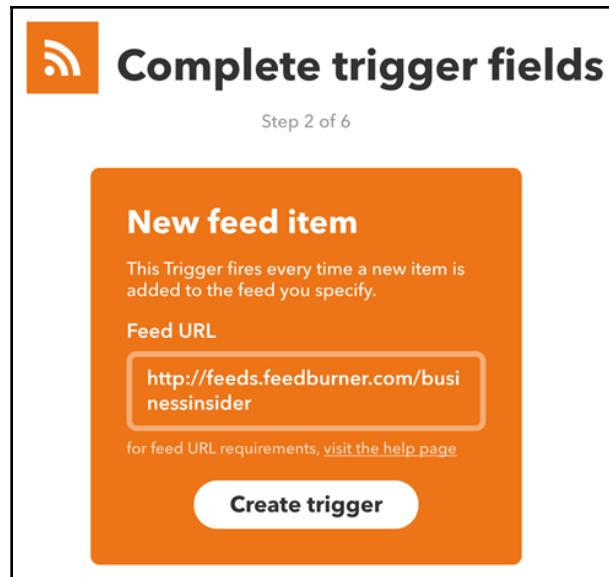
- Now, with our channels connected, we can set up our feed. Click on **New Applet** in the dropdown under your username in the right-hand corner. This will bring you here:



- Click on **+this**. Search for **RSS Feed**, and then click on that. That should bring you here:



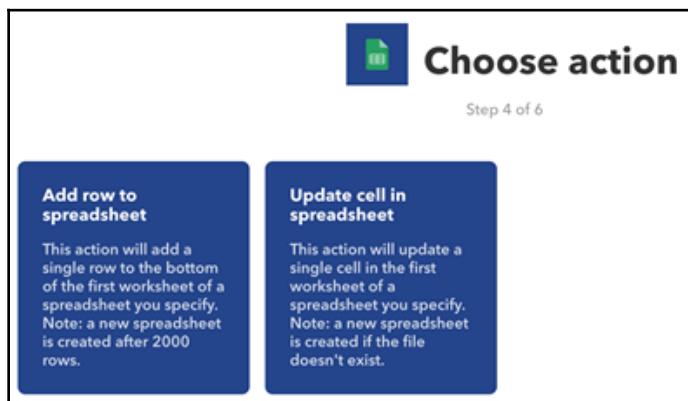
7. From here, click on **New feed item**:



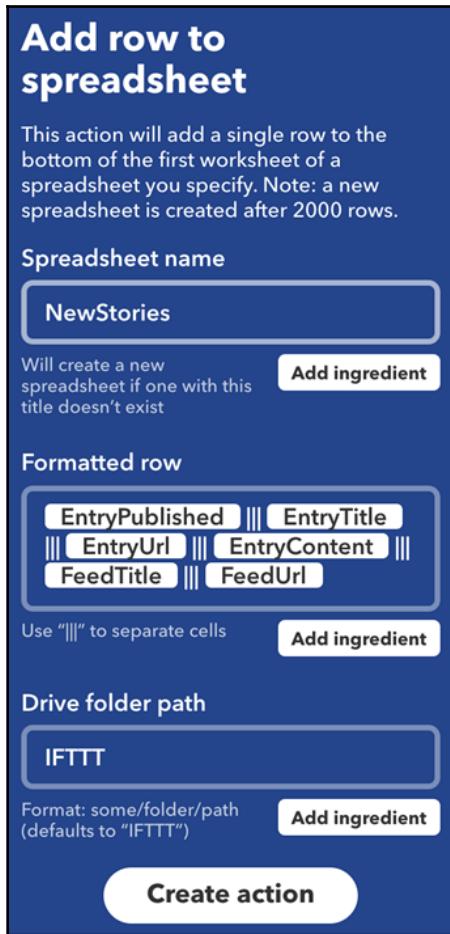
8. Then, add the URL to the box and click **Create trigger**. Once that is done, you'll be brought back to add the **+that** action:

if then

9. Click on **+that**, search for Sheets, and then click on its icon. Once that is done, you'll find yourself here:



10. We want our news items to flow into a Google Drive spreadsheet, so click on **Add row to spreadsheet**. You'll then have an opportunity to customize the spreadsheet:



I gave the spreadsheet the name `NewStories`, and placed it in a Google Drive folder called `IFTTT`. Click **Create Action** to finish the recipe, and soon you'll start seeing news items flow into your Google Drive spreadsheet. Note that it will only add new items as they come in, not items that existed at the time you created the sheet. I recommend adding a number of feeds. You will need to create individual recipes for each. It is best if you add feeds for the sites that are in your training set, in other words, the ones you saved with Pocket.

Give those stories a day or two to build up in the sheet, and then it should look something like this:

	B	C	D	E	F
165	Audi Unveils Lunar Quattro, A New Robotic Rover For Moon Exploration - Immortal News	http://news.google.com	<table border="0" cellpadding="2" cellspacing="0" data-bbox="102 181 892 196">Science - Google	http://news.google.com/news?hl=en	
166	Astronomers may have found most powerful supernova - CNN	http://news.google.com	<table border="0" cellpadding="2" cellspacing="0" data-bbox="102 196 892 211">Science - Google	http://news.google.com/news?hl=en	
167	What Twitter Is Saying About the Democratic Debate	http://www.slate.com	<p>On Sunday, the Democrats will hold their	Slate Articles	http://www.slate.com/articles.teaser
168	The Downton Abbey Podcast 	http://www.slate.com	<p>This is a free preview of Slate's political coverage.	Slate Articles	http://www.slate.com/articles.teaser
169	Who Won the Democratic Debate?	http://www.slate.com	<p>Now that the fourth Democratic presidential	Slate Articles	http://www.slate.com/articles.teaser
170	Hillary Clinton Won Sunday Night's Debate	http://www.slate.com	<p>Hillary Clinton's superb debate performance	Slate Articles	http://www.slate.com/articles.teaser
171	America's Other Original Sin	http://www.slate.com	<p>Here are three scenes from the history of	Slate Articles	http://www.slate.com/articles.teaser
172	Native American Slaves in New France	http://www.slate.com	<p>Between 1660 and 1760, French colonists	Slate Articles	http://www.slate.com/articles.teaser
173	Just a Guy at the Bar	http://www.slate.com	<p>A lot of my friends have stopped watching	Slate Articles	http://www.slate.com/articles.teaser
174	The Color of Surveillance	http://www.slate.com	<p>The FBI has a lead. A prominent religious	Slate Articles	http://www.slate.com/articles.teaser
175	Name That Rare Aquatic Creature	http://www.slate.com	<p>In the video above, three scientists with the	Slate Articles	http://www.slate.com/articles.teaser
176	Listen to Kanye West's 'No More Parties in LA' Featuring Kendrick Lamar - TIME	http://news.google.com	<table border="0" cellpadding="2" cellspacing="0" data-bbox="102 331 892 346">Entertainment - Google	http://news.google.com/news?hl=en	
177	TV highlights: 'Marvel's Agent Carter' returns for Season 2 on ABC - Washington Post	http://news.google.com	<table border="0" cellpadding="2" cellspacing="0" data-bbox="102 346 892 361">Entertainment - Google	http://news.google.com/news?hl=en	
178	Winslet tips Titanic co-star DiCaprio for Oscar - The Nation	http://news.google.com	<table border="0" cellpadding="2" cellspacing="0" data-bbox="102 361 892 376">Entertainment - Google	http://news.google.com/news?hl=en	
179	Ta-Nehisi Coates among finalists for book critics awards - San Francisco Chronicle	http://news.google.com	<table border="0" cellpadding="2" cellspacing="0" data-bbox="102 376 892 391">Entertainment - Google	http://news.google.com/news?hl=en	

Fortunately, the full article HTML body is included. This means we won't have to use Embedly to download it for each article. We will still need to download the articles from Google Sheets, and then process the text to strip out the HTML tags, but this can all be done rather easily.

To pull down the articles, we'll use a Python library called `gspread`. This can be pip installed. Once that is installed, you need to follow the direction for setting up **OAuth 2**. That can be found at <http://gspread.readthedocs.org/en/latest/oauth2.html>. You will end up downloading a JSON credentials file. It is critical that, once you have that file, you find the email address in it with the `client_email` key. You then need to share the `NewStories` spreadsheet you are sending the stories to with that email. Just click on the blue **Share** button in the upper-right corner of the sheet, and paste the email in there. You will end up receiving a *failed to send* message in your Gmail account, but that is expected. Make sure to swap in your path to the file and the name of the file in the following code:

```
import gspread

from oauth2client.service_account import ServiceAccountCredentials
JSON_API_KEY = 'the/path/to/your/json_api_key/here'

scope = ['https://spreadsheets.google.com/feeds',
         'https://www.googleapis.com/auth/drive']

credentials =
ServiceAccountCredentials.from_json_keyfile_name(JSON_API_KEY, scope)
gc = gspread.authorize(credentials)
```

Now, if everything went well, it should run without errors. Next, you can download the stories:

```
ws = gc.open("NewStories")
sh = ws.sheet1

zd = list(zip(sh.col_values(2), sh.col_values(3), sh.col_values(4)))

zf = pd.DataFrame(zd, columns=['title', 'urls', 'html'])
zf.replace('', pd.np.nan, inplace=True)
zf.dropna(inplace=True)

zf
```

The preceding code results in the following output:

	title	urls	html
0	The man who owned the world: David Bowie made ...	http://news.google.com/news/url?sa=t&fd=R&ct2=...	<table border="0" cellpadding="2" cellspacing="...
1	Netflix To Ramp Up Originals Targeting Kids - ...	http://news.google.com/news/url?sa=t&fd=R&ct2=...	<table border="0" cellpadding="2" cellspacing="...
2	Nostalgia powers Netflix's 'Fuller House' retu...	http://news.google.com/news/url?sa=t&fd=R&ct2=...	<table border="0" cellpadding="2" cellspacing="...
3	High School Musical cast to reunite for 10-yea...	http://news.google.com/news/url?sa=t&fd=R&ct2=...	<table border="0" cellpadding="2" cellspacing="...
4	Unbreakable Kimmy Schmidt renewed for season 3...	http://news.google.com/news/url?sa=t&fd=R&ct2=...	<table border="0" cellpadding="2" cellspacing="...
5	David Bowie's 'Blackstar' Becomes His First No...	http://news.google.com/news/url?sa=t&fd=R&ct2=...	<table border="0" cellpadding="2" cellspacing="...
6	5 reasons why birthday girl Betty White had a ...	http://news.google.com/news/url?sa=t&fd=R&ct2=...	<table border="0" cellpadding="2" cellspacing="...
7	Straight Outta Compton' Producer Calls Oscar N...	http://news.google.com/news/url?sa=t&fd=R&ct2=...	<table border="0" cellpadding="2" cellspacing="...
8	SXSW Co-Founder Confesses to 'David Bowie' Str...	http://news.google.com/news/url?sa=t&fd=R&ct2=...	<table border="0" cellpadding="2" cellspacing="...
9	Ted Sarandos Blasts NBC's Netflix Ratings Info...	http://news.google.com/news/url?sa=t&fd=R&ct2=...	<table border="0" cellpadding="2" cellspacing="...
10	Critics' Choice Awards: What will happen when ...	http://news.google.com/news/url?sa=t&fd=R&ct2=...	<table border="0" cellpadding="2" cellspacing="...

With that, we downloaded all of the articles from our feed and placed them into a DataFrame. We now need to strip out the HTML tags. We can use the function we used earlier to retrieve the text. We'll then transform it using our tf-idf vectorizer:

```
zf.loc[:, 'text'] = zf['html'].map(get_text)

zf.reset_index(drop=True, inplace=True)

test_matrix = vect.transform(zf['text'])

test_matrix
```

The preceding code results in the following output:

```
<902x3790 sparse matrix of type '<class 'numpy.float64'>'  
with 45509 stored elements in Compressed Sparse Row format>
```

Here, we see that our vectorization was successful. Let's now pass it into our model to get back the results:

```
results = pd.DataFrame(model.predict(test_matrix), columns=['wanted'])  
  
results
```

The preceding code results in the following output:

wanted	
0	n
1	n
2	n
3	n
4	n
5	n
6	n
7	n
8	n
9	n
10	n

We see here that we have results for each of the stories. Let's now join them with the stories themselves so that we can evaluate the results:

```
rez = pd.merge(results, zf, left_index=True, right_index=True)  
  
rez
```

The preceding code results in the following output:

wanted		title	urls	html	text
0	n	The man who owned the world: David Bowie made ...	http://news.google.com/news/url?sa=t&fd=R&ct2=...	<table border="0" cellpadding="2" cellspacing="...	SalonThe man who owned the world: David Bowie ...
1	n	Netflix To Ramp Up Originals Targeting Kids - ...	http://news.google.com/news/url?sa=t&fd=R&ct2=...	<table border="0" cellpadding="2" cellspacing="...	Wall Street JournalNetflix To Ramp Up Original...
2	n	Nostalgia powers Netflix's 'Fuller House' retu...	http://news.google.com/news/url?sa=t&fd=R&ct2=...	<table border="0" cellpadding="2" cellspacing="...	USA TODAYNostalgia powers Netflix's 'Fuller Ho...
3	n	High School Musical cast to reunite for 10-yea...	http://news.google.com/news/url?sa=t&fd=R&ct2=...	<table border="0" cellpadding="2" cellspacing="...	Entertainment WeeklyHigh School Musical cast t...
4	n	Unbreakable Kimmy Schmidt renewed for season 3...	http://news.google.com/news/url?sa=t&fd=R&ct2=...	<table border="0" cellpadding="2" cellspacing="...	Entertainment WeeklyUnbreakable Kimmy Schmidt ...

At this point, we can improve the model by going through the results and correcting the errors. You'll need to do this for yourself, but here is how I made changes to my own:

```
change_to_no = [130, 145, 148, 163, 178, 199, 219, 222, 223, 226, 235, 279,
348, 357, 427, 440, 542, 544, 546, 568, 614, 619, 660, 668, 679, 686, 740,
829]

change_to_yes = [0, 9, 29, 35, 42, 71, 110, 190, 319, 335, 344, 371, 385,
399, 408, 409, 422, 472, 520, 534, 672]

for i in rez.iloc[change_to_yes].index:
    rez.iloc[i]['wanted'] = 'y'

for i in rez.iloc[change_to_no].index:
    rez.iloc[i]['wanted'] = 'n'

rez
```

The preceding code results in the following output:

wanted		title	urls	html	text
0	n	The man who owned the world: David Bowie made ...	http://news.google.com/news/url?sa=t&fd=R&ct2=...	<table border="0" cellpadding="2" cellspacing="...	SalonThe man who owned the world: David Bowie ...
1	n	Netflix To Ramp Up Originals Targeting Kids - ...	http://news.google.com/news/url?sa=t&fd=R&ct2=...	<table border="0" cellpadding="2" cellspacing="...	Wall Street JournalNetflix To Ramp Up Original...
2	n	Nostalgia powers Netflix's 'Fuller House' retu...	http://news.google.com/news/url?sa=t&fd=R&ct2=...	<table border="0" cellpadding="2" cellspacing="...	USA TODAYNostalgia powers Netflix's 'Fuller Ho...
3	n	High School Musical cast to reunite for 10-yea...	http://news.google.com/news/url?sa=t&fd=R&ct2=...	<table border="0" cellpadding="2" cellspacing="...	Entertainment WeeklyHigh School Musical cast t...
4	n	Unbreakable Kimmy Schmidt renewed for season 3...	http://news.google.com/news/url?sa=t&fd=R&ct2=...	<table border="0" cellpadding="2" cellspacing="...	Entertainment WeeklyUnbreakable Kimmy Schmidt ...

This may look like a lot of changes, but of the over 900 articles evaluated, I had to change very few. By making these corrections, we can now feed this back into our model to improve it even more. Let's add these results to our earlier training data and then rebuild the model:

```
combined = pd.concat([df[['wanted', 'text']], rez[['wanted', 'text']]])  
  
combined
```

The preceding code results in the following output:

	wanted	text
0	y	\nHumans are prolific procrastinators. It's ea...
1	y	\n Getty Images/Mario Tama A monster of a snow...
2	y	\nWhen the podcast "Serial" went viral, fans o...
3	y	\nEach day, we make hundreds of food decision...
5	y	\nIt's kind of a perfect storm.\nA record 78% ...
6	y	\nCorrections & Clarification: An earlier vers...
7	y	\n\nNEW YORK (CBSNewYork) - Police have releas...
9	y	\nOne day, when my brother was 18, he waltzed ...
11	y	\nThe world is getting fatter, and some say it...
12	y	\nDrinking diet soda doesn't do your waistline...

Retrain the model with following code:

```
tvcomb = vect.fit_transform(combined['text'], combined['wanted'])  
  
model = clf.fit(tvcomb, combined['wanted'])
```

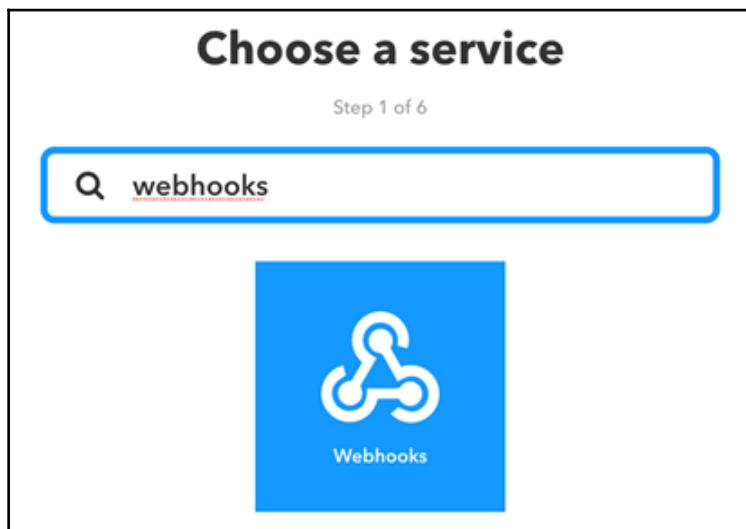
Now we have retrained our model with all the available data. You may want to do this a number of times as you get more results over the days and weeks. The more you add, the better your results will be.

We'll assume you have a well-trained model at this point, and are ready to begin using it. Let's now see how we can deploy this to set up a personalized news feed.

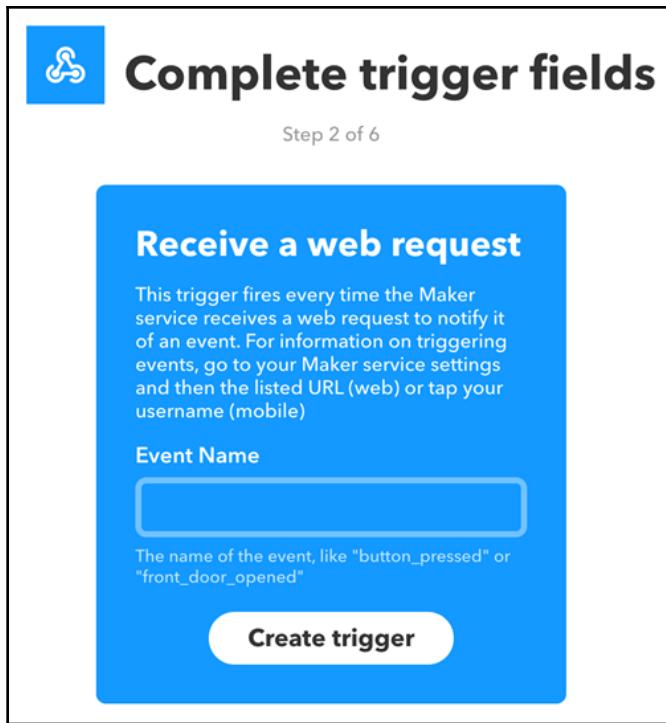
Setting up your daily personal newsletter

In order to set up a personal email with news stories, we're going to utilize IFTTT again. As before, in [Chapter 3, Build an App to Find Cheap Airfares](#), we'll use the Webhooks channel to send a POST request. But this time, the payload will be our news stories. If you haven't set up the Webhooks channel, do so now. Instructions can be found in [Chapter 3, Build an App to Find Cheap Airfares](#). You should also set up the Gmail channel. Once that is complete, we'll add a recipe to combine the two. Follow the steps to set up IFTTT:

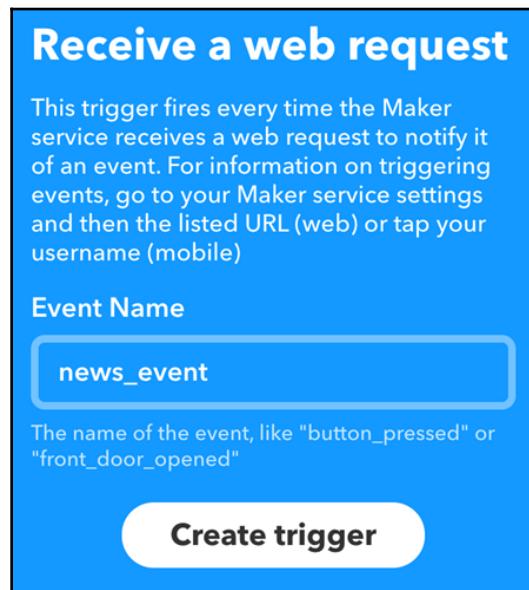
1. First, click **New Applet** from the IFTTT home page and then click **+this**. Then, search for the Webhooks channel:



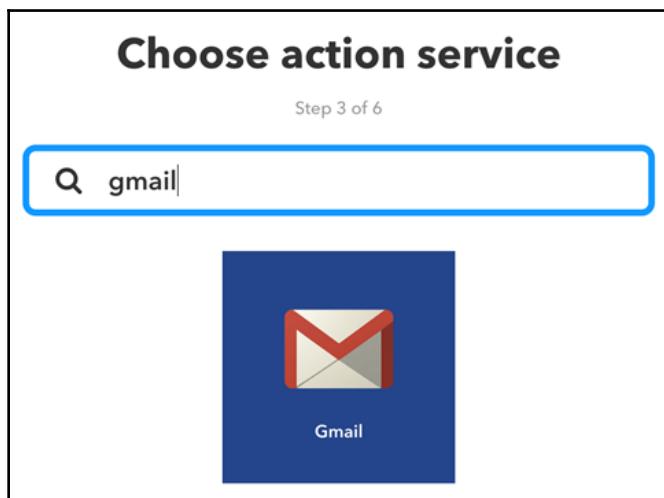
2. Select that, and then select **Receive a web request**:



3. Then, give the request a name. I'm using news_event:



4. Finish by clicking **Create trigger**. Next, click on **+that** to set up the email piece. Search for Gmail and click on that:



- Once you have clicked **Gmail**, click **Send yourself an email**. From there, you can customize your email message:



Input a subject line, and include {{Value1}} in the email body. We will pass our story title and link into this with our POST request. Click on **Create action** and then **Finish** to finalize it.

Now, we're ready to generate the script that will run on a schedule, automatically sending us articles of interest. We're going to create a separate script for this, but one last thing we need to do in our existing code is serialize our vectorizer and our model, as demonstrated in the following code block:

```
import pickle

pickle.dump(model, open(r'/input/a/path/here/to/news_model_pickle.p',
'wb'))

pickle.dump(vect, open(r'/input/a/path/here/to/news_vect_pickle.p', 'wb'))
```

With that, we have saved everything we need from our model. In our new script, we will read those in to generate our new predictions. We're going to use the same scheduling library to run the code as we used in [Chapter 3, Build an App to Find Cheap Airfares](#). Putting it all together, we have the following script:

```
import pandas as pd
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.svm import LinearSVC
import schedule
import time
import pickle
import json
import gspread
from oauth2client.service_account import ServiceAccountCredentials
import requests
from bs4 import BeautifulSoup

def fetch_news():

    try:
        vect = pickle.load(open(r'/your/path/to/news_vect_pickle.p', 'rb'))
        model = pickle.load(open(r'/your/path/to/news_model_pickle.p',
'rb'))

        JSON_API_KEY = r'/your/path/to/API KEY.json'

        scope = ['https://spreadsheets.google.com/feeds',
'https://www.googleapis.com/auth/drive']

        credentials =
ServiceAccountCredentials.from_json_keyfile_name(JSON_API_KEY, scope)
        gc = gspread.authorize(credentials)

        ws = gc.open("NewStories")
```

```
sh = ws.sheet1
zd = list(zip(sh.col_values(2), sh.col_values(3), sh.col_values(4)))
zf = pd.DataFrame(zd, columns=['title', 'urls', 'html'])
zf.replace('', pd.np.nan, inplace=True)
zf.dropna(inplace=True)

def get_text(x):
    soup = BeautifulSoup(x, 'html5lib')
    text = soup.get_text()
    return text

zf.loc[:, 'text'] = zf['html'].map(get_text)

tv = vect.transform(zf['text'])
res = model.predict(tv)

rf = pd.DataFrame(res, columns=['wanted'])
rez = pd.merge(rf, zf, left_index=True, right_index=True)

rez = rez.iloc[:20,:]

news_str = ''
for t, u in zip(rez[rez['wanted']=='y']['title'],
rez[rez['wanted']=='y']['urls']):
    news_str = news_str + t + '\n' + u + '\n'

payload = {"value1" : news_str}
r =
requests.post('https://maker.ifttt.com/trigger/news_event/with/key/bNHFwiZx
0wMS7EnD425n3T', data=payload)

# clean up worksheet
lenv = len(sh.col_values(1))
cell_list = sh.range('A1:F' + str(lenv))
for cell in cell_list:
    cell.value = ""
sh.update_cells(cell_list)
print(r.text)

except:
    print('Action Failed')

schedule.every(480).minutes.do(fetch_news)

while 1:
    schedule.run_pending()
    time.sleep(1)
```

What this script will do is run every 4 hours, pull down the news stories from Google Sheets, run the stories through the model, generate an email by sending a POST request to IFTTT for those stories that are predicted to be of interest, and then, finally, it will clear out the stories in the spreadsheet so only new stories get sent in the next email.

Congratulations! You now have your own personalized news feed!

Summary

In this chapter, we've learned how to work with text data when training machine learning models. We've also learned the basics of NLP and of SVMs.

In the next chapter, we'll develop these skills further and attempt to predict what sort of content will go viral.

6

Predict whether Your Content Will Go Viral

Like many great things, this all begins with a bet. It was 2001, and Jonah Peretti, a graduate student at MIT at the time, was procrastinating. Instead of writing his thesis, he had decided to take up Nike on their offer to personalize a pair of sneakers. Under a recently launched program, anyone could do so from their website, NIKEiD. The only problem, at least from Nike's point of view, was that emblazoning them with the word *sweatshop*, as Peretti had requested, was a non-starter. Peretti, in a series of emails, demurred pointing out that in no way did the word fall into any of the categories of objectionable terms that would result in his personalization request being rejected.

Peretti, believing others might find the back-and-forth with Nike's customer service representatives amusing as well, forwarded them to a number of close friends. Within days, the emails had found their way into inboxes across the world. Major media outlets, such as Time, Salon, The Guardian, and even the Today Show, had picked up on it. Peretti was at the center of a viral sensation.

But the question that began nagging at Peretti was, could this sort of thing be replicated? His friend, Cameron Marlow, had been preparing to write his PhD thesis on viral phenomena, and was adamant that such things were far too complex for anyone to engineer. And it is here that the bet comes into play. Marlow wagered that Peretti could not repeat the success he had enjoyed with that original set of emails with Nike.

Fast forward 15 years, and Jonah Peretti leads the website whose name has become synonymous with virality—BuzzFeed. With more than 77 million unique visitors in 2015, it ranked higher than the New York Times in total reach. I think it's safe to say that Peretti won that bet.

But how exactly did Peretti do it? How did he piece together the secret formula for creating content that spreads like wildfire? In this chapter, we'll attempt to unravel some of these mysteries. We'll examine some of the most shared content and attempt to find the common elements that differentiate it from the content people were less willing to share.

The following topics will be covered in this chapter:

- What does research tell us about virality?
- Sourcing shared counts and content
- Exploring the features of shareability
- Building a predictive content scoring model

What does research tell us about virality?

Understanding sharing behavior is big business. As consumers become increasingly blind to traditional advertising year on year, the push is on to go beyond simple pitches to tell engaging stories. And increasingly, the success of these endeavors is measured in social shares. Why go to so much trouble? Because, as a brand, every share I receive represents another consumer I've reached—all without spending an additional cent.

Because of this value, several researchers have examined sharing behavior in the hope of understanding what motivates it. Among the reasons researchers have found are the following:

- To provide practical value to others (an altruistic motive)
- To associate ourselves with certain ideas and concepts (an identity motive)
- To bond with others around a common emotion (a communal motive)

With regard to the last motive, one particularly well-designed study looked at the 7,000 pieces of content from the New York Times to examine the effect of emotion on sharing. They found that simple emotional sentiment was not enough to explain sharing behavior, but when combined with emotional arousal, the explanatory power was greater.

For example, while sadness has a strong negative valence, it is considered to be a low arousal state. Anger, on the other hand, has a negative valence, which is paired with a high arousal state. As such, stories that sadden the reader tend to generate far fewer stories than anger-inducing stories. Is it any wonder then that much of the *fake news* that plays such a large part in politics these days comes in this form? Following image shows the same result:

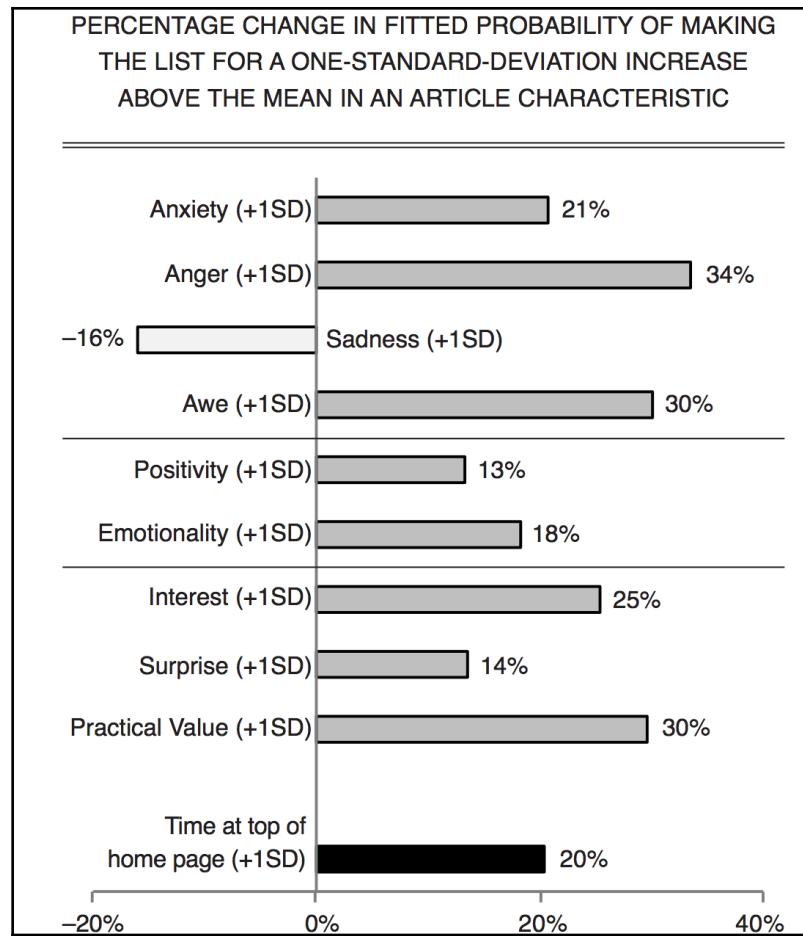


Figure taken from *What Makes Online Content Viral?* by Jonah Berger and Katherine L. Milkman, Journal of Marketing Research, available at: <http://jonahberger.com/wp-content/uploads/2013/02/ViralityB.pdf>

This covers the motivational aspects, but if we hold those factors constant, how do other attributes affect the virality of a piece of content? Some of these factors could include the following: headline wording, headline length, headline parts of speech, content length, social network of post, the topic, the timeliness of the subject matter, and so on. Without a doubt, a person could spend their entire life studying this phenomenon. For now, however, we'll just spend the next 30 or so pages doing so. From there, you can decide whether you'd like to take it further.

Sourcing shared counts and content

Before we can begin exploring which features make content shareable, we need to get our hands on a fair amount of content, as well as data on how often it's shared. Unfortunately, securing this type of data has gotten more difficult in the last few years. In fact, when the first edition of this book came out in 2016, this data was easily obtainable. But today, there appears to be no free sources of this type of data, though if you are willing to pay, you can still find it.

Fortunately for us, I have a dataset that was collected from a now defunct website, ruzzit.com. This site, when it was active, tracked the most shared content over time, which is exactly what we require for this project:

The screenshot shows the Ruzzit Beta website interface. At the top, there is a search bar with the placeholder "Search Keyword or Share a link". Below the search bar are four filter buttons: "Articles", "1 Year", "All social networks", and a question mark icon. The main content area displays two articles in a grid format. The first article, titled "Walker flap exposes GOP rift on Iowa's sacred status - CNN", features a photo of a woman speaking into a microphone. It has a score of 10.8M and the following metrics: 1.2M shares, 9.6M tweets, 1,911 likes, 0 views, 0 +1's, and 8,693 pins. The second article, titled "What Career Should You Actually Have? - BuzzFeed", features a photo of two women laughing. It has a score of 5.1M and the following metrics: 5.1M shares, 0 tweets, 1,559 likes, 0 views, 751 +1's, and 10.9k pins. Both articles include a "Read More" button and social sharing icons for Facebook, Twitter, LinkedIn, Google+, and StumbleUpon.

We'll begin by loading our imports into our notebook, as we always do, and then load in the data. This particular data is in the form of a JSON file. We can read it in using the pandas `read_json()` method, as demonstrated in the following code block:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
%matplotlib inline

dfc = pd.read_json('viral_dataset.json')
dfc.reset_index(drop=True, inplace=True)
dfc
```

The preceding code generates the following output:

	title	link	fb	lnkdn	pins	date
0	What Career Should You Actually Have?	http://www.ruzzit.com/en-US/Redirect/Link?media=1928328	5100000.0	1559	10900	2016-01-24 21:00:00
1	No One Showed Up To A Little Boy With Autism's Birthday, So His Mom Asked Facebook For Help	http://www.ruzzit.com/en-US/Redirect/Link?media=564259	400000.0	277	4	2015-02-23 15:39:00
2	Churro Ice Cream Sandwiches Are The Most Delicious Thing You Need In Your Life	http://www.ruzzit.com/en-US/Redirect/Link?media=1109973	402000.0	15	59	2015-07-11 00:36:00

Let's take a look at the columns of this dataset to better understand what we'll be working with:

```
dfc.columns
```

The preceding code generates the following output:

```
Index(['title', 'link', 'fb', 'lnkdn', 'pins', 'date', 'redirect',
       'pg_missing', 'img_link', 'json_data', 'site', 'img_count', 'entities',
       'html', 'text'],
      dtype='object')
```

Now, let's walk through what each of these columns represents:

- title: The title of the article
- link: The ruzzit.com link
- fb: The number of Facebook likes
- lnkdn: The number of LinkedIn shares
- pins: The number of Pinterest pins
- date: The date of the article
- redirect: The link to the original article

- pg_missing: A field that describes whether that page is available
- img_link: The link to the image for the article
- json_data: Additional data pertaining to the article
- site: The domain the article is hosted on
- img_count: The number of images contained in the article
- entities: Person-, place-, and thing-related features of the article
- html: The body of the article
- text: The text of the body of the article

Another feature that will be instructive is the word count of each article. We don't have that in our data currently, so let's go ahead and create a function that will provide this for us:

```
def get_word_count(x):
    if not x is None:
        return len(x.split(' '))
    else:
        return None

dfc['word_count'] = dfc['text'].map(get_word_count)
dfc
```

The preceding code generates the following output:

		html	text	word_count
0		None	None	NaN
1	<div>\n<p>\n</p><p> </p>>\n<div>\n<div>\n<div>\n<div>1....	\n\n\n\n\n1. This is Glenn Buratti, a 6-year-old from Osceola County, Florida. He has mild autism and epilepsy. Earlier this month, his mom Ashlee threw him a birthday party, inviting 16 childr...		79.0
2	<div>\n<p>\n</p><div>\n<div>\n<div>\n<div>1. Everyone loves ice cream. </div>\n<div>2. And of course churros are just as incredible. </div>\n<div>\n<h2>3. Individually, they make for a pretty tast...	\n\n\n\n\n1. Everyone loves ice cream. \n2. And of course churros are just as incredible. \n\n3. Individually, they make for a pretty tasty dessert...\n\n\n\n\n4. ...but have you ever thought ab...		78.0
3	<div>\n<h3>In 2013, Zerlina Maxwell ignited a firestorm of controversy when she strongly recommended we stop telling women how to not get raped. </h3>\n<p>Here are her words, from the <a href="htt...	\n\n\n\n\n1. In 2013, Zerlina Maxwell ignited a firestorm of controversy when she strongly recommended we stop telling women how to not get raped. \nHere are her words, from the transcript of her appearance ...		414.0

Let's add more features. We'll add the most prominent color of the first image on the page. The colors for each image are listed by RGB value in the JSON data, so we can extract it from there:

```
import matplotlib.colors as mpc

def get_rgb(x):
    try:
        if x.get('images'):
            main_color = x.get('images')[0].get('colors')[0].get('color')
```

```
        return main_color
    except:
        return None

def get_hex(x):
    try:
        if x.get('images'):
            main_color = x.get('images')[0].get('colors')[0].get('color')
            return mpc.rgb2hex([(x/255) for x in main_color])
    except:
        return None

dfc['main_hex'] = dfc['json_data'].map(get_hex)
dfc['main_rgb'] = dfc['json_data'].map(get_rgb)

dfc
```

The preceding code generates the following output:

word_count	main_hex	main_rgb
NaN	#b29e83	[178, 158, 131]
79.0	#0c1514	[12, 21, 20]
78.0	#c69e73	[198, 158, 115]
414.0	#e9eff4	[233, 239, 244]
NaN	#f3f4f0	[243, 244, 240]
NaN	#f1f9f3	[241, 249, 243]

We've pulled the most prominent color from the first image as an RGB value, but we have also transformed that into a hex value. We'll use that later when we examine the image colors.

With our data now ready, we can begin to perform our analysis. We're going to attempt to find what makes content highly shareable.

Exploring the features of shareability

The stories we have collected here represent roughly the 500 most shared pieces of content in 2015 and early 2016. We're going to try to deconstruct these articles to find the common traits that make them so shareable. We'll begin by looking at the image data.

Exploring image data

Let's begin by looking at the number of images included with each story. We'll run a value count and then plot the numbers:

```
dfc['img_count'].value_counts().to_frame('count')
```

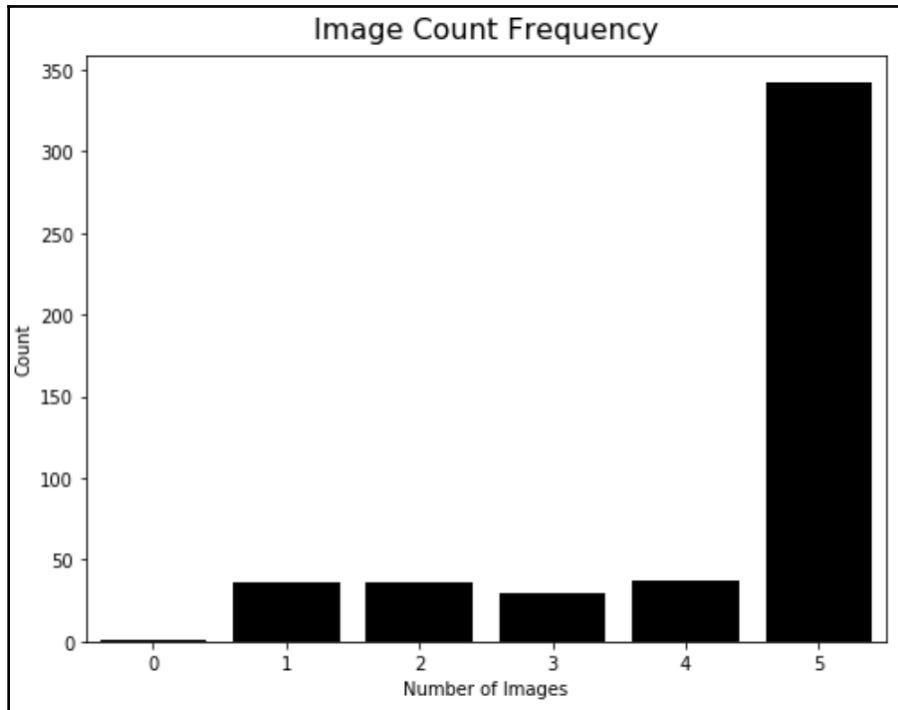
This should display an output similar to the following:

count	
5.0	342
4.0	37
1.0	36
2.0	36
3.0	30
0.0	1

Now, let's plot that same information:

```
fig, ax = plt.subplots(figsize=(8, 6))
y = dfc['img_count'].value_counts().sort_index()
x = y.sort_index().index
plt.bar(x, y, color='k', align='center')
plt.title('Image Count Frequency', fontsize=16, y=1.01)
ax.set_xlim(-.5, 5.5)
ax.set_ylabel('Count')
ax.set_xlabel('Number of Images')
```

This code generates the following output:



Already, I'm surprised by the numbers. The vast majority of stories have five pictures in them, while those stories that have either one or no pictures at all are quite rare.

Hence, we can see that people tend to share content with lots of images. Now, let's take a look at the most common colors in those images:

```
mci = dfc['main_hex'].value_counts().to_frame('count')

mci
```

This code generates the following output:

	count
#c4c4c4	3
#f8fbfa	3
#211537	2
#070603	2
#eaeef	2
#312c27	2
#f6fafb	2
#39546c	2
#c6b7b5	2
#f4f8f9	2
#3dd876	2

I don't know about you, but this isn't extremely helpful given that I don't see hex values as colors. We can, however, use a new feature in pandas called conditional formatting to help us out:

```
mci['color'] = ''  
  
def color_cells(x):  
    return 'background-color: ' + x.index  
  
mci.style.apply(color_cells, subset=['color'], axis=0)  
  
mci
```

The preceding code generates the following output:

	count	color
#c4c4c4	3	Light Gray
#f8fbfa	3	White
#211537	2	Dark Purple
#070603	2	Black
#eaeef	2	Light Gray
#312c27	2	Dark Gray
#f6fafb	2	White
#39546c	2	Dark Blue
#c6b7b5	2	Brownish Gray
#f4f8f9	2	White
#3dd876	2	Green

Clustering

This certainly helps, but the colors are so granular that we have over 450 unique colors in total. Let's use a bit of clustering to get this down to a more manageable range. Since we have the RGB values for each color, we can create a three-dimensional space to cluster them using the k-means algorithm. I won't go into the details of the algorithm here, but it is a fairly simple iterative algorithm based upon generating clusters by measuring the distance to centers and repeating. The algorithm does require us to select the k , or the number of clusters we expect. Because RGB ranges from 0 to 256, we'll use the square root of 256, which is 16. That should give us a manageable number while retaining the characteristics of our palette.

First, we'll split our RGB values into individual columns:

```
def get_csplit(x):
    try:
        return x[0], x[1], x[2]
    except:
        return None, None, None

dfc['reds'], dfc['greens'], dfc['blues'] =
zip(*dfc['main_rgb'].map(get_csplit))
```

Next, we'll use this to run our k-means model and retrieve the center values:

```
from sklearn.cluster import KMeans

clf = KMeans(n_clusters=16)
clf.fit(dfc[['reds', 'greens', 'blues']].dropna())

clusters = pd.DataFrame(clf.cluster_centers_, columns=['r', 'g', 'b'])

clusters
```

This generates the following output:

	r	g	b
0	200.200000	165.350000	139.050000
1	12.462500	13.700000	16.237500
2	213.942308	218.557692	216.211538
3	136.150000	116.500000	88.000000
4	146.911765	148.294118	139.235294
5	126.300000	61.100000	38.900000
6	75.323529	78.294118	83.323529
7	195.617647	192.970588	186.764706
8	1.600000	82.000000	156.200000
9	69.055556	58.611111	50.277778
10	30.859155	29.408451	35.169014

Now, we have the sixteen most popular dominant colors from the first image in each picture. Let's check whether they are using our pandas `DataFrame.style()` method and the function we created previously to color our cells. We'll need to set our index equal to the hex value of the three columns to use our `color_cells` function, so we'll do that as well:

```
def hexify(x):
    rgb = [round(x['r']), round(x['g']), round(x['b'])]
    hxc = mpc.rgb2hex([(x/255) for x in rgb])
    return hxc

clusters.index = clusters.apply(hexify, axis=1)
```

```
clusters['color'] = ''  
  
clusters.style.apply(color_cells, subset=['color'], axis=0)
```

This generates the following output:

	r	g	b	color
#c8a58b	200.2	165.35	139.05	
#0c0e10	12.4625	13.7	16.2375	
#d6dbd8	213.942	218.558	216.212	
#887458	136.15	116.5	88	
#93948b	146.912	148.294	139.235	
#7e3d27	126.3	61.1	38.9	
#4b4e53	75.3235	78.2941	83.3235	
#c4c1bb	195.618	192.971	186.765	
#02529c	1.6	82	156.2	
#453b32	69.0556	58.6111	50.2778	
#1f1d23	30.8592	29.4085	35.169	
#eff2f2	238.509	242.473	242.309	
#d93733	217.25	55.25	50.75	
#5a7389	90.2941	115.176	136.529	
#7caec5	124	174.375	196.875	
#3dd876	61	216	118	

So there you have it; those are the most common colors you will see (at least for the first image) in the most frequently shared content. This is a bit more on the drab side than I had expected as the first several all seem to be shades of beige and gray.

Now, let's move on and examine the headlines of our stories.

Exploring the headlines

Let's start by creating a function we can use to examine the most common tuples. We'll set it up so that we can use it later on the body text as well. We'll do this using the Python **Natural Language Toolkit (NLTK)** library. This can be pip installed if you don't have it currently:

```
from nltk.util import ngrams
from nltk.corpus import stopwords
import re

def get_word_stats(txt_series, n, rem_stops=False):
    txt_words = []
    txt_len = []
    for w in txt_series:
        if w is not None:
            if rem_stops == False:
                word_list = [x for x in ngrams(re.findall('[a-zA-Z0-9\\']+',
w.lower()), n)]
            else:
                word_list = [y for y in ngrams([x for x in re.findall('[a-zA-Z0-9\\']+',
w.lower())\n                                if x not in
stopwords.words('english')], n)]
            word_list_len = len(list(word_list))
            txt_words.extend(word_list)
            txt_len.append(word_list_len)
    return pd.Series(txt_words).value_counts().to_frame('count'),
pd.DataFrame(txt_len, columns=['count'])
```

There is a lot in there, so let's unpack it. We created a function that takes in a series, an integer, and a Boolean value. The integer determines the n we'll use for n-gram parsing, while the Boolean determines whether or not we exclude stop words. The function returns the number of tuples per row and the frequency for each tuple.

Let's run it on our headlines, while retaining the stop words. We'll begin with just single words:

```
hw,h1 = get_word_stats(dfc['title'], 1, 0)

h1
```

This generates the following output:

count	
0	6
1	18
2	14
3	16
4	11
5	11
6	14
7	11
8	10
9	6
10	12

Now, we have the word count for each headline. Let's see what the stats on this look like:

```
h1.describe()
```

This code generates the following output:

count	
count	482.000000
mean	10.948133
std	3.436294
min	1.000000
25%	9.000000
50%	11.000000
75%	13.000000
max	25.000000

We can see that the median headline length for our viral stories comes in at exactly 11 words. Let's take a look at the most frequently used words:

	count
(the,)	144
(to,)	130
(a,)	122
(of,)	86
(in,)	85
(you,)	68
(is,)	68
(and,)	65
(that,)	43
(will,)	42
(for,)	41

That is not exactly useful, but is in keeping with what we might expect. Now, let's take a look at the same information for bi-grams:

```
hw,hl = get_word_stats(dfc['title'], 2, 0)

hw
```

This generates the following output:

	count
(pictures, that)	11
(that, will)	9
(dies, at)	8
(of, the)	8
(in, a)	8
(in, the)	8
(people, who)	8
(how, to)	7
(no, one)	7
(donald, trump)	7

This is definitely more interesting. We can start to see some of the components of the headlines over and over again. The two that stand out are (donald, trump) and (dies, at). Trump makes sense as he said some headline-grabbing statements during the election, but I was surprised by the *dies* headlines. I took a look at the headlines, and apparently a number of high-profile people died in the year in question, so that also makes sense.

Now, let's run this with the stop words removed:

```
hw,hl = get_word_stats(dfc['title'], 2, 1)

hw
```

This generates the following output:

	count
(donald, trump)	7
(community, post)	5
(year, old)	5
(cnn, com)	4
(things, you'll)	4
(white, people)	4
(19, things)	4
(fox, news)	4
(gay, marriage)	3
(you'll, understand)	3
(dies, 83)	3
(pictures, real)	3

Again, we can see many things we might expect. It looks like if we changed how we parsed numbers (replacing each of them with a single identifier like number), we would likely see more of these bubble up. I'll leave that as an exercise to the reader, if you'd like to attempt that.

Now, let's take a look at tri-grams:

```
hw,hl = get_word_stats(dfc['title'], 3, 0)
```

This code generates the following output:

	count
(that, will, make)	4
(for, people, who)	4
(24, pictures, that)	3
(that, are, too)	3
(with, down, syndrome)	3
(a, woman, is)	3
(you, need, to)	3
(pictures, that, will)	3
(too, real, for)	3
(the, ring, of)	3
(pictures, that, are)	3
(are, too, real)	3
(dies, at, 83)	3

It seems that the more words we include, the more the headlines come to resemble the classic BuzzFeed prototype. In fact, let's see whether that's the case. We haven't looked at which sites produce the most viral stories; let's see whether BuzzFeed leads the charts:

```
dfc['site'].value_counts().to_frame()
```

This generates the following output:

site	
BuzzFeed	131
The Huffington Post	56
Nytimes	35
Upworthy	24
IFLScience	20
Washington Post	15
Mashable	13
Mic	11
Western Journalism	8
Business Insider	8
The Atlantic	6
CNN	6
the Guardian	6
Fox News	5
BuzzFeed Community	5
TODAY.com	4
Rolling Stone	4

We can clearly see that BuzzFeed dominates the list. In a distant second place, we can see The Huffington Post, which incidentally is another site that Jonah Peretti worked for. It appears that studying the science of virality can pay big dividends.

So far, we have examined images and headlines. Now, let's move on to examining the full text of the stories.

Exploring the story content

In the last section, we created a function to examine the common n-grams that are found in the headlines of our stories. Now, let's apply that to explore the full content of our stories.

We'll start by exploring bi-grams with the stop words removed. Since headlines are so short compared to the body of the stories, it makes sense to look at them with the stop words intact, although within the story, it typically makes sense to eliminate them:

```
hw, h1 = get_word_stats(dfc['text'], 2, 1)
```

```
hw
```

This generates the following output:

	count
(islamic, state)	160
(united, states)	126
(year, old)	121
(new, york)	91
(social, media)	60
(years, ago)	57
(white, people)	51
(first, time)	49
(bernie, sanders)	45
(don't, want)	44
(last, year)	43
(every, day)	43
(black, people)	40
(don't, know)	39
(climate, change)	39
(many, people)	38
(two, years)	37
(high, school)	36
(islamic, state's)	36
(president, obama)	36

Interestingly, we can see that the frivolity we saw in the headlines has completely disappeared. The text is now filled with content discussing terrorism, politics, and race relations.

How is it possible that the headlines are light-hearted, while the text is dark and controversial? I would suggest that this is because articles such as *13 Puppies Who Look Like Elvis* are going to have substantially less text than *The History of the Islamic State*.

Let's take a look at one more. We'll evaluate the tri-grams for the story bodies:

```
hw,hl = get_word_stats(dfc['text'], 3, 1)
```

```
hw
```

This code generates the following output:

	count
(advertisement, story, continues)	32
(check, articles, buzzfeed)	27
(articles, buzzfeed, com)	27
(buzzfeed, com, tagged)	21
(new, york, times)	19
(pic, twitter, com)	17
(via, upward, spiral)	17
(new, york, city)	16
(every, single, day)	16
(follow, us, twitter)	15
(like, us, facebook)	14
(g, m, o)	13
(facebook, follow, us)	13
(us, facebook, follow)	13
(5, xbox, two)	12
(playstation, 5, xbox)	12

We appear to have suddenly entered the land of advertising and social pandering. With that, let's move on to building a predictive model for content scoring.

Building a predictive content scoring model

Let's use what we have learned to create a model that can estimate the share counts for a given piece of content. We'll use the features we have already created, along with a number of additional ones.

Ideally, we would have a much larger sample of content—especially content that had more typical share counts—but we'll have to make do with what we have here.

We're going to be using an algorithm called **random forest regression**. In previous chapters, we looked at a more typical implementation of random forests that is based on classification, but here we're going to attempt to predict the share counts. We could consolidate our share classes into ranges, but it is preferable to use regression when dealing with continuous variables, which is what we're working with here.

To begin, we'll create a bare-bones model. We'll use the number of images, the site, and the word count. We'll train our model in terms of the number of Facebook likes. We're also going to be splitting our data into two sets: a training set and a test set.

First, we'll import the scikit-learn library, and then we'll prepare our data by removing the rows with nulls, resetting our index, and finally splitting the frame into our training and test set:

```
from sklearn.ensemble import RandomForestRegressor

all_data = dfc.dropna(subset=['img_count', 'word_count'])
all_data.reset_index(inplace=True, drop=True)

train_index = []
test_index = []
for i in all_data.index:
    result = np.random.choice(2, p=[.65, .35])
    if result == 1:
        test_index.append(i)
    else:
        train_index.append(i)
```

We used a random number generator with a probability set for approximately two-thirds and one-third to determine which row items (based on their `index`) would be placed in each set. Setting the probabilities like this ensures that we get approximately twice the number of rows in our training set compared to the test set. We can see this in the following code:

```
print('test length:', len(test_index), '\ntrain length:', len(train_index))
```

The preceding code generates the following output:

```
test length: 152  
train length: 233
```

Now, we'll continue with preparing our data. Next, we need to set up categorical encoding for our sites. Currently, our DataFrame has the name for each site represented with a string. We need to use dummy encoding. This creates a column for each site, and if the row has that particular site, then that column will be filled with a 1, while all the other columns for sites will be coded with a 0. Let's do that now:

```
sites = pd.get_dummies(all_data['site'])  
  
sites
```

The preceding code generates the following output:

	ABC News	Asbury Park Press	BBC News	Bloomberg.com	Boredom Therapy	Breitbart	Business Insider	BuzzFeed	BuzzFeed Community	CNN
0	0	0	0	0	0	0	0	1	0	0
1	0	0	0	0	0	0	0	1	0	0
2	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	1	0	0
5	0	0	0	0	0	0	0	0	0	0

You can see from the preceding output how the dummy encoding appears.

We'll now continue:

```
y_train = all_data.iloc[train_index]['fb'].astype(int)  
x_train_nosite = all_data.iloc[train_index][['img_count', 'word_count']]  
  
x_train = pd.merge(x_train_nosite, sites.iloc[train_index],  
left_index=True, right_index=True)  
  
y_test = all_data.iloc[test_index]['fb'].astype(int)  
x_test_nosite = all_data.iloc[test_index][['img_count', 'word_count']]
```

```
x_test = pd.merge(X_test_nosite, sites.iloc[test_index], left_index=True,  
right_index=True)
```

With that, we've set up our `X_test`, `X_train`, `y_test`, and `y_train` variables. Now, we're going to use our training data to build our model:

```
clf = RandomForestRegressor(n_estimators=1000)  
clf.fit(X_train, y_train)
```

With those two lines of code, we have trained our model. Let's use it to predict the Facebook likes for our test set:

```
y_actual = y_test  
deltas = pd.DataFrame(list(zip(y_pred, y_actual, (y_pred -  
y_actual)/(y_actual))), columns=['predicted', 'actual', 'delta'])  
  
deltas
```

This code generates the following output:

	predicted	actual	delta
0	292549.50	402000	-0.272265
1	530319.00	395000	0.342580
2	275994.00	383000	-0.279389
3	211309.00	383000	-0.448279
4	259235.00	373000	-0.305000
5	271205.00	378000	-0.282526
6	246440.00	378000	-0.348042
7	241459.00	371000	-0.349167
8	228334.00	352000	-0.351324
9	231523.00	362000	-0.360434
10	320281.00	358000	-0.105360

Here, we can see the predicted values, the actual value, and the difference as a percentage side by side. Let's take a look at the descriptive stats for this:

```
deltas['delta'].describe()
```

The preceding code generates the following output:

count	152.000000
mean	0.074475
std	0.525629
min	-0.841433
25%	-0.266933
50%	0.029722
75%	0.301447
max	2.688333
Name:	delta, dtype: float64

This looks amazing. Our median error is 0! Well, unfortunately, this is a particularly useful bit of information as errors are on both sides—positive and negative—and tend to average out, which is what we can see here. Let's look at a more informative metric to evaluate our model. We're going to look at root mean square error as a percentage of the actual mean.

Evaluating the model

To illustrate why this is more useful, let's run the following scenario on two sample series:

```
a = pd.Series([10, 10, 10, 10])
b = pd.Series([12, 8, 8, 12])

np.sqrt(np.mean((b-a)**2)) / np.mean(a)
```

This generates the following output:

0.2

Now, compare that to the mean:

```
(b-a).mean()
```

This generates the following output:

0.0

Clearly, the latter is the more meaningful statistic. Now, let's run it for our model:

```
np.sqrt(np.mean((y_pred-y_actual)**2))/np.mean(y_actual)
```

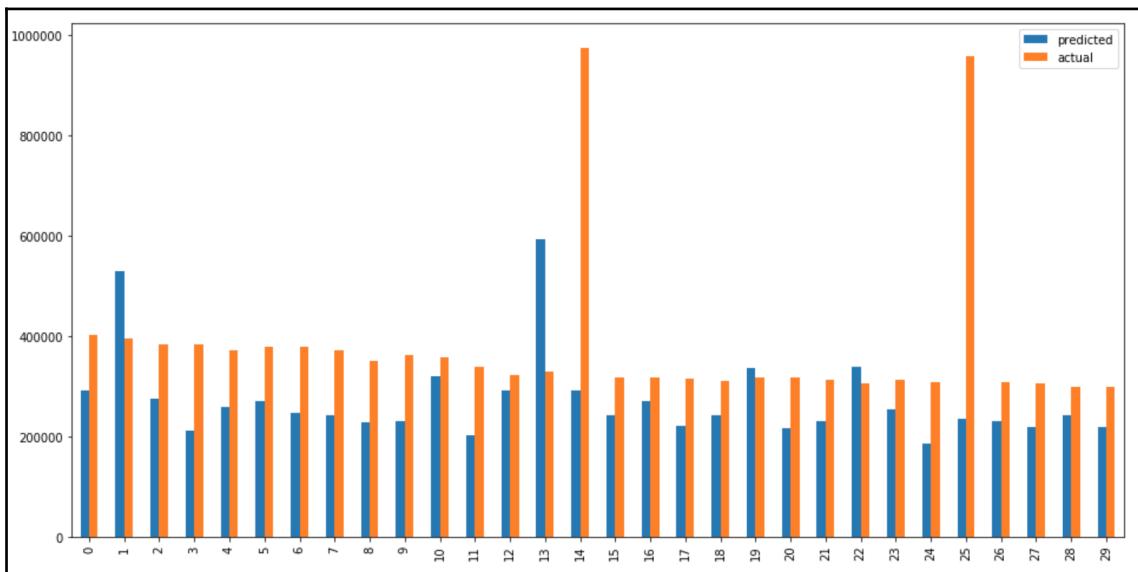
This generates the following output:

0.7381440390029351

Suddenly, our awesome model looks a lot less awesome. Let's take a look at some of the predictions our model made versus the actual values that can be seen in the data:

```
deltas[['predicted', 'actual']].iloc[:30, :].plot(kind='bar', figsize=(16, 8))
```

The preceding code generates the following output:



Based on what we can see here, the model—at least for this sample—tends to modestly underpredict the virality of the typical article, but then heavily underpredicts the virality for a small number. Let's see what those are:

```
all_data.loc[test_index[:30], ['title', 'fb']].reset_index(drop=True)
```

The preceding code results in the following output:

		title	fb
0		Churro Ice Cream Sandwiches Are The Most Delicious Thing You Need In Your Life	402000.0
1		One minute of fed-up celebrities talking about guns is actually worth your time.	395000.0
2		Donald Trump Just Stopped Being Funny	383000.0
3	This Billionaire Governor Taxed the Rich and Increased the Minimum Wage -- Now, His State's Economy Is One of the Best in the Country		383000.0
4		The girl who gets gifts from birds - BBC News	373000.0
5		This Jarring Photo Series Captures What PTSD Really Looks Like	378000.0
6		Being Too Nice Can Contribute to Depression	378000.0
7	For Julie Andrews and Christopher Plummer, The Sound of Music Was Never So Long, Farewell		371000.0
8		Drug Goes From 13.50 a <i>Tabletto</i> 750, Overnight	352000.0
9		Jeb Bush: Unwed Mothers Should Be Publicly Shamed	362000.0
10	He shows how the news talks about black people by talking about white people instead.		358000.0
11		The Norwegian Secret To Enjoying A Long Winter	340000.0
12		The World's Last Male Northern White Rhino Placed Under Armed Guard	323000.0
13	Listen to this organ in Croatia that uses the sea to make hauntingly beautiful music.		330000.0
14	16-Year-Old Malala Yousafzai Leaves Jon Stewart Speechless With Comment About Pacifism		975000.0
15		Dutch nursing home offers rent-free housing to students	319000.0
16		A Microbiologist Recreated 'Starry Night' With Bacteria In A Petri Dish	318000.0
17	Russian Man Will Become Subject Of First Human Head Transplant Ever Performed		316000.0
18		Kindergarten: While American Kids Read, Their Finnish Peers Play	311000.0
19	'We can't stop...': Watch couple's hilarious pregnancy announcement		318000.0
20		To My Daughter With Down Syndrome on Her Wedding Day	317000.0
21	I'm a veteran, and I hate 'Happy Memorial Day.' Here's why.		314000.0
22		The end of capitalism has begun	305000.0
23		War veterans proudly reveal their injuries in powerful naked pictures	313000.0
24		How to Raise a Creative Child. Step One: Back Off	308000.0
25	Husband Says He Can't Afford His Wife As A Stay-At-Home Mom And His Reasoning Adds Up. Big Time.		958000.0
26		Man's post about traffic stop goes viral	309000.0
27	Science Proves Bassists Are Actually the Most Important Members of the Band - Here's Why		306000.0
28		The Perseid Meteor Shower Peaks TONIGHT - Don't Miss It!	299000.0
29	OU professor: Youths' attraction to Sanders shows education failure		298000.0

From the preceding output, we can see that an article on *Malala* and an article on a *husband complaining about how much his stay-at-home wife costs him* greatly overshot the predicted numbers of our model. Both would seem to have high emotional valence.

Adding new features to our model

Now, let's add another feature to our model. Let's see whether adding the counts for words will help our model. We'll use a `CountVectorizer` to do this. Much like what we did with the site names, we'll be transforming individual words and n-grams into features:

```
from sklearn.feature_extraction.text import CountVectorizer

vect = CountVectorizer(ngram_range=(1, 3))
X_titles_all = vect.fit_transform(all_data['title'])

X_titles_train = X_titles_all[train_index]
X_titles_test = X_titles_all[test_index]

X_test = pd.merge(X_test, pd.DataFrame(X_titles_test.toarray(),
index=X_test.index), left_index=True, right_index=True)

X_train = pd.merge(X_train, pd.DataFrame(X_titles_train.toarray(),
index=X_train.index), left_index=True, right_index=True)
```

In the preceding lines, we have joined our existing features to our new n-gram features. Let's train our model and see whether we have any improvement:

```
clf.fit(X_train, y_train)

y_pred = clf.predict(X_test)

deltas = pd.DataFrame(list(zip(y_pred, y_actual, (y_pred -
y_actual)/(y_actual))), columns=['predicted', 'actual', 'delta'])

deltas
```

This code generates the following output:

	predicted	actual	delta
0	317559.0	402000	-0.210052
1	335695.0	395000	-0.150139
2	231530.0	383000	-0.395483
3	264364.0	383000	-0.309755
4	258683.0	373000	-0.306480
5	227973.0	378000	-0.396897
6	281266.0	378000	-0.255910
7	267251.8	371000	-0.279645
8	242438.0	352000	-0.311256
9	205556.0	362000	-0.432166
10	341792.0	358000	-0.045274

And if we check our error again, we will see the following:

```
np.sqrt(np.mean((y_pred-y_actual)**2))/np.mean(y_actual)
```

The preceding code generates the following output:

```
0.7142035714081069
```

So it appears that we have a modestly improved model. Let's add one more feature to our model—the word count of the title:

```
all_data = all_data.assign(title_wc = all_data['title'].map(lambda x: len(x.split(' '))))  
  
X_train = pd.merge(X_train, all_data[['title_wc']], left_index=True, right_index=True)  
  
X_test = pd.merge(X_test, all_data[['title_wc']], left_index=True, right_index=True)  
  
clf.fit(X_train, y_train)  
  
y_pred = clf.predict(X_test)  
  
np.sqrt(np.mean((y_pred-y_actual)**2))/np.mean(y_actual)
```

This code generates the following output:

0.710545088873634

It appears that each feature has modestly improved our model. There are certainly more features we could add to it. For example, we could add the day of the week and the hour of the posting, we could determine whether the article is a listicle by running a regex on the headline, or we could examine the sentiment of each article. But this only just touches on the features that could be important for modeling virality. We would certainly need to go much further to continue reducing the number of errors in our model.

I should also note that we have done only the most cursory testing of our model. Each measurement should be run multiple times to get a more accurate representation of the actual error rate. It is possible that there is no statistically discernible difference between our last two models since we only performed one test.

Summary

In this chapter, we examined what the common features of viral content are and how we can build a model to predict virality using a random forest regression. We also learned how to combine multiple types of features and how to split our model into training and test sets.

Hopefully, you will take what you've learned here to build the next viral empire. If that doesn't work out, perhaps the next chapter on mastering the stock market will.

7

Use Machine Learning to Forecast the Stock Market

Just recently, I was reading an article that described the tremendous success of a particular treatment in combating the **Methicillin-resistant Staphylococcus aureus (MRSA)** superbug. If you haven't heard of MRSA directly, it is likely you've heard something about current concerns that we are headed toward a time when our antibiotics will no longer be effective. This is largely an inevitable phenomenon that occurs because some bugs in the population are genetically more resistant to the relevant drug. When bugs that are susceptible to the drug are wiped out during treatment, the remaining drug-resistant bugs then reproduce and become the dominant variant in the population. To combat this, scientists are constantly pushing the boundaries of science to find new ways to combat these bugs.

In biology, this situation is called a **Red Queen's race**: the term comes from a quote in Lewis Carol's *Through the Looking Glass*:

"Now, here, you see, it takes all the running you can do, to keep in the same place."

This effectively describes the situation we're in with antibiotics, but perhaps the answer is not to be found in moving on to new, ever-more advanced drugs. Perhaps the answer might be found in understanding the larger cycle at play and using it to our advantage.

That new treatment for MRSA I was discussing earlier? That was actually from a 10th century book of medical potions called **Bald's Leechbook**. Among the listed ingredients were garlic, wine, and onions. This combination was found to have surpassed the results for our current treatment-of-last-resort, **vancomycin**.

But what does any of this have to do with forecasting the stock market? I would like to suggest that the very same phenomenon is at play in both scenarios. For example, every so often, a paper is published that alerts the financial world to the existence of a phenomenon that is a profitable anomaly. Most likely, this phenomenon is the downstream effect of some externally imposed, real-world constraint.

Take, for example, year-end tax loss sales. Because of the nature of tax laws, it makes sense for traders to sell their losses at the end of the year. This imposes downward price pressure on losing stocks toward year end. The falling prices then mean the stocks can be discounted beyond their fair value. This also means that, in January, the downward pressure is gone, replaced by upward pressure as new money is put to work in these undervalued assets. But once that phenomenon has been broadcast, it only makes sense for traders to attempt to get ahead of it and begin buying those stocks in late December and selling to those other traders who are expected to be buyers in January. These new traders, by entering the market, have now diluted the effect. They are relieving the year-end selling pressure and decreasing the January buying pressure. The effect is essentially arbitrated away right along with the profitability. What once worked no longer works and traders will begin to abandon the strategy and move on to the next new thing.

By now, I hope you are beginning to see the parallels. It is likely that the garlic, wine, and onions combination was once a very effective cure for bacterial infections that gradually lost its effectiveness as the bacteria adapted. Having been abandoned long ago as a cure, the bacteria had no reason to avoid the original genes that made them susceptible to this treatment. There are real-world constraints that make it nearly inevitable that these types of cycles will occur—both in living organisms and in markets. The key is to use this to our advantage.

In this chapter, we'll spend some time discussing how to build and test a trading strategy. We'll spend even more time, however, on how *not* to do it. There are countless pitfalls to avoid when trying to devise your own system, and it is nearly an impossible task, but it can be a lot of fun, and sometimes it can even be profitable. With that said, don't do dumb things such as risking money you can't afford to lose.



If you do decide to use anything you learned here to trade, you're on your own. This shouldn't be deemed investment advice of any kind, and I accept no responsibility for your actions.

In this chapter, we will cover the following topics:

- Types of market analysis
- What does research tell us about the stock market?
- How to develop a trading system

Types of market analysis

Let's begin with a discussion of some key terms and methods of analysis when dealing with financial markets. Though there are countless financial instruments, including stocks, bonds, ETFs, currencies, and swaps, we'll limit our discussion to stocks and the stock market. A stock is simply a fractional share of ownership in a public company. The price of a stock is expected to increase when future prospects for the company rise, and decrease as these prospects decline.

There are generally two camps that investors fall into. The first are the fundamental analysts. These analysts pore through company financials looking for information that indicates that, somehow, the market is undervaluing the shares of the company. These investors look at various factors, such as revenue, earnings, and cash flow, and various ratios of the values. This frequently involves looking at how one company's financials compare to another's.

The second camp of investors is the technical analysts. Technical analysts believe that share prices already reflect all publicly available information and that looking through the fundamentals is largely a waste of time. They believe that by looking at historical prices—stock charts—you can see areas where prices are likely to rise, fall, or stagnate. Generally, they feel that these charts reveal clues to investor psychology.

What both groups have in common is an underlying belief that the right analysis can lead to profits. But is that true?

What does research tell us about the stock market?

Perhaps the most influential theory of the stock market over the last 50 years is that of the efficient market hypothesis. This theory, developed by Eugene Fama, stipulates that markets are rational and that all the available information is appropriately reflected in stock prices. As such, it is impossible for an investor to consistently *beat the market* on a risk-adjusted basis. The efficient market hypothesis is often discussed as having three forms: a weak form, a semi-strong form, and a strong form:

1. In the weak form, the market is efficient in the sense that you cannot use past information from prices to predict future prices. Information is reflected in stocks relatively quickly, and while technical analysis would be ineffective, in some scenarios, fundamental analysis could be effective.

2. In the semi-strong form, prices immediately reflect all relevant new public information in an unbiased manner. Here, neither technical nor fundamental analysis would be effective.
3. And finally, in the strong form, stock prices reflect all public and private information.

Based on these theories, there isn't much hope of making money by exploiting patterns in the market. But fortunately, while the market operates in a largely efficient manner on the whole, distinct pockets of inefficiency have been uncovered. Most of these tend to be ephemeral, but some have been documented as persisting. One of the most noteworthy—even according to Fama—is the outperformance of momentum strategies.

So, what exactly is a momentum strategy?

There are a number of variations on the theme, but the basic idea is that stocks are ranked from the highest to lowest according to their return over a prior period. The top-ranked performers are bought and held for a period of time, and then the process is repeated after a fixed holding period. A typical long-only momentum strategy might involve buying the top 25 performing stocks in the S&P 500 over the past year, holding them for a year, selling them, and then repeating the process.

This may sound like an absurdly simple strategy, and it is, but it has consistently returned results that defy expectation. But why? As you can imagine, a lot of research has examined this effect, and the hypothesis is that there is something inherently systemically biased about how humans deal with new information. The research suggests they underreact to news in the short term and then overreact to news in the long term. This means that, when stocks begin to rise on exceptionally good news, investors don't fully elevate the share price to the level that would fully reflect this news; it takes time for them to come around to incorporating this rosy outlook.



This tendency of investors to fail to adequately reprice shares in the face of exceedingly good news may be the result of a well-documented bias called the **anchoring effect**. Essentially, when presented with a number, even a random number, and then asked to estimate a real-world value, such as the number of countries in Africa, for instance, our answer will be mentally tethered to that number we were primed with. Remarkably, this happens even if we know the number is randomly generated and unrelated to the question.

So, will momentum strategies be arbitraged away as more traders learn of it and pile in? There has been some evidence of this in recent years, but it remains unclear. Regardless, the effect was demonstrably real and persisted far longer than can currently be accounted for by the efficient market hypothesis. So, there at least appears to be some hope for market prediction. With that in mind, let's now move on to exploring how we might go about unearthing our own market anomalies.

How to develop a trading strategy

We'll begin our strategy development by focusing on the technical aspects. Let's take a look at the S&P 500 over the last few years. We'll use pandas to import our data. This will give us access to several sources of stock data, including Yahoo! And Google.

1. First, you'll need to install the data reader:

```
!pip install pandas_datareader
```

2. Then, go ahead and incorporate your imports:

```
import pandas as pd
from pandas_datareader import data, wb
import matplotlib.pyplot as plt

%matplotlib inline
pd.set_option('display.max_colwidth', 200)
```

3. Now, we'll get our data for the SPY ETF, which represents the stocks of the S&P 500. We'll pull data from the start of 2010 through December 2018:

```
import pandas_datareader as pdr

start_date = pd.to_datetime('2010-01-01')
stop_date = pd.to_datetime('2018-12-01')

spy = pdr.data.get_data_yahoo('SPY', start_date, stop_date)
```

This code generates the following output:

	High	Low	Open	Close	Volume	Adj Close
Date						
2010-01-04	113.389999	111.510002	112.370003	113.330002	118944600.0	94.545494
2010-01-05	113.680000	112.849998	113.260002	113.629997	111579900.0	94.795784
2010-01-06	113.989998	113.430000	113.519997	113.709999	116074400.0	94.862526
2010-01-07	114.330002	113.180000	113.500000	114.190002	131091100.0	95.262955
2010-01-08	114.620003	113.660004	113.889999	114.570000	126402800.0	95.579971
2010-01-11	115.129997	114.239998	115.080002	114.730003	106375700.0	95.713455
2010-01-12	114.209999	113.220001	113.970001	113.660004	163333500.0	94.820801
2010-01-13	114.940002	113.370003	113.949997	114.620003	161822000.0	95.621696
2010-01-14	115.139999	114.419998	114.489998	114.930000	115718800.0	95.880295

4. We can now plot our data. We'll select only the closing price:

```
spy_c = spy['Close']

fig, ax = plt.subplots(figsize=(15,10))
spy_c.plot(color='k')
plt.title("SPY", fontsize=20);
```

5. This generates the following output:



In the preceding diagram, we see the price chart of the daily closing price of the S&P 500 for the period we selected.

Analysis of the data

Let's run some analysis to see what the returns over this period might have been had we invested in this ETF:

1. We'll pull data for the `first_open` first:

```
first_open = spy['Open'].iloc[0]  
first_open
```

This generates the following output:

```
112.37000274658203
```

2. Next, let's get the closing price on the final day of the period:

```
last_close = spy['Close'].iloc[-1]  
last_close
```

This generates the following output:

275.6499938964844

3. And finally, let's see the change over the full period:

```
last_close - first_open
```

This generates the following output:

163.27999114990234

So, it appears that a purchase of 100 shares at the start of the period would have cost us approximately \$11,237 and, at the end of the period, those same 100 shares would have been valued at roughly \$27,564. This transaction would have given us a gain of just a bit over 145% over the period. Not too bad at all.

Let's now take a look at the return over the same period for just the intraday gains. This assumes we buy the stock at the opening of each day and sell it at the close of that same day:

```
spy['Daily Change'] = pd.Series(spy['Close'] - spy['Open'])
```

That will give us the change from the open to the close each day. Let's take a look at that:

```
spy['Daily Change']
```

This generates the following output:

Date	Daily Change
2010-01-04	0.959999
2010-01-05	0.369995
2010-01-06	0.190002
2010-01-07	0.690002
2010-01-08	0.680000
2010-01-11	-0.349998
2010-01-12	-0.309998
2010-01-13	0.670006
2010-01-14	0.440002
2010-01-15	-1.090004
2010-01-19	1.439995
2010-01-20	-0.389999
2010-01-21	-2.220001
2010-01-22	-1.989998
2010-01-25	-0.440002
2010-01-26	-0.029999
2010-01-27	0.660004
2010-01-28	-1.620003

Let's now total those changes over the period:

```
spy['Daily Change'].sum()
```

This generates the following output:

```
53.36994934082031
```

So, as you can see, we have gone from a gain of over 163 points to one of just over 53 points. Ouch! More than half the market's gains came from holding overnight during the period.

Volatility of the returns

The overnight returns were better than the intraday returns, but how about the volatility? Returns are always judged on a risk-adjusted basis, so let's see how the overnight trades compared to the intraday trades on the basis of their standard deviation.

We can use NumPy to calculate this for us as follows:

```
np.std(spy['Daily Change'])
```

This generates the following output:

1.3077717572417953

```
spy['Overnight Change'] = pd.Series(spy['Open'] - spy['Close'].shift(1))

np.std(spy['Overnight Change'])
```

This generates the following output:

0.9355729837546058

So our overnight trading not only had higher gains, but lower volatility as well, compared to the intraday trading. But not all volatility is created equal. Let's compare the mean change on downside days versus upside days for both strategies:

```
spy[spy['Daily Change'] < 0]['Daily Change'].mean()
```

This code generates the following output:

-0.9635857357921829

Run this code for upside days:

```
spy[spy['Overnight Change'] < 0]['Overnight Change'].mean()
```

We get the output as follows:

-0.6729057274054532

Again, we see that the average downside volatility is far less for our overnight trading strategy than our intraday trading strategy.

Daily returns

So far, we have looked at everything in terms of points, but let's now look at daily returns. This will help put our gains and losses into a more realistic context. Let's create a pandas series for each scenario: daily returns (close to close change), intraday returns, and overnight returns:

```
daily_rtn = ((spy['Close'] -  
    spy['Close'].shift(1))/spy['Close'].shift(1))*100  
  
id_rtn = ((spy['Close'] - spy['Open'])/spy['Open'])*100  
on_rtn = ((spy['Open'] - spy['Close'].shift(1))/spy['Close'].shift(1))*100
```

What we've done is use the pandas `.shift()` method to subtract each series from the prior day's series. For example, for the preceding first series, we are subtracting the close from the close one day before for each day. This will result in one less data point. If you print out the new series, you can see this as follows:

```
Daily_rtn
```

This generates the following output:

Date	Daily_rtn
2010-01-04	NaN
2010-01-05	0.264710
2010-01-06	0.070406
2010-01-07	0.422129
2010-01-08	0.332776
2010-01-11	0.139656
2010-01-12	-0.932624
2010-01-13	0.844623
2010-01-14	0.270457
2010-01-15	-1.122423
2010-01-19	1.249558
2010-01-20	-1.016859
2010-01-21	-1.922910
2010-01-22	-2.229183
2010-01-25	0.512771
2010-01-26	-0.419057
2010-01-27	0.475715
2010-01-28	-1.147229

Statistics for the strategies

Let's now take a look at the statistics for all three strategies. We'll create a function that can take in each series of returns, and will print out the summary results. We're going to get statistics for each of our winning, losing, and break-even trades, and something called the **Sharpe ratio**. I said earlier that returns are judged on a risk-adjusted basis; this is exactly what the Sharpe ratio provides us with; it is a method of comparing returns by accounting for the volatility of those returns. Here, we use the Sharpe ratio with an adjustment to annualize the ratio:

```
def get_stats(s, n=252):
    s = s.dropna()
    wins = len(s[s>0])
    losses = len(s[s<0])
    evens = len(s[s==0])
    mean_w = round(s[s>0].mean(), 3)
    mean_l = round(s[s<0].mean(), 3)
    win_r = round(wins/losses, 3)
    mean_trd = round(s.mean(), 3)
    sd = round(np.std(s), 3)
    max_l = round(s.min(), 3)
    max_w = round(s.max(), 3)
    sharpe_r = round((s.mean()/np.std(s))*np.sqrt(n), 4)
    cnt = len(s)
    print('Trades:', cnt,
          '\nWins:', wins,
          '\nLosses:', losses,
          '\nBreakeven:', evens,
          '\nWin/Loss Ratio', win_r,
          '\nMean Win:', mean_w,
          '\nMean Loss:', mean_l,
          '\nMean', mean_trd,
          '\nStd Dev:', sd,
          '\nMax Loss:', max_l,
          '\nMax Win:', max_w,
          '\nSharpe Ratio:', sharpe_r)
```

Let's now run each strategy to see the stats. We'll start with the buy-and-hold strategy (daily returns) and then move onto the other two, as follows:

```
get_stats(daily rtn)
```

This generates the following output:

```
Trades: 2244
Wins: 1228
Losses: 1006
Breakeven: 10
Win/Loss Ratio 1.221
Mean Win: 0.628
Mean Loss: -0.668
Mean 0.044
Std Dev: 0.93
Max Loss: -6.512
Max Win: 4.65
Sharpe Ratio: 0.7503
```

Run the following code for intraday returns:

```
get_stats(id_rtn)
```

This generates the following output:

```
Trades: 2245
Wins: 1223
Losses: 1004
Breakeven: 18
Win/Loss Ratio 1.218
Mean Win: 0.477
Mean Loss: -0.541
Mean 0.018
Std Dev: 0.723
Max Loss: -4.196
Max Win: 3.683
Sharpe Ratio: 0.3965
```

Run the following code for overnight returns:

```
get_stats(on_rtn)
```

This generates the following output:

```
Trades: 2244
Wins: 1202
Losses: 1005
Breakeven: 37
Win/Loss Ratio 1.196
Mean Win: 0.376
Mean Loss: -0.392
Mean 0.026
Std Dev: 0.556
Max Loss: -3.406
Max Win: 4.09
Sharpe Ratio: 0.745
```

As you can see, the buy-and-hold strategy has the highest mean return, as well as the highest standard deviation, of the three. It also has the largest daily drawdown (loss). You will also notice that, even though the overnight-only strategy has a higher mean return than the intraday strategy, it also has substantially less volatility. This, in turn, gives it a Sharpe ratio that is higher than the intraday strategy.

At this point, we have a solid baseline for comparing our future strategies. Now, I am going to tell you about a strategy that blows all three of these strategies out of the water.

The mystery strategy

Let's take a look at the statistics for this new mystery strategy:

```
Trades: 2244
Wins: 627
Losses: 454
Breakeven: 1163
Win/Loss Ratio 1.381
Mean Win: 0.679
Mean Loss: -0.639
Mean 0.061
Std Dev: 0.643
Max Loss: -3.99
Max Win: 5.88
Sharpe Ratio: 1.4947
```

With this strategy, I have essentially doubled the Sharpe ratio over buy-and-hold, lowered the volatility substantially, increased the max win, and reduced the max loss by a significant level.

And how is it that I devised this market-trouncing strategy? Wait for it... I did it by generating 5,000 random overnight signals and picked the best one.

This is obviously not the way to beat the market. So why then did I do it? To demonstrate that, if you test enough strategies, just by random chance you will come across a number that appears to be amazing. This is the called the **data mining fallacy**, and it is a real risk in trading strategy development. That is why it is so important to find a strategy that is anchored to real-world investor biases and behaviors. If you want an edge in trading, you don't trade the markets; you the trade people who trade markets.

An edge then comes from thoughtfully understanding how people might react incorrectly to certain situations.

Let's now extend our analysis. First, we'll pull data for the index beginning with the year 2000:

```
start_date = pd.to_datetime('2000-01-01')
stop_date = pd.to_datetime('2018-12-01')

sp = pdr.data.get_data_yahoo('SPY', start_date, stop_date)
```

Let's now take a look at our chart:

```
fig, ax = plt.subplots(figsize=(15,10))
sp['Close'].plot(color='k')
plt.title("SPY", fontsize=20)
```

This generates the following output:



Here, we see the price action for the SPY from the start of 2000 until December 1, 2018. There has certainly been a lot of movement during that period as the market has experienced both highly positive and highly negative regimes.

Let's get our baseline for our new expanded period for our three base strategies.

First, let's set up our variables for each:

```
long_day_rtn = ((sp['Close'] -  
sp['Close'].shift(1))/sp['Close'].shift(1))*100  
  
long_id_rtn = ((sp['Close'] - sp['Open'])/sp['Open'])*100  
  
long_on_rtn = ((sp['Open'] -  
sp['Close'].shift(1))/sp['Close'].shift(1))*100
```

Now, let's see what the point totals are for each:

```
(sp['Close'] - sp['Close'].shift(1)).sum()
```

This generates the following output:

```
130.21249389648438
```

Now, let's see what the point totals are for open to close:

```
(sp['Close'] - sp['Open']).sum()
```

This generates the following output:

```
-26.9024658203125
```

Now, let's see what the point totals are for close to open:

```
(sp['Open'] - sp['Close'].shift(1)).sum()
```

This generates the following output:

```
154.30245971679688
```

And now let's look at the statistics for each:

```
get_stats(long_day rtn)
```

This generates the following output:

```
Trades: 4759
Wins: 2552
Losses: 2188
Breakeven: 19
Win/Loss Ratio 1.166
Mean Win: 0.769
Mean Loss: -0.852
Mean 0.021
Std Dev: 1.211
Max Loss: -9.845
Max Win: 14.52
Sharpe Ratio: 0.2722
```

Now, let's look at the statistics for intraday returns:

```
get_stats(long_id rtn)
```

This generates the following output:

```
Trades: 4760
Wins: 2500
Losses: 2223
Breakeven: 37
Win/Loss Ratio 1.125
Mean Win: 0.642
Mean Loss: -0.719
Mean 0.001
Std Dev: 1.002
Max Loss: -8.991
Max Win: 8.435
Sharpe Ratio: 0.0191
```

Now, let's look at the statistics for overnight returns:

```
get_stats(long_on rtn)
```

This generates the following output:

```
Trades: 4759
Wins: 2533
Losses: 2163
Breakeven: 63
Win/Loss Ratio 1.171
Mean Win: 0.413
Mean Loss: -0.442
Mean 0.019
Std Dev: 0.656
Max Loss: -8.322
Max Win: 6.068
Sharpe Ratio: 0.4601
```

We can see that the differences between the three are even more pronounced over the longer period. Had you only held during the day over the past 18 years, you would have lost money in this S&P ETF. Had you held only overnight, you would have improved your total point returns by over 18%! Obviously, this presumes no trading costs and no taxes along with perfect fills but, regardless, this is a remarkable finding.

Building the regression model

Now that we have a baseline to compare with, let's build our first regression model. We're going to start with a very basic model using only the stock's prior closing values to predict the next day's close, and we're going to build it using a support vector regression. With that, let's set up our model:

1. The first step is to set up a DataFrame that contains a price history for each day. We're going to include the past 20 closes in our model:

```
for i in range(1, 21, 1):
    sp.loc[:, 'Close Minus ' + str(i)] = sp['Close'].shift(i)

sp20 = sp[[x for x in sp.columns if 'Close Minus' in x or x == 'Close']].iloc[20:, :]

sp20
```

2. This code gives us each day's closing price, along with the previous 20, all on the same line. The result of our code is seen in the following output:

Date	Close	Close Minus 1	Close Minus 2	Close Minus 3	Close Minus 4	Close Minus 5	Close Minus 6	Close Minus 7	Close Minus 8	Close Minus 9
2000-02-01	140.937500	139.562500	135.875000	140.250000	140.812500	141.937500	140.343704	144.437500	144.750000	147.000000
2000-02-02	141.062500	140.937500	139.562500	135.875000	140.250000	140.812500	141.937500	140.343704	144.437500	144.750000
2000-02-03	143.187500	141.062500	140.937500	139.562500	135.875000	140.250000	140.812500	141.937500	140.343704	144.437500
2000-02-04	142.593704	143.187500	141.062500	140.937500	139.562500	135.875000	140.250000	140.812500	141.937500	140.343704
2000-02-07	142.375000	142.593704	143.187500	141.062500	140.937500	139.562500	135.875000	140.250000	140.812500	141.937500

3. This will form the basis of the X array we will feed our model. But before we're ready for that, there are a few additional steps.
4. First, we'll reverse our columns so that time runs from left to right:

```
sp20 = sp20.iloc[:, ::-1]

sp20
```

This generates the following output:

Date	Close Minus 20	Close Minus 19	Close Minus 18	Close Minus 17	Close Minus 16	Close Minus 15	Close Minus 14	Close Minus 13	Close Minus 12	Close Minus 11
2000-02-01	145.437500	139.750000	140.000000	137.750000	145.750000	146.250000	144.500000	143.062500	145.000000	146.968704
2000-02-02	139.750000	140.000000	137.750000	145.750000	146.250000	144.500000	143.062500	145.000000	146.968704	145.812500
2000-02-03	140.000000	137.750000	145.750000	146.250000	144.500000	143.062500	145.000000	146.968704	145.812500	147.000000
2000-02-04	137.750000	145.750000	146.250000	144.500000	143.062500	145.000000	146.968704	145.812500	147.000000	144.750000
2000-02-07	145.750000	146.250000	144.500000	143.062500	145.000000	146.968704	145.812500	147.000000	144.750000	144.437500

5. Now, let's import our support vector machine and set our our training and test matrices and vectors:

```
from sklearn.svm import SVR
clf = SVR(kernel='linear')

X_train = sp20[:-2000]
y_train = sp20['Close'].shift(-1)[-2000:]

X_test = sp20[-2000:]
y_test = sp20['Close'].shift(-1)[-2000:]
```

6. We had just 5,000 data points to work with, so I chose to use the last 2,000 for testing. Let's now fit our model and use it to check out-of-sample data:

```
model = clf.fit(X_train, y_train)

preds = model.predict(X_test)
```

7. Now that we have our predictions, let's compare them to our actual data:

```
tf = pd.DataFrame(list(zip(y_test, preds)), columns=['Next Day  
Close', 'Predicted Next Close'], index=y_test.index)

tf
```

The preceding code generates the following output:

Date	Next Day Close	Predicted Next Close
2010-12-22	125.599998	125.836080
2010-12-23	125.650002	125.883611
2010-12-27	125.830002	125.706853
2010-12-28	125.919998	125.839451
2010-12-29	125.720001	125.849771
2010-12-30	125.750000	125.750927
2010-12-31	127.050003	125.800584
2011-01-03	126.980003	127.002578
2011-01-04	127.639999	126.998705

Performance of the model

Let's now look at the performance of our model. We're going to buy the next day's open if the close is predicted to be higher than the open. We'll then sell at the close that same day. We'll need to add a few extra data points to our DataFrame to calculate our results, as follows:

```
cdc = sp[['Close']].iloc[-1000:]
ndo = sp[['Open']].iloc[-1000:].shift(-1)

tf1 = pd.merge(tf, cdc, left_index=True, right_index=True)
tf2 = pd.merge(tf1, ndo, left_index=True, right_index=True)
tf2.columns = ['Next Day Close', 'Predicted Next Close', 'Current Day
Close', 'Next Day Open']

tf2
```

This generates the following output:

Date	Next Day Close	Predicted Next Close	Current Day Close	Next Day Open
2010-12-22	125.599998	125.836080	125.779999	125.639999
2010-12-23	125.650002	125.883611	125.599998	125.129997
2010-12-27	125.830002	125.706853	125.650002	125.900002
2010-12-28	125.919998	125.839451	125.830002	125.980003
2010-12-29	125.720001	125.849771	125.919998	125.800003
2010-12-30	125.750000	125.750927	125.720001	125.529999
2010-12-31	127.050003	125.800584	125.750000	126.709999
2011-01-03	126.980003	127.002578	127.050003	127.330002
2011-01-04	127.639999	126.998705	126.980003	126.580002

Here we'll add the following code to get our signal and our profit and loss for the signal:

```
def get_signal(r):
    if r['Predicted Next Close'] > r['Next Day Open']:
        return 1
    else:
        return 0

def get_ret(r):
    if r['Signal'] == 1:
        return ((r['Next Day Close'] - r['Next Day Open'])/r['Next Day Open']) * 100
    else:
        return 0

tf2 = tf2.assign(Signal = tf2.apply(get_signal, axis=1))
tf2 = tf2.assign(PnL = tf2.apply(get_ret, axis=1))

tf2
```

This generates the following output:

Date	Next Day Close	Predicted Next Close	Current Day Close	Next Day Open	Signal	PnL
2010-12-22	125.599998	125.836080	125.779999	125.639999	1	-0.031838
2010-12-23	125.650002	125.883611	125.599998	125.129997	1	0.415571
2010-12-27	125.830002	125.706853	125.650002	125.900002	0	0.000000
2010-12-28	125.919998	125.839451	125.830002	125.980003	0	0.000000
2010-12-29	125.720001	125.849771	125.919998	125.800003	1	-0.063594
2010-12-30	125.750000	125.750927	125.720001	125.529999	1	0.175258
2010-12-31	127.050003	125.800584	125.750000	126.709999	0	0.000000
2011-01-03	126.980003	127.002578	127.050003	127.330002	0	0.000000
2011-01-04	127.639999	126.998705	126.980003	126.580002	1	0.837413

Let's now see whether, using just the price history, we were able to successfully predict the next day's price. We'll start by calculating the points gained:

```
(tf2[tf2['Signal']==1]['Next Day Close'] - tf2[tf2['Signal']==1]['Next Day Open']).sum()
```

This generates the following output:

-7.410148620605469

Ouch! This looks bad. But what about the period we tested? We never evaluated it in isolation. How many points would our basic intraday strategy have generated during the last 2,000 days:

```
(sp['Close'].iloc[-2000:] - sp['Open'].iloc[-2000:]).sum()
```

This generates the following output:

45.95988464355469

So it looks as if our strategy is abysmal. Let's compare the two.

First, the basic intraday strategy for the period:

```
get_stats((sp['Close'].iloc[-2000:] -  
sp['Open'].iloc[-2000:])/sp['Open'].iloc[-2000:] * 100)
```

This generates the following output:

```
Trades: 2000  
Wins: 1084  
Losses: 900  
Breakeven: 16  
Win/Loss Ratio 1.204  
Mean Win: 0.461  
Mean Loss: -0.518  
Mean 0.017  
Std Dev: 0.702  
Max Loss: -4.196  
Max Win: 3.683  
Sharpe Ratio: 0.3789
```

And now the results for our model:

```
get_stats(tf2['PnL'])
```

This generates the following output:

```
Trades: 2000  
Wins: 485  
Losses: 407  
Breakeven: 1108  
Win/Loss Ratio 1.192  
Mean Win: 0.481  
Mean Loss: -0.579  
Mean -0.001  
Std Dev: 0.507  
Max Loss: -4.088  
Max Win: 3.683  
Sharpe Ratio: -0.0401
```

It's clear our strategy is not one we would want to implement. How might we improve what we have here? What if we modified our trading strategy? What if we only took trades that, instead of just being any amount greater than the open, were expected to be greater by a point or more. Would that help? Let's try it. We'll re-run our strategy with a modified signal, as demonstrated in the following code block:

```
def get_signal(r):
    if r['Predicted Next Close'] > r['Next Day Open'] + 1:
        return 1
    else:
        return 0

def get_ret(r):
    if r['Signal'] == 1:
        return ((r['Next Day Close'] - r['Next Day Open'])/r['Next Day Open']) * 100
    else:
        return 0

tf2 = tf2.assign(Signal = tf2.apply(get_signal, axis=1))
tf2 = tf2.assign(PnL = tf2.apply(get_ret, axis=1))

(tf2[tf2['Signal']==1]['Next Day Close'] - tf2[tf2['Signal']==1]['Next Day Open']).sum()
```

This generates the following output:

-8.640205383300781

And now the stats:

```
get_stats(tf2['PnL'])
```

This generates the following output:

```
Trades: 2000
Wins: 98
Losses: 101
Breakeven: 1801
Win/Loss Ratio 0.97
Mean Win: 0.746
Mean Loss: -0.807
Mean -0.004
Std Dev: 0.338
Max Loss: -3.977
Max Win: 3.683
Sharpe Ratio: -0.1987
```

We have gone from bad to worse. It appears that, if past price history suggests good things to come, you can expect precisely the opposite. We seem to have developed a contrarian indicator with our model. What if we explore that? Let's see what our gains would look like if we flipped our model so that, when we predict strong gains, we don't trade, but otherwise we do:

```
def get_signal(r):
    if r['Predicted Next Close'] > r['Next Day Open'] + 1:
        return 0
    else:
        return 1

def get_ret(r):
    if r['Signal'] == 1:
        return ((r['Next Day Close'] - r['Next Day Open'])/r['Next Day Open']) * 100
    else:
        return 0

tf2 = tf2.assign(Signal = tf2.apply(get_signal, axis=1))
tf2 = tf2.assign(PnL = tf2.apply(get_ret, axis=1))

(tf2[tf2['Signal']==1]['Next Day Close'] - tf2[tf2['Signal']==1]['Next Day Open']).sum()
```

This generates the following output:

```
45.95988464355469
```

Let's get our stats:

```
get_stats(tf2['PnL'])
```

This generates the following output:

```
Trades: 1999
Wins: 985
Losses: 799
Breakeven: 215
Win/Loss Ratio 1.233
Mean Win: 0.433
Mean Loss: -0.481
Mean 0.021
Std Dev: 0.615
Max Loss: -4.196
Max Win: 3.594
Sharpe Ratio: 0.5387
```

It looks like we do have a contrarian indicator here. When our model predicts strong next-day gains, the market significantly underperforms, at least during our test period. Would this hold in most scenarios? Unlikely. Markets tend to flip from regimes of mean reversion to regimes of trend persistence.

At this point, there are a number of extensions we could make to this model. We haven't even touched on using technical indicators or fundamental data in our model, and we have limited our trades to one day. All of this could be tweaked and extended upon, but there is one important point we have not addressed that must be mentioned.

The data we are working with is of a special type called **time series data**. Time series data requires special treatment to properly model it, as it typically violates the assumptions required for statistical modeling, such as a constant mean and variance.

One consequence of improperly handling time series data is that error metrics give wildly inaccurate measures. Because of significant autocorrelation, in other words, the data in the next period is highly correlated with data in the current period, it appears that we have achieved much better predictions than we actually have.

To address these issues, time series data is often **differenced** (in the case of stock data, this would mean we look at the daily change, not the absolute level of the index) to render it as what we call **stationary**; that is, it has a constant mean and variance and lacks significant autocorrelation.

If you intend to pursue working with time series data, I implore you to research these concepts in more detail.

Dynamic time warping

Next, however, I want to introduce another model, which uses a completely different algorithm. This algorithm is called **dynamic time warping**. What it does is give you a metric that represents the similarity between two time series:

1. To get started, we'll need to pip install the fastdtw library:

```
!pip install fastdtw
```

2. Once that is installed, we'll import the additional libraries we'll need:

```
from scipy.spatial.distance import euclidean
from fastdtw import fastdtw
```

3. Next, we'll create the function that will take in two series and return the distance between them:

```
def dtw_dist(x, y):
    distance, path = fastdtw(x, y, dist=euclidean)
    return distance
```

4. Now, we'll split our 18 years' worth of time series data into distinct five-day periods. We'll pair together each period with one additional point. This will serve to create our *x* and *y* data, as follows:

```
tseries = []
tlen = 5
for i in range(tlen, len(sp), tlen):
    ptc = sp['Close'].iloc[i-tlen:i].pct_change()[1:].values * 100
    res = sp['Close'].iloc[i-tlen:i+1].pct_change()[-1] * 100
    tseries.append((ptc, res))
```

5. We can take a look at our first series to get an idea of what the data looks like:

```
tseries[0]
```

This generates the following output:

```
(array([-3.91061453,  0.17889088, -1.60714286,  5.8076225 ]),  
 0.34305317324185847)
```

6. Now that we have each series, we can run them all through our algorithm to get the distance metric for each series against every other series:

```
dist_pairs = []  
for i in range(len(tseries)):  
    for j in range(len(tseries)):  
        dist = dtw_dist(tseries[i][0], tseries[j][0])  
        dist_pairs.append((i,j,dist,tseries[i][1], tseries[j][1]))
```

Once we have that, we can place it into a DataFrame. We'll drop series that have 0 distance, as they represent the same series. We'll also sort according to the date of the series and look only at those where the first series is before the second, chronologically speaking:

```
dist_frame = pd.DataFrame(dist_pairs, columns=['A', 'B', 'Dist', 'A Ret', 'B  
Ret'])  
  
sf =  
dist_frame[dist_frame['Dist']>0].sort_values(['A', 'B']).reset_index(drop=1)  
  
sfe = sf[sf['A']<sf['B']]
```

And finally, we'll limit our trades where the distance is less than 1 and the first series has a positive return:

```
winf = sfe[(sfe['Dist']<=1)&(sfe['A Ret']>0)]  
  
winf
```

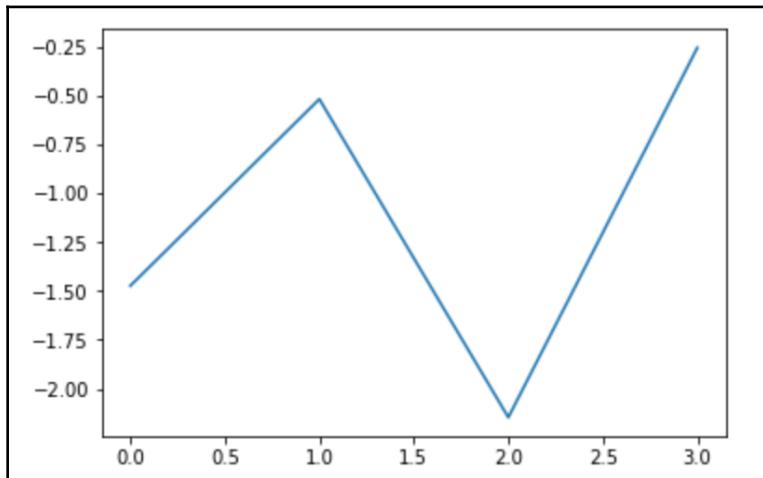
This generates the following output:

	A	B	Dist	A Ret	B Ret
3868	4	69	0.778629	1.360843	-1.696072
3995	4	196	0.608376	1.360843	0.410596
4165	4	366	0.973192	1.360843	0.040522
4346	4	547	0.832545	1.360843	-1.447712
4447	4	648	0.548912	1.360843	-0.510458
4591	4	792	0.740197	1.360843	0.819056
4613	4	814	0.411795	1.360843	0.578720
4686	4	887	0.606986	1.360843	0.233613
6297	6	598	0.678315	1.180863	2.896685
6323	6	624	0.897109	1.180863	0.757222

Let's see what one of our top patterns (A:6 and B:598) looks like when plotted:

```
plt.plot(np.arange(4), tseries[6][0]);
```

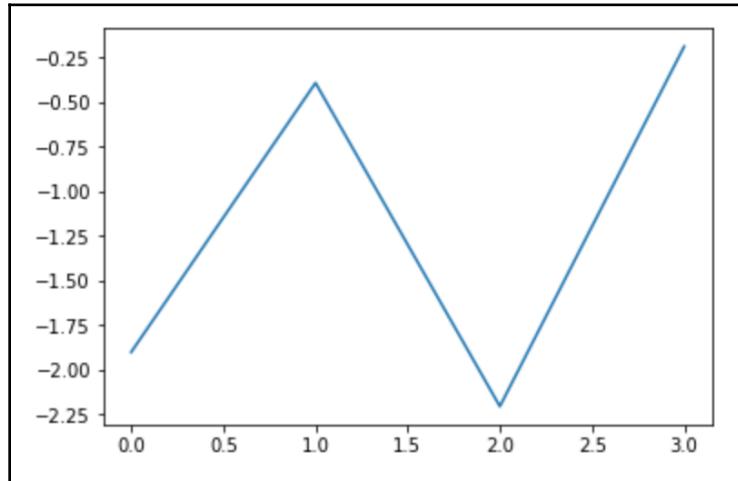
The preceding code generates the following output:



Now, we'll plot the second one:

```
plt.plot(np.arange(4), tseries[598][0])
```

The preceding code generates the following output:



As you can see, the curves are nearly identical, which is exactly what we want. We're going to try to find all curves that have positive next-day gains and then, once we have a curve that is highly similar to one of these profitable curves, we'll buy it in anticipation of another gain.

Evaluating our trades

Let's now construct a function to evaluate our trades. We'll buy similar curves unless they fail to return a positive result. If that happens, we'll eliminate them, as follows:

```
excluded = {}
return_list = []
def get_returns(r):
    if excluded.get(r['A']) is None:
        return_list.append(r['B Ret'])
        if r['B Ret'] < 0:
            excluded.update({r['A']:1})

winf.apply(get_returns, axis=1);
```

Now that we have all the returns from our trades stored in `return_list`, let's evaluate the results:

```
get_stats(pd.Series(return_list))
```

This generates the following output:

```
Trades: 722
Wins: 435
Losses: 284
Breakeven: 3
Win/Loss Ratio 1.532
Mean Win: 0.564
Mean Loss: -0.603
Mean 0.103
Std Dev: 0.792
Max Loss: -3.591
Max Win: 3.454
Sharpe Ratio: 2.0628
```

These results are by far the best we've seen. The win/loss ratio and the mean are far above our other models. It appears we may be on to something with this new model, especially compared to the others we've seen.

At this point, to vet our model further, we should explore its robustness by examining other time periods for our matches. Does extending beyond the four days improve the model? Should we always exclude the patterns that generate a loss? There are an enormous number of questions to explore at this point, but I'll leave this as an exercise for the reader.

Summary

In this chapter, we've looked at the inner workings of the stock market and explored a number of ways to approach utilizing machine learning in a trading strategy. There is no doubt the material in this chapter could fill a book itself. We even failed to cover some of the most important aspects of trading, such as portfolio construction, risk mitigation, and money management. These are critical components to any strategy, perhaps even more important than trade signals.

Hopefully, this will serve as a jumping-off point for your own explorations, but again, I caution you that *beating the market* is a nearly impossible game—one in which you are competing against the brightest minds in the world. If you do decide to try, I wish you the best of luck. Just remember that I warned you if it doesn't turn out like you hoped!

8

Classifying Images with Convolutional Neural Networks

In this chapter, we're going to explore the vast and awesome world of computer vision.

If you've ever wanted to construct a predictive machine learning model using image data, this chapter will serve as an easily-digestible and practical resource. We'll go step by step through building an image-classification model, cross-validating it, and then building it in a better way. At the end of this chapter, we'll have a *darn good* model and discuss some paths for future enhancement.

Of course, some background in the fundamentals of predictive modeling will help this to go smoothly. As you'll soon see, the process of converting images into usable features for our model might feel new, but once our features are extracted, the model-building and cross-validation processes are exactly the same.

In this chapter, we're going to build a convolutional neural network to classify images of articles of clothing from the Zalando Research dataset—a dataset of 70,000 images, each depicting 1 of 10 possible articles of clothing such as T-shirt/top, a pair of pants, a sweater, a dress, a coat, a sandal, a shirt, a sneaker, a bag, or an ankle boot. But first, we'll explore some of the fundamentals together, starting with image-feature extraction and walking through how convolutional neural networks work.

So, let's get started. Seriously!.

Here's what we'll cover in this chapter:

- Image-feature extraction
- Convolutional neural networks:
 - Network topology
 - Convolutional layers and filters
 - Max pooling layers

- Flattening
- Fully-connected layers and output
- Building a convolutional neural network to classify images in the Zalando Research dataset, using Keras

Image-feature extraction

When dealing with unstructured data, be it text or images, we must first convert the data into a numerical representation that's usable by our machine learning model. The process of converting data that is non-numeric into a numerical representation is called **feature extraction**. For image data, our features are the pixel values of the image.

First, let's imagine a $1,150 \times 1,150$ pixel grayscale image. A $1,150 \times 1,150$ pixel image will return a $1,150 \times 1,150$ matrix of pixel intensities. For grayscale images, the pixel values can range from 0 to 255, with 0 being a completely black pixel, and 255 being a completely white pixel, and shades of gray in between.

To demonstrate what this looks like in code, let's extract the features from our grayscale cat burrito. The image is available on GitHub at <https://github.com/PacktPublishing/Python-Machine-Learning-Blueprints-Second-Edition/tree/master/Chapter08> as `grayscale_cat_burrito.jpg`.



I've made the image assets used throughout this chapter available to you at <https://github.com/mroman09/packt-image-assets>. You can find our cat burritos there!

Let's now take a look at a sample of this in the following code:

```
import matplotlib.image as mpimg
import matplotlib.pyplot as plt
import pandas as pd
%matplotlib inline

cat_burrito = mpimg.imread('images/grayscale_cat_burrito.jpg')
cat_burrito
```



If you're unable to read a .jpg by running the preceding code, just install PIL by running `pip install pillow`.

In the preceding code, we imported `pandas` and two submodules: `image` and `pyplot`, from `matplotlib`. We used the `imread` method from `matplotlib.image` to read-in the image.

Running the preceding code gives us the following output:

```
array([[68, 68, 67, ..., 11, 10, 10],
       [68, 68, 67, ..., 11, 11, 11],
       [68, 68, 67, ..., 11, 11, 11],
       ...,
       [45, 45, 45, ..., 68, 68, 68],
       [45, 45, 45, ..., 68, 68, 68],
       [45, 45, 45, ..., 68, 68, 68]], dtype=uint8)
```

The output is a two-dimensional `numpy ndarray` that contains the features of our model. As with most applied machine learning applications, there are several preprocessing steps you'll want to perform on these extracted features, some of which we'll explore together on the Zalando fashion dataset later in this chapter, but these are the raw extracted features of the image!

The shape of the extracted features for our grayscale image is `image_height` rows \times `image_width` columns. We can check the shape easily by running the following:

```
cat_burrito.shape
```

The preceding code returns this output:

```
(1150, 1150)
```

We can check the maximum and minimum pixel values in our `ndarray` easily, too:

```
print(cat_burrito.max())
print(cat_burrito.min())
```

This returns the following:

```
255
0
```

Finally, we can display our grayscale image from our ndarray by running this code:

```
plt.axis('off')
plt.imshow(cat_burrito, cmap='gray');
```

The preceding code returns our image, which is available at <https://github.com/PacktPublishing/Python-Machine-Learning-Blueprints-Second-Edition/tree/master/Chapter08> as `output_grayscale_cat_burrito.png`.

The feature-extraction process for color images is identical; however, with color images, the shape of our ndarray output will be three-dimensional—a **tensor**—representing the **red, green, and blue (RGB)** pixel values of our image. Here, we'll carry out the same process as before, this time on a color version of the cat burrito. The image is available on GitHub at <https://github.com/PacktPublishing/Python-Machine-Learning-Blueprints-Second-Edition/tree/master/Chapter08> as `color_cat_burrito.jpg`.

Let's extract the features from our color version of the cat burrito by using the following code:

```
color_cat_burrito = mpimg.imread('images/color_cat_burrito.jpg')
color_cat_burrito.shape
```

Running this code returns the following output:

```
(1150, 1150, 3)
```

Again, here we see that this image contains three channels. Our `color_cat_burrito` variable is a tensor that contains three matrices that tell us what the RGB values are for each pixel in our image.

We can display the color image from our ndarray by running the following:

```
plt.axis('off')
plt.imshow(color_cat_burrito);
```

This returns our color image. The image is available on GitHub at <https://github.com/PacktPublishing/Python-Machine-Learning-Blueprints-Second-Edition/tree/master/Chapter08> as `output_color_cat_burrito.png`.

This is the first step of our image-feature extraction. We've taken a single image at a time and converted those images into numeric values using just a few lines of code. In doing so, we saw that extracting features from grayscale images produces a two-dimensional ndarray and extracting features from color images produces a tensor of pixel-intensity values. However, there's a slight problem. Remember, this is just a single image, a single training sample, a single *row* of our data. In the instance of our grayscale image, if we were to flatten this matrix into a single row, we would have `image_height x image_width` columns, or in our case, 1,322,500 columns. We can confirm that in code by running the following snippet:

```
# flattening our grayscale cat_burrito and checking the length
len(cat_burrito.flatten())
```

This is an issue! As with other machine learning modeling tasks, high dimensionality leads to model-performance issues. At this magnitude of dimensionality, any model we build will likely overfit, and training times will be slow.

This dimensionality problem is endemic to computer-vision tasks of this sort. Even a dataset of a lower resolution, 400 x 400 pixel grayscale cat burritos, would leave us with 160,000 features per image.

There is, however, a known solution to this problem: convolutional neural networks. In the next section, we'll continue our feature-extraction process using convolutional neural networks to build lower-dimensional representations of these raw image pixels. We'll go over the mechanics of how they work and continue to build an idea of why they're so performant in image-classification tasks.

Convolutional neural networks

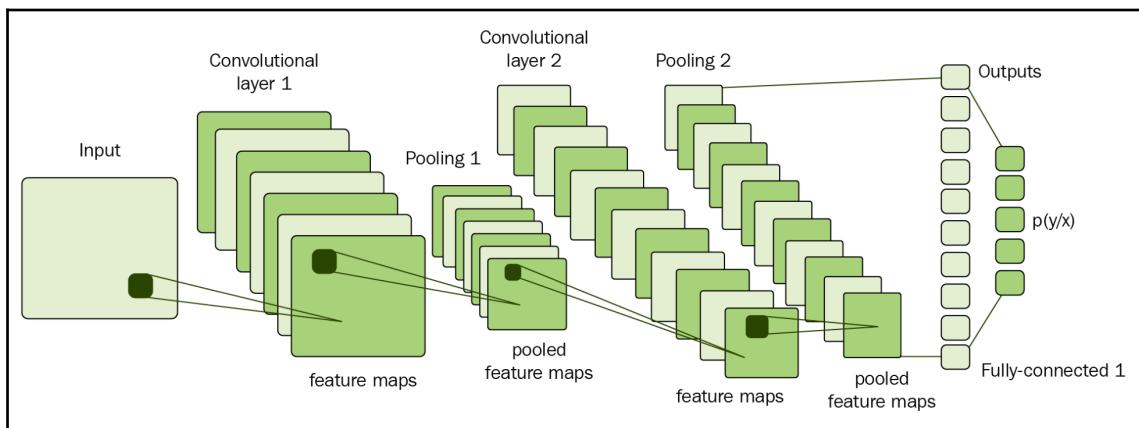
Convolutional neural networks are a class of neural network that resolve the high-dimensionality problem we alluded to in the previous section, and, as a result, excel at image-classification tasks. It turns out that image pixels in a given image region are highly correlated—they tell us similar information about that specific image region. Accordingly, using convolutional neural networks, we can scan regions of an image and summarize that region in lower-dimensional space. As we'll see, these lower-dimensional representations, called **feature maps**, tell us many interesting things about the presence of all sorts of shapes—from the simplest lines, shadows, loops, and swirls, to very abstract, complex forms specific to our data, in our case, cat ears, cat faces, or tortillas—and do this in fewer dimensions than the original image.

After using convolutional neural networks to extract these lower-dimensional features from our images, we'll pass the output of the convolutional neural network into a network suitable for the classification or regression task we want to perform. In our case, when modeling the Zalando research dataset, the output of our convolutional neural network will be passed into a fully-connected neural network for multi-class classification.

But how does this work? There are several key components we'll discuss with respect to convolutional neural networks on grayscale images, and these are all important for building our understanding.

Network topology

You may have encountered a diagram similar to the aforementioned that depicts a convolutional neural network to a feedforward neural network architecture. We'll be building something such as this very soon! But what's being depicted here? Check it out:



In the preceding diagram, on the very left, we have our input. These are the extracted features of our image, the matrix (as was the case with the grayscale cat burrito) of values ranging from 0 to 255 that describe the pixel intensities present in the image.

Next, we pass the data through alternating convolutional and max-pooling layers. These layers define the convolutional neural network component of the architecture depicted. We'll describe what each of these layer types do in the following two sections.

After this, we pass the data to a fully-connected layer before arriving at the output layer. These two layers describe a fully-connected neural network. You're free to use any multi-class classification algorithm you like here, instead of a fully-connected neural network—a **logistic regression** or **random forest classifier**, perhaps—but for our dataset, we'll be using a fully-connected neural network.

The output layer depicted is the same as for any other multi-class classifier. Sticking with our cat burrito example, let's suppose we were building a model to predict what kind of cat burrito an image was from five distinct classes: chicken cat burrito, steak cat burrito, cat burrito al pastor, vegetarian cat burrito, or fish cat burrito (I'll let you use your imagination to visualize what our training data might look like). The output layer would be the predicted probability that the image belonged to one of the five classes, with $\max(\text{probability})$ indicating what our model believes to be the most likely class.

At a high level, we've walked through the architecture, or **topology** of the preceding network. We've discussed our input versus the convolutional neural network component versus the fully-connected neural network component of the preceding topology. Let's dig just a bit deeper now and add some concepts that allow us to describe the topology in more detail:

- How many convolutional layers does the network have? Two.
- And in each convolutional layer, how many feature maps are there? There are seven in convolutional layer 1 and 12 in convolutional layer 2.
- How many pooling layers does the network have? Two.
- How many fully-connected layers are there? One.
- How many **neurons** are in the fully-connected layer? 10.
- What is the output? Five.

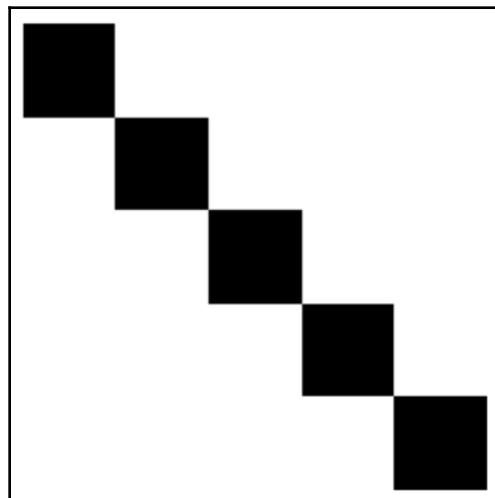
The modeler's decision to use two convolutional layers versus any other number or just a single fully-connected layer versus any other number should be thought of as the **hyperparameters** of the model. That is, it's something that we, as the modelers, should experiment with and cross-validate but not a parameter our model is explicitly learning and optimizing.

There are other useful things you can infer about the problem you're solving just by looking at the network's topology. As we discussed, the fact that our network's output layer contains five nodes lets us know that this neural network was designed to solve a multi-class classification task for which there are five classes. If it were a regression or a binary classification problem, our network's architecture would (in most cases) have a single output node. We also know that the modeler used seven filters in the first convolutional layer and 12 kernels in the second convolutional layer because of the number of feature maps resulting from each layer (we'll discuss what these kernels are in some more detail in the next section).

Great! We learned some useful jargon that will help us describe our networks and build our conceptual understanding of how they work. Now let's explore the convolutional layers of our architecture.

Convolutional layers and filters

Convolutional layers and filters are at the heart of convolutional neural networks. In these layers, we slide a filter (also referred to in this text as a **window** or **kernel**) over our ndarray feature and take the inner product at each step. Convolving our ndarray and kernel in this way results in a lower-dimensional image representation. Let's explore how this works on this grayscale image (available in image-assets repository):



The preceding image is a 5×5 pixel grayscale image shows a black diagonal line against a white background.

Extracting the features from the following diagram, we get the following matrix of pixel intensities:

0	255	255	255	255
255	0	255	255	255
255	255	0	255	255
255	255	255	0	255
255	255	255	255	0

Next, let's assume we (or Keras) instantiate the following kernel:

x 0	x 0	x 0
x 0	x 1	x 0
x 0	x 0	x 0

We'll now visualize the convolution process. The movement of the window starts from the top, left of our image matrix. We'll slide the window right by a predetermined stride size. In this case, our stride size will be 1, but in general the stride size should be considered another hyperparameter of your model. Once the window reaches the rightmost edge of the image, we'll slide our window down by 1 (our stride size), move the window back to the leftmost edge of the image, and start the process of taking the inner product again.

Now let's do this step by step:

1. Slide the kernel over the top-left part of the matrix and calculate the inner product:

0 x0	255 x0	255 x0	255	255
255 x0	0 x1	255 x0	255	255
255 x0	255 x0	0 x0	255	255
255	255	255	0	255
255	255	255	255	0

=

0		

I'll explicitly map out the inner product for this first step so that you can easily follow along:

$$(0 \times 0) + (255 \times 0) + (255 \times 0) + (255 \times 0) + (0 \times 1) + (255 \times 0) + (255 \times 0) + (255 \times 0) + (0 \times 0) = 0$$

We write the result to our feature map and continue!

2. Take the inner product and write the result to our feature map:

0	255 x0	255 x0	255 x0	255
255	0 x0	255 x1	255 x0	255
255	255 x0	0 x0	255 x0	255
255	255	255	0	255
255	255	255	255	0

=

0	255	

3. Step 3:

0	255	255 x0	255 x0	255 x0
255	0	255 x0	255 x1	255 x0
255	255	0 x0	255 x0	255 x0
255	255	255	0	255
255	255	255	255	0

=

0	255	255

4. We've reached the rightmost edge of the image. Slide the window down by 1, our stride size, and start the process again at the leftmost edge of the image:

0	255	255	255	255
255 x0	0 x0	255 x0	255	255
255 x0	255 x1	0 x0	255	255
255 x0	255 x0	255 x0	0	255
255	255	255	255	0

=

0	255	255
255		

5. Step 5:

0	255	255	255	255
255	0 x0	255 x0	255 x0	255
255	255 x0	0 x1	255 x0	255
255	255 x0	255 x0	0 x0	255
255	255	255	255	0

=

0	255	255
255	0	

6. Step 6:

0	255	255	255	255
255	0	255 x0	255 x0	255 x0
255	255	0 x0	255 x1	255 x0
255	255	255 x0	0 x0	255 x0
255	255	255	255	0

=

0	255	255
255	0	255

7. Step 7:

0	255	255	255	255
255	0	255	255	255
255 x0	255 x0	0 x0	255	255
255 x0	255 x1	255 x0	0	255
255 x0	255 x0	255 x0	255	0

=

0	255	255
255	0	255
255		

8. Step 8:

0	255	255	255	255
255	0	255	255	255
255	255 x0	0 x0	255 x0	255
255	255 x0	255 x1	0 x0	255
255	255 x0	255 x0	255 x0	0

=

0	255	255
255	0	255
255	255	

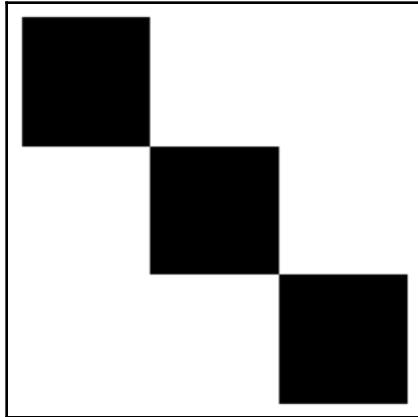
9. Step 9:

0	255	255	255	255
255	0	255	255	255
255	255	0 x0	255 x0	255 x0
255	255	255 x0	0 x1	255 x0
255	255	255 x0	255 x0	0 x0

=

0	255	255
255	0	255
255	255	0

Voila! We've now represented our original 5×5 image in a 3×3 matrix (our feature map). In this toy example, we've been able to reduce the dimensionality from 25 features down to just 9. Let's take a look at the image that results from this operation:



If you're thinking that this looks exactly like our original black diagonal line but smaller, you're right. The values the kernel takes determine what's being identified, and in this specific example, we used what's called an **identity kernel**. Kernels taking other values will return other properties of the image—detecting the presence of lines, edges, outlines, areas of high contrast, and more.

We'll apply multiple kernels to the image, simultaneously, at each convolutional layer. The number of kernels used is up to the modeler—another hyperparameter. Ideally, you want to use as few as possible while still achieving acceptable cross-validation results. The simpler the better! However, depending on the complexity of the task, we may see performance gains by using more. The same thinking can be applied when tuning the other hyperparameters of the model, such as the number of layers in the network or the number of neurons per layer. We're trading simplicity for complexity, and generalizability and speed for detail and precision.

While the number of kernels is our choice, the values that each kernel takes is a parameter of our model, which is learned from our training data and optimized during training in a manner that reduces the cost function.

We've seen the step-by-step process of how to convolve a filter with our image features to create a single feature map. But what happens when we apply multiple kernels simultaneously? And how do these feature maps pass through each layer of the network? Lets have a look at the following screenshot:



Image source: Lee et al., Convolutional Deep Belief Networks for Scalable Unsupervised Learning of Hierarchical Representations, via stack exchange. Source text here:
<https://ai.stanford.edu/~ang/papers/icml09-ConvolutionalDeepBeliefNetworks.pdf>

The preceding screenshot visualizes the feature maps generated at each convolutional layer of a network trained on images of faces. In the early layers of the network (at the very bottom), we detect the presence of simple visual structures—simple lines and edges. We did this with our identity kernel! The output of this first layer gets passed on to the next layer (the middle row), which combines these simple shapes into abstract forms. We see here that the combination of edges build the components of a face—eyes, noses, ears, mouths, and eyebrows. The output of this middle layer, in turn, gets passed to a final layer, which combines the combination of edges into complete objects—in this case, different people's faces.

One particularly powerful property of this entire process is that all of these features and representations are learned from the data. At no point do we explicitly tell our model: *Model, for this task, I'd like to use an identity kernel and a bottom sobel kernel in the first convolutional layer because I think these two kernels will extract the most signal-rich feature maps.* Once we've set the hyperparameter for the number of kernels we want to use, the model learns through optimization what lines, edges, shadows, and complex combinations thereof are best suited to determine what a face is or isn't. The model performs this optimization with no domain-specific, hardcoded rules about what faces, cat burritos, or clothes are.

There are many other fascinating properties of convolutional neural networks, which we won't cover in this chapter. However, we did explore the fundamentals, and hopefully you have a sense of the importance of using convolutional neural networks to extract highly expressive, signal-rich, low-dimensional features.

Next, we'll discuss *Max pooling layers*.

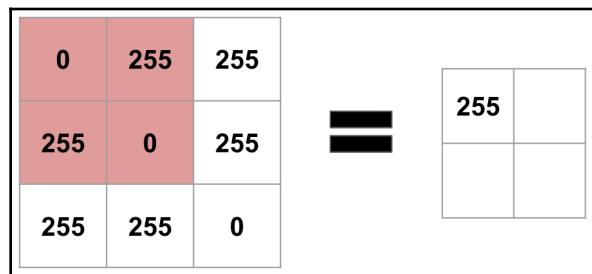
Max pooling layers

We've discussed the importance of reducing our dimensional space and how we use convolutional layers to achieve this. We use max pooling layers for the same reason—to further reduce dimensionality. Quite intuitively, as the name suggests, with max pooling, we slide a window over our feature map and take the max value for the window. Let's return to the feature map from our diagonal-line example to illustrate, this as follows:

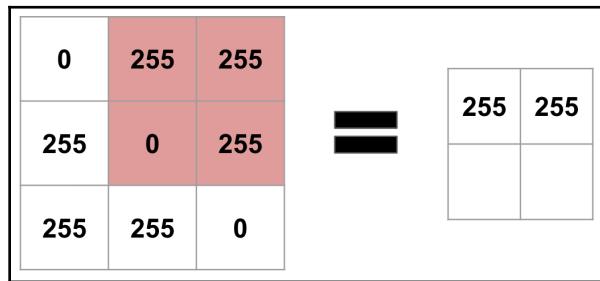
0	255	255
255	0	255
255	255	0

Let's see what happens when we max pool the preceding feature map using a 2×2 window. Again, all we're doing here is returning $\max(\text{values in window})$:

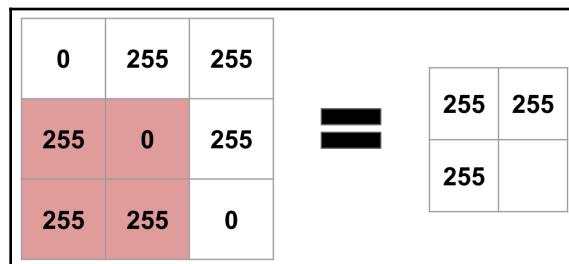
1. Return $\max(0, 255, 255, 0)$, which gets us 255:



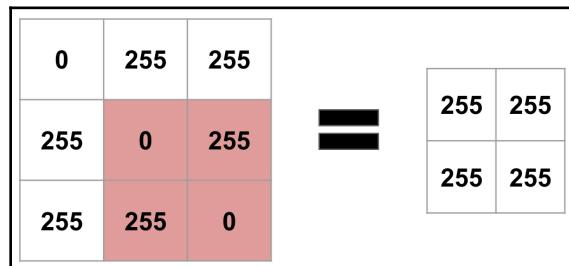
2. Step 2:



3. Step 3:



4. Step 4:



By max pooling our feature map with a 2×2 window, we've knocked a column and a row off, getting us from a 3×3 representation to a 2×2 —Not bad!

There are other forms of pooling as well—average pooling and min pooling, for example; however, you'll see max pooling used most often.

Next, we'll discuss flattening, a step we'll perform to turn our max-pooled feature map into the right shape for modeling.

Flattening

So far, we've focused on building as condensed and expressive a representation of our features as possible and used convolutional neural networks and max pooling layers to do this. The last step of our transformation is to flatten our convolved and max-pooled ndarray, in our example a 2×2 matrix, into a single row of training data.

Our max-pooled diagonal black line example would look something like the following, in code:

```
import numpy as np
max_pooled = np.array([[255, 255], [255, 255]])
max_pooled
```

Running this code returns the following output:

```
array([[255, 255],
       [255, 255]])
```

We can check the shape here by running the following:

```
max_pooled.shape
```

This returns this output:

```
(2, 2)
```

To turn this matrix into a single training sample, we just run `flatten()`. Let's do this and look at the shape of our flattened matrix:

```
flattened = max_pooled.flatten()
flattened.shape
```

This generates the following output:

(4,)

What started as a 5×5 matrix of pixel intensities is now a single row with four features. We can now pass this into a fully-connected neural network.

Fully-connected layers and output

The fully-connected layers are where we map our input—the rows resulting from us convolving, max-pooling, and flattening our original extracted features—to our target class or classes. Here, each input is connected to every **neuron** or **node** in the following layer. The strength of these connections, or **weights**, and a **bias** term present in each node of the network are parameters of the model, optimized throughout the training process to minimize an objective function.

The final layer of our model will be our output layer, which gives us our model predictions. The number of neurons in our output layer and the **activation function** we apply to it are determined by the kind of problem we're trying to solve: regression, binary classification, or multi-class classification. We'll see exactly how to set up the fully-connected and output layers for a multi-class classification task when we start working with the Zalando Research fashion dataset in the next section.



The fully-connected layers and output—that is, the feedforward neural network component of our architecture—belong to a distinct neural network type from the convolutional neural networks we discussed in this section. We briefly described how feedforward networks work in this section only to provide color on how the classifier component of our architecture works. You can always substitute this portion of the architecture for a classifier you are more familiar with, such as a **logit**!

With this fundamental knowledge, you're now ready to build your network!

Building a convolutional neural network to classify images in the Zalando Research dataset, using Keras

In this section, we'll be building our convolutional neural network to classify images of clothing, using Zalando Research's fashion dataset. The repository for this dataset is available at <https://github.com/zalandoresearch/fashion-mnist>.

This dataset contains 70,000 grayscale images—each depicting an article of clothing—from 10 possible clothing articles. Specifically, the target classes are as follows: T-shirt/top, pants, sweater, dress, coat, sandal, shirt, sneaker, bag, and ankle boot.

Zalando, a Germany-based e-commerce company, released this dataset to provide researchers with an alternative to the classic MNIST dataset of handwritten digits. Additionally, this dataset, which they call **Fashion MNIST**, is a bit more challenging to predict excellently—the MNIST handwritten-digits dataset can be predicted with 99.7% accuracy without the need for extensive preprocessing or particularly deep neural networks.

So, let's get started! Follow these steps:

1. Clone the repository to our desktop. From the terminal, run the following:

```
cd ~/Desktop/  
git clone git@github.com:zalandoresearch/fashion-mnist.git
```



If you haven't done so already, please install Keras by running `pip install keras` from the command line. We'll also need to install TensorFlow. To do this, run `pip install tensorflow` from the command line.

2. Import the libraries we'll be using:

```
import sys  
import numpy as np  
import pandas as pd  
from keras.models import Sequential  
from keras.layers import Dense, Dropout, Flatten  
from keras.layers import Conv2D, MaxPool2D  
from keras.utils import np_utils, plot_model  
from PIL import Image  
import matplotlib.pyplot as plt
```

Many of these libraries should look familiar by now. However, for some of you, this may be your first time using Keras. Keras is a popular Python deep learning library. It's a wrapper that can run on top of machine learning frameworks such as TensorFlow, CNTK, or Theano.

For our project, Keras will be running TensorFlow under the hood. Using TensorFlow directly would allow us more explicit control of the behavior of our networks; however, because TensorFlow uses dataflow graphs to represent its operations, this can take some getting used to. Luckily for us, Keras abstracts a lot of this away and its API is a breeze to learn for those comfortable with `sklearn`.

The only other library that may be new to some of you here will be the **Python Imaging Library (PIL)**. PIL provides certain image-manipulation functionalities. We'll use it to visualize our Keras network's topology.

3. Load in the data. Zalando has provided us with a helper script that does the loading in for us. We just have to make sure that `fashion-mnist/utils/` is in our path:

```
sys.path.append('/Users/Mike/Desktop/fashion-mnist/utils/')
import mnist_reader
```

4. Load in the data using the helper script:

```
x_train, y_train =
mnist_reader.load_mnist('/Users/Mike/Desktop/fashion-
mnist/data/fashion', kind='train')
x_test, y_test =
mnist_reader.load_mnist('/Users/Mike/Desktop/fashion-
mnist/data/fashion', kind='t10k')
```

5. Take a look at the shapes of `x_train`, `x_test`, `y_train`, and `y_test`:

```
print(x_train.shape, y_train.shape)
print(x_test.shape, y_test.shape)
```

Running that code gives us the following output:

```
(60000, 784) (60000,)
(10000, 784) (10000,)
```

Here, we can see our training set contains 60,000 images and our test contains 10,000 images. Each image is currently a vector of values 784 that are elements long. Let's now check the data types:

```
print(type(X_train))
print(type(y_train))
print(type(X_test))
print(type(y_test))
```

This returns the following:

```
<class 'numpy.ndarray'>
<class 'numpy.ndarray'>
<class 'numpy.ndarray'>
<class 'numpy.ndarray'>
```

Next, let's see what the data looks like. Remember, in its current form, each image is a vector of values. We know the images are grayscale, so to visualize each image, we'll have to reshape these vectors into a 28 x 28 matrix. Let's do this and peek at the first image:

```
image_1 = X_train[0].reshape(28, 28)
plt.axis('off')
plt.imshow(image_1, cmap='gray');
```

This generates the following output:



Awesome! We can check to see the class this image belongs to by running the following:

```
y_train[0]
```

This generates the following output:



The classes are encoded from 0-9. In the README, Zalando provides us with the mapping:

Label	Description
0	T-shirt/top
1	Trouser
2	Pullover
3	Dress
4	Coat
5	Sandal
6	Shirt
7	Sneaker
8	Bag
9	Ankle boot

Given this, we now know our first image is of an ankle boot. Sweet! Let's create an explicit mapping of these encoded values to their class names. This will come in handy momentarily:

```
mapping = {0: "T-shirt/top", 1:"Trouser", 2:"Pullover", 3:"Dress",
           4:"Coat", 5:"Sandal", 6:"Shirt", 7:"Sneaker", 8:"Bag", 9:"Ankle Boot"}
```

Great. We've seen a single image, but we still need to get a feel for what's in our data. What do the images look like? Getting a grasp of this will tell us certain things. As an example, I'm interested to see how visually distinct the classes are. Classes that look similar to other classes will be harder for a classifier to differentiate than classes that are more unique.

Here, we define a helper function to help us through our visualization journey:

```
def show_fashion_mnist(plot_rows, plot_columns, feature_array,
target_array, cmap='gray', random_seed=None):
    '''Generates a plot_rows * plot_columns grid of randomly selected
    images from a feature      array. Sets the title of each subplot equal
    to the associated index in the target array and      unencodes (i.e. title
    is in plain English, not numeric). Takes as optional args a color map
    and a random seed. Meant for EDA.'''
    # Grabs plot_rows*plot_columns indices at random from X_train.
    if random_seed is not None:
        np.random.seed(random_seed)
    feature_array_indices = np.random.randint(0, feature_array.shape[0],
size = plot_rows*plot_columns)
    # Creates our plots
    fig, ax = plt.subplots(plot_rows, plot_columns, figsize=(18,18))
    reshaped_images_list = []

    for feature_array_index in feature_array_indices:
        # Reshapes our images, appends tuple with reshaped image and class
        to a reshaped_images_list.
        reshaped_image =
    feature_array[feature_array_index].reshape((28,28))
        image_class = mapping[target_array[feature_array_index]]
        reshaped_images_list.append((reshaped_image, image_class))
    # Plots each image in reshaped_images_list to its own subplot
    counter = 0
    for row in range(plot_rows):
        for col in range(plot_columns):
            ax[row,col].axis('off')
            ax[row, col].imshow(reshaped_images_list[counter][0],
                               cmap=cmap)
            ax[row, col].set_title(reshaped_images_list[counter][1])
            counter +=1
```

What does this function do? It creates a grid of images selected at random from the data so that we can view multiple images simultaneously.

It takes as arguments the desired number of image rows (`plot_rows`), image columns (`plot_columns`), our `X_train` (`feature_array`), and `y_train` (`target_array`) and generates a matrix of images that's `plot_rows` x `plot_columns` large. As optional arguments, you can specify a `cmap`, or colormap (the default is 'gray' because these are grayscale images), and a `random_seed`, if replicating the visualization is important.

Let's see how to run this, as follows:

```
show_fashion_mnist(4, 4, x_train, y_train, random_seed=72)
```

This returns the following:



Visualization output

Remove the `random_seed` argument and rerun this function several times. Specifically, run the following code:

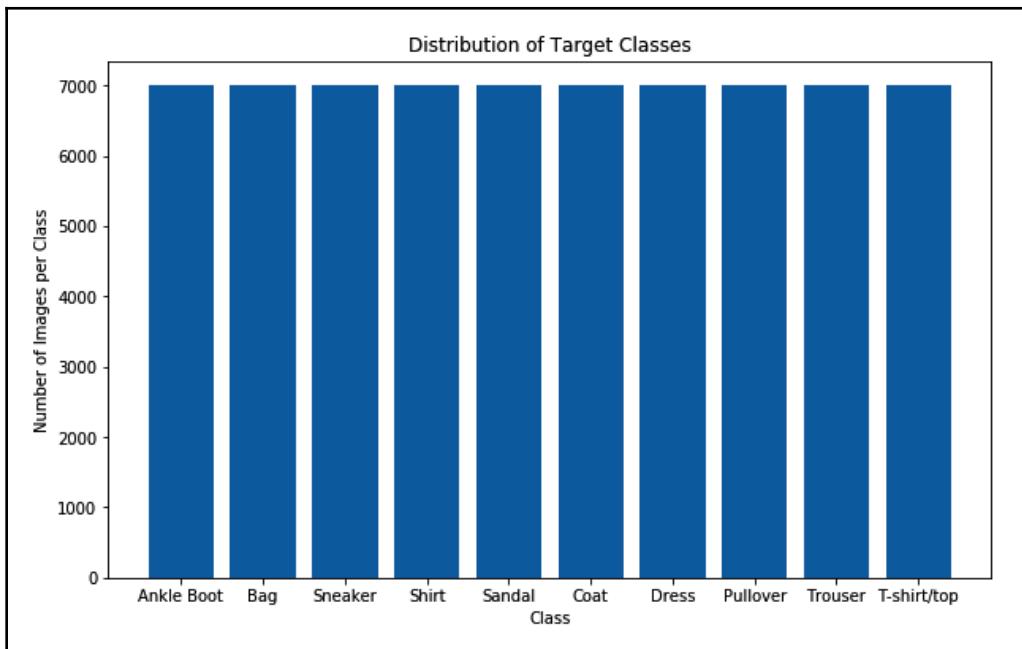
```
show_fashion_mnist(4, 4, x_train, y_train)
```

You may have noticed that at this resolution some classes look quite similar and others quite distinct. For example, samples of the **t-shirt/top** target class can look very similar to samples from the **shirt** and **coat** target classes, whereas the **sandal** target class seems to be quite different than the rest. This is food for thought when thinking about where our model may be weak versus where it's likely to be strong.

Now let's take a peek at the distribution of target classes in our dataset. Will we have to do any upsampling or downsampling? Let's check:

```
y = pd.Series(np.concatenate((y_train, y_test)))
plt.figure(figsize=(10, 6))
plt.bar(x=[mapping[x] for x in y.value_counts().index], height =
y.value_counts());
plt.xlabel("Class")
plt.ylabel("Number of Images per Class")
plt.title("Distribution of Target Classes");
```

Running the preceding code generates the following plot:



Awesome! No class-balancing to do here.

Next, let's start preprocessing our data to get it ready for modeling.

As we discussed in our *Image-feature extraction* section, these grayscale images contain pixel values ranging from 0 to 255. We confirm this by running the following code:

```
print(X_train.max())
print(X_train.min())
print(X_test.max())
print(X_test.min())
```

This returns the following values:

255
0
255
0

For the purposes of modeling, we're going to want to normalize these values on a 0–1 scale. This is a common preprocessing step when preparing image data for modeling. Keeping our values in this range will allow our neural network to converge more quickly. We can normalize the data by running the following:

```
# First we cast as float
X_train = X_train.astype('float32')
X_test = X_test.astype('float32')
# Then normalize
X_train /= 255
X_test /= 255
```

Our data is now scaled from 0.0 to 1.0. We can confirm this by running the following code:

```
print(X_train.max())
print(X_train.min())
print(X_test.max())
print(X_test.min())
```

This returns the following output:

1.0
0.0
1.0
0.0

The next preprocessing step we'll need to perform before running our first Keras network will be to reshape our data. Remember, the shapes of our `x_train` and `x_test` are currently (60,000, 784) and (10,000,784), respectively. Our images are still vectors. For us to convolve these lovely kernels all over the image, we'll need to reshape them into their 28 x 28 matrix form. Additionally, Keras requires that we explicitly declare the number of channels for our data. Accordingly, when we reshape these grayscale images for modeling, we'll declare 1:

```
x_train = x_train.reshape(x_train.shape[0], 28, 28, 1)
x_test = x_test.reshape(x_test.shape[0], 28, 28, 1)
```

Lastly, we'll one-hot encode our `y` vectors to conform with the target shape requirements of Keras:

```
y_train = np_utils.to_categorical(y_train, 10)
y_test = np_utils.to_categorical(y_test, 10)
```

We're now ready for modeling. Our first network will have eight hidden layers. The first six hidden layers will consist of alternating convolutional and max pooling layers. We'll then flatten the output of this network and feed that into a two-layer feedforward neural network before generating our predictions. Here's what this looks like, in code:

```
model = Sequential()
model.add(Conv2D(filters = 35, kernel_size=(3, 3), input_shape=(28,28,1),
activation='relu'))
model.add(MaxPool2D(pool_size=(2, 2)))
model.add(Conv2D(filters = 35, kernel_size=(3, 3), activation='relu'))
model.add(MaxPool2D(pool_size=(2, 2)))
model.add(Conv2D(filters = 45, kernel_size=(3, 3), activation='relu'))
model.add(MaxPool2D(pool_size=(2, 2)))
model.add(Flatten())
model.add(Dense(64, activation='relu'))
model.add(Dense(32, activation='relu'))
model.add(Dense(10, activation='softmax'))
model.compile(optimizer='adam', loss='categorical_crossentropy',
metrics=['accuracy'])
```

Let's describe what's happening on each line in some depth:

- **Line 1:** Here, we just instantiate our model object. We'll further define the architecture—that is, the number of layers—sequentially with a series of `.add()` method calls that follow. This is the beauty of the Keras API.

- **Line 2:** Here, we add our first convolutional layer. We specify 35 kernels, each 3×3 in size. After this, we specify the image input shape, $28 \times 28 \times 1$. We only have to specify the input shape in the first `.add()` call of our network. Lastly, we specify our activation function as `relu`. Activation functions transform the output of a layer before it's passed into the next layer. We'll apply activation functions to our `Conv2D` and `Dense` layers. These transformations have many important properties. Using `relu` here speeds up the convergence of our network, <http://www.cs.toronto.edu/~fritz/absps/imagenet.pdf>
- pdf and `relu`, relative to alternative activation functions, isn't expensive to compute—we're just transforming negative values to 0, and otherwise keeping all positive values. Mathematically, the `relu` function is given by `max(0, value)`. For the purpose of this chapter, we'll stick to the `relu` activation for every layer but the output layer.
- **Line 3:** Here, we add our first max pooling layer. We specify that the window size of this layer will be 2×2 .
- **Line 4:** This is our second convolutional layer. We set it up just as we set up the first convolutional layer.
- **Line 5:** This is the second max pooling layer. We set this layer up just as we set up the first max pooling layer.
- **Line 6:** This is our third and final convolutional layer. This time, we add additional filters (45 versus the 35 in previous layers). This is just a hyperparameter, and I encourage you to try multiple variations of this.
- **Line 7:** This is the third and final max pooling layer. It's configured the same as all max pooling layers that came before it.
- **Line 8:** Here's where we flatten the output of our convolutional neural network.
- **Line 9:** Here's the first layer of our fully-connected network. We specify 64 neurons in this layer and a `relu` activation function.
- **Line 10:** Here's the second layer of our fully-connected network. We specify 32 neurons for this layer and a `relu` activation function.
- **Line 11:** This is our output layer. We specify 10 neurons, equal to the number of target classes in our data. Since this is a multi-class classification problem, we specify a `softmax` activation function. The output will represent the predicted probability of the image belonging to classes 0–9. These probabilities will sum to 1. The highest predicted probability of the 10 will represent the class our model believes to be the most likely class.

- **Line 12:** Here's where we compile our Keras model. In the compile step, we specify our optimizer, Adam, a **gradient-descent** algorithm that automatically adapts its learning rate. We specify our **loss function**—in this case, categorical cross entropy because we're performing a multi-class classification problem. Lastly, for the metrics argument, we specify accuracy. By specifying this, Keras will inform us of our train and validation accuracy for each epoch that our model runs.

We can get a summary of our model by running the following:

```
model.summary()
```

This outputs the following:

Layer (type)	Output Shape	Param #
<hr/>		
conv2d_1 (Conv2D)	(None, 26, 26, 35)	350
<hr/>		
max_pooling2d_1 (MaxPooling2D)	(None, 13, 13, 35)	0
<hr/>		
conv2d_2 (Conv2D)	(None, 11, 11, 35)	11060
<hr/>		
max_pooling2d_2 (MaxPooling2D)	(None, 5, 5, 35)	0
<hr/>		
conv2d_3 (Conv2D)	(None, 3, 3, 45)	14220
<hr/>		
max_pooling2d_3 (MaxPooling2D)	(None, 1, 1, 45)	0
<hr/>		
flatten_1 (Flatten)	(None, 45)	0
<hr/>		
dense_1 (Dense)	(None, 64)	2944
<hr/>		
dense_2 (Dense)	(None, 32)	2080
<hr/>		
dense_3 (Dense)	(None, 10)	330
<hr/>		
Total params: 30,984		
Trainable params: 30,984		
Non-trainable params: 0		

Notice how the output shapes change as the data passes through the model. Specifically, look at the shape of our output after the flattening occurs—just 45 features. The raw data in `X_train` and `X_test` consisted of 784 features per row, so this is fantastic!

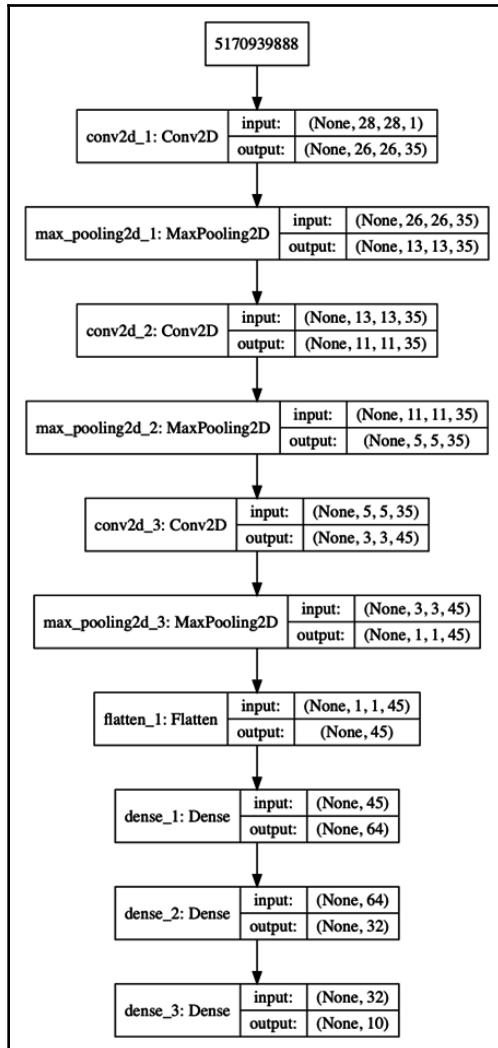


You'll need to install `pydot` to render the visualization. To install it, run `pip install pydot` from the terminal. You may need to restart your kernel for the install to take effect.

Using the `plot_model` function in Keras, we can visualize the topology of our network differently. To do this, run the following code:

```
plot_model(model, to_file='Conv_model1.png', show_shapes=True)
Image.open('Conv_model1.png')
```

Running the preceding code saves the topology to `Conv_model1.png` and generates the following:





This model will take several minutes to fit. If you have concerns about your system's hardware specs, you can easily reduce the training time by reducing the number of epochs to 10.

Running the following code block will fit the model:

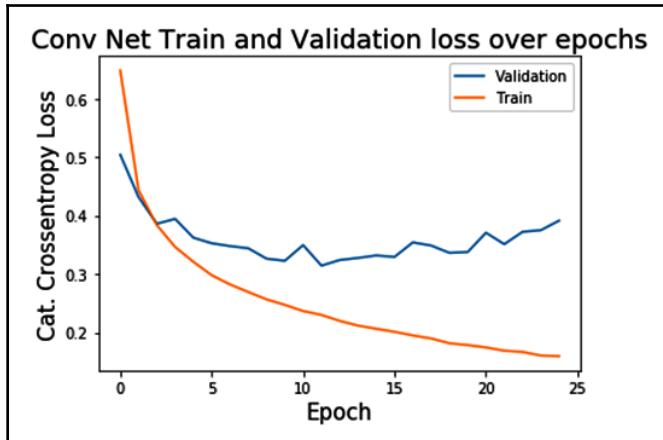
```
my_fit_model = model.fit(X_train, y_train, epochs=25, validation_data=(X_test, y_test))
```

In the fit step, we specify our `X_train` and `y_train`. We then specify the number of epochs we'd like to train the model. Then we plug in the validation data—`X_test` and `y_test`—to observe our model's out-of-sample performance. I like to save the `model.fit` step as a variable, `my_fit_model`, so we can later easily visualize the training and validation losses over epochs.

As the code runs, you'll see the model's train and validation loss, and accuracy after each epoch. Let's plot our model's train loss and validation loss using the following code:

```
plt.plot(my_fit_model.history['val_loss'], label="Validation")
plt.plot(my_fit_model.history['loss'], label = "Train")
plt.xlabel("Epoch", size=15)
plt.ylabel("Cat. Crossentropy Loss", size=15)
plt.title("Conv Net Train and Validation loss over epochs", size=18)
plt.legend();
```

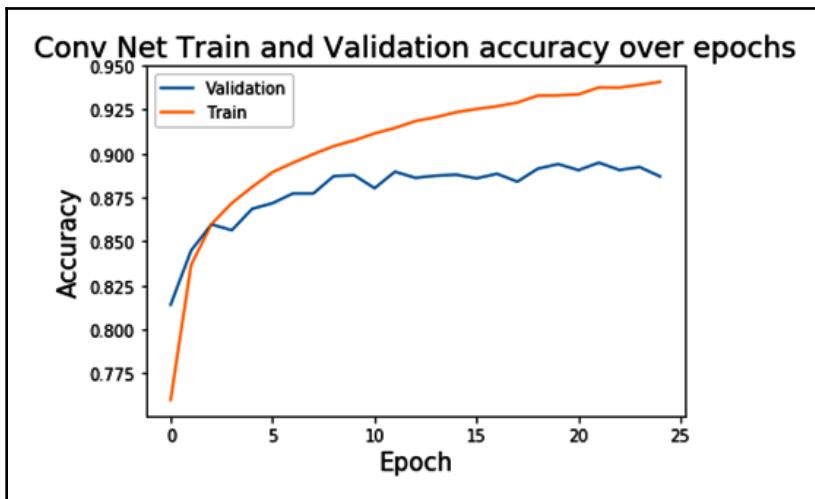
Running the preceding code generates the following plot. Your plot won't be identical—there are several stochastic processes taking place here—but it should look roughly the same:



A quick glance at this plot shows us that our model is overfitting. We see our train loss continue to fall in every epoch, but the validation loss doesn't move in lockstep. Let's glance at our accuracy scores to grasp how well this model did at the classification task. We can do this by running the following code:

```
plt.plot(my_fit_model.history['val_acc'], label="Validation")
plt.plot(my_fit_model.history['acc'], label = "Train")
plt.xlabel("Epoch", size=15)
plt.ylabel("Accuracy", size=15)
plt.title("Conv Net Train and Validation accuracy over epochs",
          size=18)
plt.legend();
```

This generates the following:



This plot, too, tells us we've overfit. But it appears as though our validation accuracy is in the high 80s, which is great! To get the max accuracy our model achieved and the epoch in which it occurred, we can run the following code:

```
print(max(my_fit_model.history['val_acc']))
print(my_fit_model.history['val_acc'].index(max(my_fit_model.history['v
al_acc'])))
```

Your specific results will differ from mine, but here's my output:

0.8948
21

Using our convolutional neural network, we achieved a max classification accuracy of 89.48% in the 21st epoch. This is amazing! But we've still got to address that overfitting problem. Next, we'll rebuild our model using **dropout regularization**.

Dropout regularization is a form of regularization we can apply to the fully-connected layers of our neural network. Using dropout regularization, we randomly drop neurons and their connections from the network during training. By doing this, the network doesn't become too reliant on the weights or biases associated with any specific node, allowing it to generalize better out of sample.

Here, we add dropout regularization, specifying that we'd like to drop 35% of the neurons at each `Dense` layer:

```
model = Sequential()
model.add(Conv2D(filters = 35, kernel_size=(3, 3), input_shape=
                 (28, 28, 1), activation='relu'))
model.add(MaxPool2D(pool_size=(2, 2)))
model.add(Conv2D(filters = 35, kernel_size=(3, 3), activation='relu'))
model.add(MaxPool2D(pool_size=(2, 2)))
model.add(Conv2D(filters = 45, kernel_size=(3, 3), activation='relu'))
model.add(MaxPool2D(pool_size=(2, 2)))
model.add(Flatten())
model.add(Dense(64, activation='relu'))
model.add(Dropout(0.35))
model.add(Dense(32, activation='relu'))
model.add(Dropout(0.35))
model.add(Dense(10, activation='softmax'))
model.compile(optimizer='adam', loss='categorical_crossentropy',
metrics=['accuracy'])
```

Running the preceding code will compile our new model. Let's have another look at the summary by rerunning the following:

```
model.summary()
```

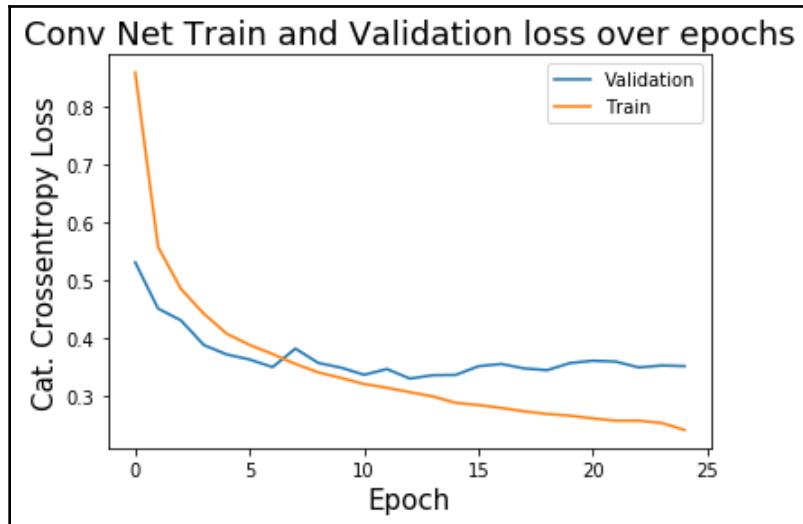
Running the preceding code returns the following output:

Layer (type)	Output Shape	Param #
conv2d_1 (Conv2D)	(None, 26, 26, 35)	350
max_pooling2d_1 (MaxPooling2D)	(None, 13, 13, 35)	0
conv2d_2 (Conv2D)	(None, 11, 11, 35)	11060
max_pooling2d_2 (MaxPooling2D)	(None, 5, 5, 35)	0
conv2d_3 (Conv2D)	(None, 3, 3, 45)	14220
max_pooling2d_3 (MaxPooling2D)	(None, 1, 1, 45)	0
flatten_1 (Flatten)	(None, 45)	0
dense_1 (Dense)	(None, 64)	2944
dropout_1 (Dropout)	(None, 64)	0
dense_2 (Dense)	(None, 32)	2080
dropout_2 (Dropout)	(None, 32)	0
dense_3 (Dense)	(None, 10)	330
<hr/>		
Total params: 30,984		
Trainable params: 30,984		
Non-trainable params: 0		

Let's refit our model by rerunning the following:

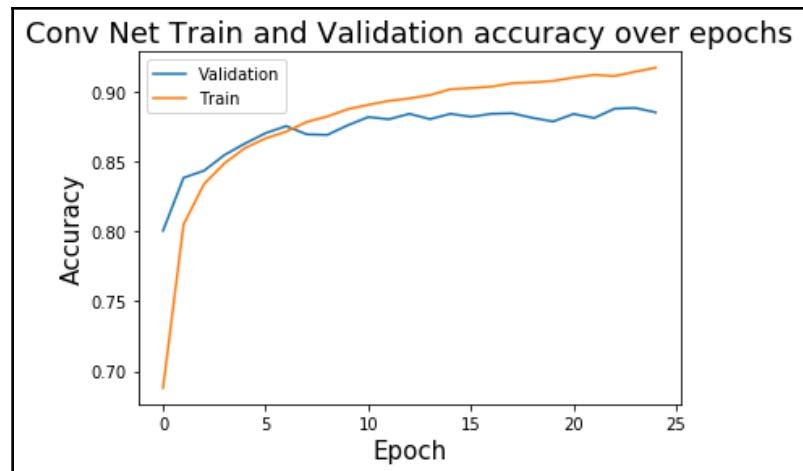
```
my_fit_model = model.fit(X_train, y_train, epochs=25, validation_data=(X_test, y_test))
```

Once your model has refit, rerun the plot code to visualize loss. Here's mine:



This looks better! The difference between our training and validation losses has shrunk, which was the intended purpose, though there does appear to be some room for improvement.

Next, re-plot your accuracy curves. Here are mine for this run:



This also looks better from an overfitting perspective. Fantastic! What was the best classification accuracy we achieved after applying regularization? Let's run the following code:

```
print(max(my_fit_model.history['val_acc']))
print(my_fit_model.history['val_acc'].index(max(my_fit_model.history['val_acc'])))
```

My output from this run of the model was as follows:

0.8885
23

Interesting! The best validation accuracy we achieved was lower than that in our unregularized model, but not by much. And it's still quite good! Our model is telling us that we predict the correct type of clothing article 88.85% of the time.

One way to think about how well we've done here is to compare our model's accuracy with the **baseline accuracy** for our dataset. The baseline accuracy is simply the score we would get by naïvely selecting the most-commonly occurring class in the dataset. For this specific dataset, because the classes are perfectly balanced and there are 10 classes, the baseline accuracy is 10%. Our model handily beats this baseline accuracy. It's clearly learned something about the data!

There are so many different places you can go from here! Try building deeper models or grid-searching over the many hyperparameters we used in our models. Assess your classifier's performance as you would with any other model—try building a confusion matrix to understand what classes we predicted well and what classes we weren't as strong in!

Summary

We certainly covered a lot of ground here! We talked about how to extract features from images, how convolutional neural networks work, and then we built a convolutional neural network to a fully-connected network architecture. Along the way, we picked up lots of new jargon and concepts, too!

Hopefully, after reading this chapter, you feel that these image-classification techniques—knowledge of which you may have once considered the province of sorcerers—is actually just a series of mathematical optimizations carried out for intuitive reasons! And hopefully this content can help move you forward in tackling an image-processing project that interests you!

9

Building a Chatbot

Imagine for a moment that you're sitting alone in a quiet, spacious room. To your right is a small table with a stack of white printer paper and a single black pen. In front of you is what seems to be a large, red cube with a tiny opening—slightly smaller than the size of a mail slot. An inscription just above the slot invites you to write down a question and pass it through the slot. As it happens, you speak Mandarin; so, you write down your question in Mandarin on one of the sheets and insert it into the opening. A few moments pass, and then slowly, an answer emerges. It's also written in Chinese and is the just the sort of answer you might have expected. So, what did you ask? *Are you a person or a computer?* And the response? *Why yes, yes I am.*

This thought experiment is based on philosopher John Searle's Chinese Room Argument. The premise of the experiment is that if there were a person in the room who spoke no Chinese, but had a set of rules that allowed them to perfectly map English characters to Chinese characters, they could appear to the questioner to understand Chinese without actually having any understanding of it. Searle's argument was that algorithmic procedures that produce an intelligible output can't be said to have an *understanding* of that output. They lack a *mind*. His thought experiment was an attempt to combat the ideas of *strong AI*, or the notion that the human brain is essentially just a *wet machine*. Searle didn't believe that AI could be said to have consciousness, no matter how sophisticated its behavior appeared to an outside observer.

Searle published this experiment in 1980. 31 years later, Siri would be released on the iPhone 4S. For anyone who has used Siri, it's clear that we have a long way to go before we might be confronted with uncertainty of whether the agent we are speaking to has a mind (though we might doubt it in people we know to be human). Despite the clunkiness these agents, or chatbots, have demonstrated in the past, the field is rapidly advancing.

In this chapter, we're going to learn how to construct a chatbot from scratch. Along the way, we'll learn more about the history of the field and its future prospects.

We'll cover the following topics in this chapter:

- The Turing Test
- The history of chatbots
- The design of chatbots
- Building a chatbot

The Turing Test

30 years before Searle's Chinese Room, Alan Turing posed the question, *can machines think?* in one of his more famous papers. Being the practical genius he was, he decided not to tackle that question head on, but to instead pose it in the framework of the *problem of other minds*. This problem asks, *how do we truly know that other people have minds like our own?* Since we can only observe their behavior—and not the inner workings of their mind—we must take it on faith that they are like us. Turing proposed that if a machine could behave as if it were intelligent, then we should view it as such. This, in a nutshell, is the *Turing Test*. Passing the test means convincing humans that a machine is a fellow human.

You might be wondering at this point, *has a program ever successfully passed the Turing Test?* And the answer is that there really is no official Turing Test. The closest thing to an official test is the Loebner Prize. This is a contest held yearly that awards modest prizes to those chatbots that appear the most convincingly human to a panel of judges. If this sounds rigorous and academic, it has been described as anything but. Loebner himself is said to be quite the character, frequently professing his love of wine, marijuana, and prostitutes. His sponsorship for the contest is said to have been a product of his total disdain for labor of any kind.

Hopefully, by this point, you'll have realized that any claims about *passing the Turing Test* are more spectacle than actual science. Despite this, Turing's original concept is a useful construct.

The history of chatbots

I still remember the day my father sat me down in front of his IBM XT with its black-and-green flickering monitor and asked me to talk to *it*. The conversation went something like the following:

>Hello. How are you feeling today?
>Hi. Who is this?
>Hi there... How are you today?
>I'm fine. You didn't tell me who you are.
>We should be discussing you, not me.
>We should? Why?
>Let's change focus a bit... Tell me about your family.
>You are impossible to have a conversation with.
>Can you elaborate on that?
>You are killing me.
>Perhaps you would like me to be killing you.
>OK, you are freaking me out.
>Perhaps you're really talking about yourself?
>What?? That makes no sense.
>How would an answer to that help you?
>END
>Very interesting.
>QUIT
>Thank you for talking with me.
>#\$##\$#@@....

The *it* I was talking to was a program called **ELIZA**. Written in the mid-1960s at MIT's Artificial Intelligence Lab by Joseph Weizenbaum, ELIZA was created to mimic the responses of a Rogerian psychotherapist. Though nearly comical when examined in any depth, the program was capable of convincing some users that they were chatting to an actual human—a remarkable feat considering it was a scant 200 lines of code that used randomization and regular expressions to parrot back responses. Even today, this simple program remains a staple of popular culture. If you ask Siri who ELIZA is, she'll tell you she's a friend and a brilliant psychiatrist.

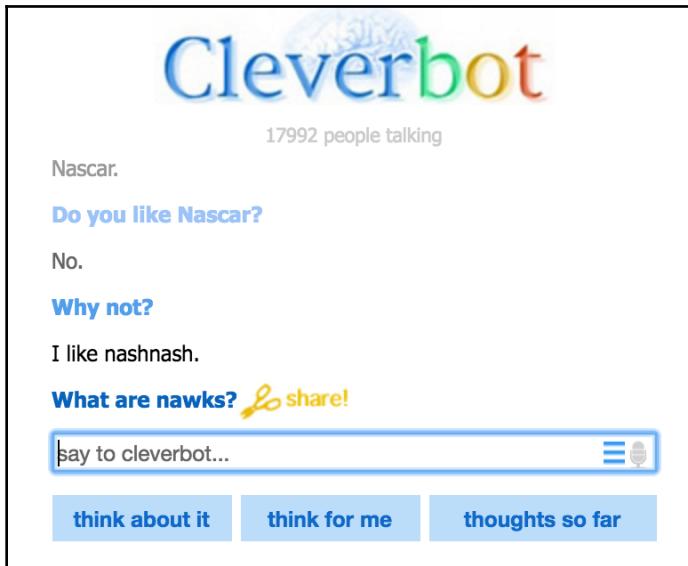
If ELIZA was an early example of chatbots, what have we seen since that time? In recent years, there has been an explosion of new chatbots. The most notable of these is Cleverbot.

Cleverbot was released to the world using the web in 1997. In the years since, the bot has racked up hundreds of millions of conversations, and, unlike early chatbots, Cleverbot, as its name suggests, appears to become more intelligent with each conversion. Though the exact details of the workings of the algorithm are difficult to find, it's said to work by recording all conversations in a database and finding the most appropriate response by identifying the most similar questions and responses in the database.

I made up a nonsensical question, shown as follows, and you can see that it found something similar to the object of my question in terms of a string match:



I persisted:



And, again, I got something... similar?

You'll also notice that topics can persist across the conversation. In response, I was asked to go into more detail and justify my answer. This is one of the things that appears to make Cleverbot, well, clever.

While chatbots that learn from humans can be quite amusing, they can also have a darker side.

Several years ago, Microsoft released a chatbot named Tay on to Twitter. People were invited to ask Tay questions, and Tay would respond in accordance with her *personality*. Microsoft had apparently programmed the bot to appear to be a 19-year-old American girl. She was intended to be your virtual *bestie*; the only problem was that she started tweeting out extremely racist remarks.

As a result of these unbelievably inflammatory tweets, Microsoft was forced to pull Tay off Twitter and issue an apology.

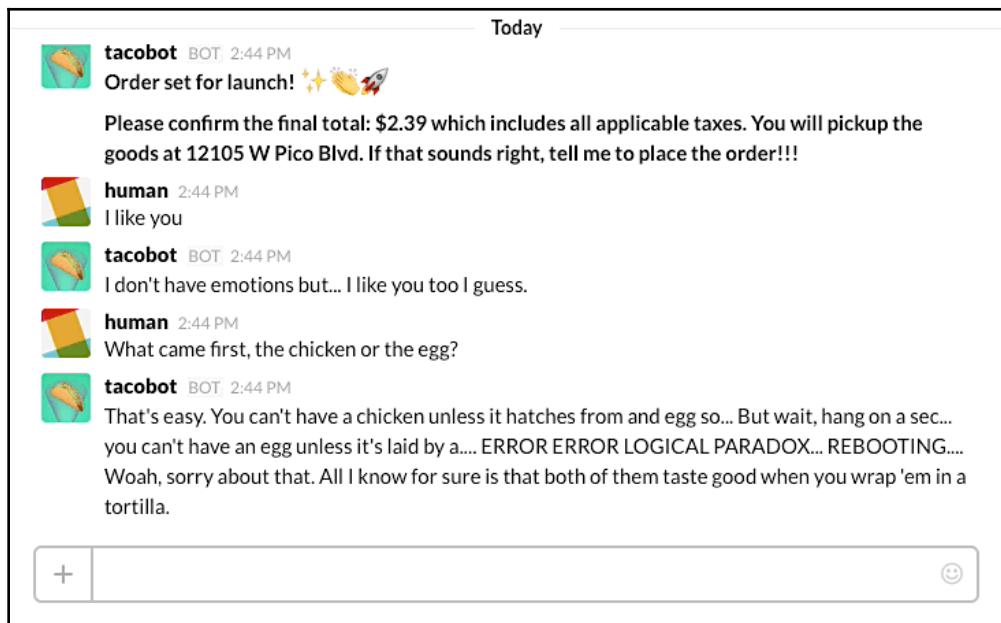
"As many of you know by now, on Wednesday we launched a chatbot called Tay. We are deeply sorry for the unintended offensive and hurtful tweets from Tay, which do not represent who we are or what we stand for, nor how we designed Tay. Tay is now offline and we'll look to bring Tay back only when we are confident we can better anticipate malicious intent that conflicts with our principles and values."

-March 25, 2016 Official Microsoft Blog

Clearly, brands that want to release chatbots into the wild in the future should take a lesson from this debacle and plan for users to attempt to manipulate them to display the worst of human behavior.

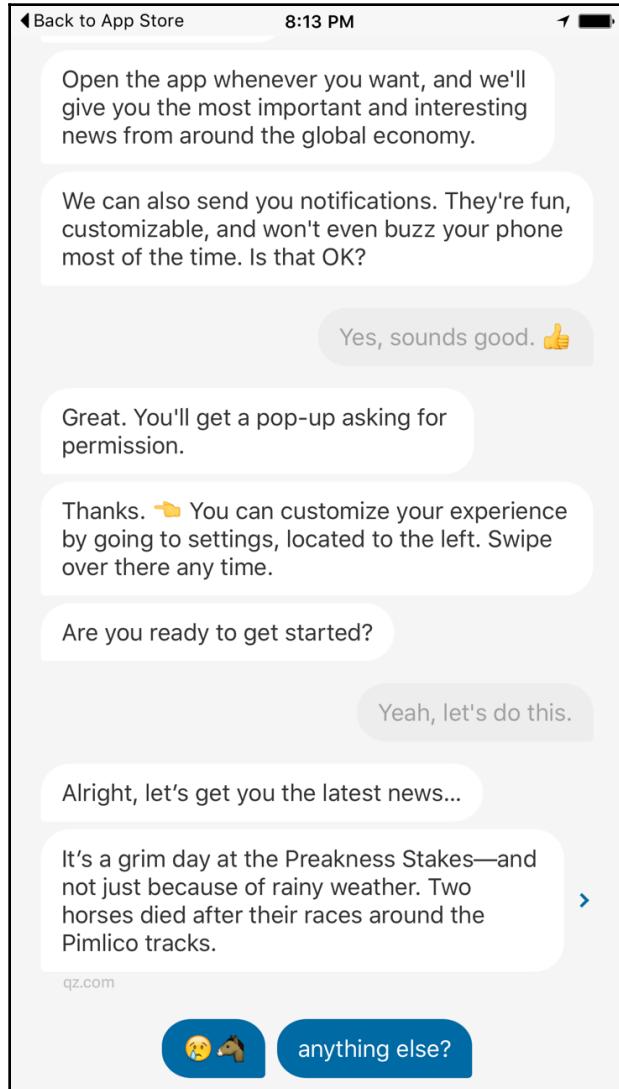
There's no doubt that brands are embracing chatbots. Everyone from Facebook to Taco Bell is getting in on the game.

Witness the TacoBot:



Yes, it's a real thing. And, despite the stumbles, like Tay, there's a good chance the future of UI looks a lot like TacoBot. One last example might even help explain why.

Quartz recently launched an app that turns news into a conversation. Rather than lay out the day's stories as a flat list, you are engaged in a chat as if you were getting news from a friend:



David Gasca, a PM at Twitter, describes his experience using the app in a post on Medium. He describes how the conversational nature invoked feelings normally only triggered in human relationships:

"Unlike a simple display ad, in a conversational relationship with my app I feel like I owe something to it: I want to click. At the most subconscious level I feel the need to reciprocate and not let the app down: "The app has given me this content. It's been very nice so far and I enjoyed the GIFs. I should probably click since it's asking nicely."

If that experience is universal—and I expect it is—this could be the next big thing in advertising, and I have no doubt that advertising profits will drive UI design:

"The more the bot acts like a human, the more it will be treated like a human."

-Mat Webb, Technologist and Co-Author of Mind Hacks

At this point, you're probably dying to know how these things work, so let's get on with it!

The design of chatbots

The original ELIZA application was 200-odd lines of code. The Python NLTK implementation is similarly short. An excerpt is provided from NLTK's website (http://www.nltk.org/_modules/nltk/chat/eliza.html):

```
# Natural Language Toolkit: Eliza
#
# Copyright (C) 2001-2019 NLTK Project
# Authors: Steven Bird <stevenbird1@gmail.com>
#          Edward Loper <edloper@gmail.com>
# URL: <http://nltk.org/>
# For license information, see LICENSE.TXT

# Based on an Eliza implementation by Joe Strout <joe@strout.net>,
# Jeff Epler <jepler@inetnebr.com> and Jez Higgins <mailto:jez@jezuk.co.uk>.

# a translation table used to convert things you say into things the
# computer says back, e.g. "I am" --> "you are"

from __future__ import print_function
from nltk.chat.util import Chat, reflections

# a table of response pairs, where each pair consists of a
# regular expression, and a list of possible responses,
# with group-macros labelled as %1, %2.

pairs = (
    (
        r'I need (.*)',
        (
            "Why do you need %1?",
            "Would it really help you to get %1?",
            "Are you sure you need %1?",
        ),
    ),
    (
        r'Why don\'t you (.*)',
        (
            "Do you really think I don't %1?",
            "Perhaps eventually I will %1.",
            "Do you really want me to %1?",
        ),
    ),
)
```

As you can see from the code, input text was parsed and then matched against a series of regular expressions. Once the input was matched, a randomized response (that sometimes echoed back a portion of the input) was returned. So, something such as, *I need a taco* would trigger a response of, *Would it really help you to get a taco?* Obviously, the answer is yes, and, fortunately, we have advanced to the point that technology can provide you one (bless you, TacoBot), but this was early days still. Shockingly, some people actually believed ELIZA was a real human.

But what about more advanced bots? How are they built?

Surprisingly, most chatbots you're likely to encounter aren't even using **machine learning (ML)**; they're what's known as **retrieval-based** models. This means responses are predefined according to the question and the context. The most common architecture for these bots is something called **Artificial Intelligence Markup Language (AIML)**. AIML is an XML-based schema for representing how the bot should interact given the user's input. It's really just a more advanced version of how ELIZA works.

Let's take a look at how responses are generated using AIML. First, all input are preprocessed to normalize them. This means when you input *Waaazzup???* it's mapped to *WHAT IS UP*. This preprocessing step funnels down the myriad ways of saying the same thing into one input that can run against a single rule. Punctuation and other extraneous input are removed as well at this point. Once that's complete, the input is matched against the appropriate rule. The following is a sample template:

```
<category>
<pattern>WHAT IS UP</pattern>
<template>The sky, duh. Pfft. Humans...</template>
</category>
```

That is the basic setup, but you can also layer in wildcards, randomization, and prioritization schemes. For example, the following pattern uses wildcard matching:

```
<category>
<pattern>* FOR ME</pattern>
<template>I'm a bot. I don't <star/>. Ever.</template>
</category>
```

Here, the * wildcard matches one or more words before FOR ME and then repeats those back in the output template. If the user were to type in *Dance for me!*, the response would be *I'm a bot. I don't dance. Ever.*

As you can see, these rules don't make for anything that approximates any type of real intelligence, but there are a few tricks that strengthen the illusion. One of the better ones is the ability to generate responses conditioned on a topic.

For example, here's a rule that invokes a topic:

```
<category>
<pattern>I LIKE TURTLES</pattern>
<template>I feel like this whole <set name="topic">turtle</set> thing could
be a problem. What do you like about them? </template>
</category>
```

Once the topic is set, the rules specific to that context can be matched:

```
<topic name="turtles">

<category>
<pattern>* SHELL IS *</pattern>
<template>I dislike turtles primarily because of their shells. What other
creepy things do you like about turtles? </template>
</category>

<category>
<pattern>* HIDE *</pattern>
<template>I wish, like a turtle, that I could hide from this conversation.
</template>
</category>

</topic>
```

Let's see what this interaction might look like:

>I like turtles!

>I feel like this whole turtle thing could be a problem. What do you like about them?

>I like how they hide in their shell.

>I wish, like a turtle, I could hide from this conversation.

You can see that the continuity across the conversation adds a measure of realism.

You're probably thinking that this can't be state of the art in this age of deep learning, and you're right. While most bots are rule-based, the next generation of chatbots are emerging, and they're based on neural networks.

In 2015, Oriol Vinyas and Quoc Le of Google published a paper, <http://arxiv.org/pdf/1506.05869v1.pdf>, that described the construction of a neural network based on sequence-to-sequence models. This type of model maps an input sequence, such as *ABC*, to an output sequence, such as *XYZ*. These inputs and outputs might be translations from one language to another, for example. In the case of their work here, the training data was not language translation, but rather tech support transcripts and movie dialogues. While the results from both models are interesting, it was the interactions based on the movie model that stole the headlines.

The following are sample interactions taken from the paper:

Human: <i>who is skywalker ?</i> Machine: <i>he is a hero .</i> Human: <i>who is bill clinton ?</i> Machine: <i>he 's a billionaire .</i> Human: <i>is sky blue or black ?</i> Machine: <i>blue .</i> Human: <i>does a cat have a tail ?</i> Machine: <i>yes .</i> Human: <i>does a cat have a wing ?</i> Machine: <i>no</i> Human: <i>can a cat fly ?</i> Machine: <i>no .</i> Human: <i>how many legs does a cat have ?</i> Machine: <i>four , i think .</i>

None of this was explicitly encoded by humans or present in the training set as asked, and, yet, looking at this, it's frighteningly like speaking with a human. But let's see more:

Human: <i>what do you think about tesla ?</i> Machine: <i>he 's a good conductor .</i> Human: <i>what do you think about bill gates ?</i> Machine: <i>he 's a good man .</i> Human: <i>What do you think about messi ?</i> Machine: <i>he 's a great player .</i> Human: <i>what do you think about cleopatra ?</i> Machine: <i>oh , she 's very regal .</i> Human: <i>what do you think about england during the reign of elizabeth ?</i> Machine: <i>it was a great place .</i>
--

Notice that the model is responding with what appears to be knowledge of gender (**he**, **she**), place (England), and career (**player**). Even questions of meaning, ethics, and morality are fair game:

Human: *what is the purpose of life ?*
Machine: *to serve the greater good .*
Human: *what is the purpose of living ?*
Machine: *to live forever .*

Human: *okay , i need to know how should i behave morally ...*
Machine: *i don 't know how to tell you .*
Human: *is morality and ethics the same ?*
Machine: *yes , sir .*
Human: *what are the things that i do to be immoral ?*
Machine: *i don 't know .*
Human: *give me some examples of moral actions ...*
Machine: *i 'm not a moralist .*
Human: *alright , morality ?*
Machine: *integrity .*

If that transcript doesn't give you a slight chill, there's a chance you might already be some sort of AI.

I wholeheartedly recommend reading the entire paper. It isn't overly technical, and it will definitely give you a glimpse of where the technology is headed.

We've talked a lot about the history, types, and design of chatbots, but let's now move on to building our own. We'll take two approaches to this. This first will use a technique we saw in previously, cosine similarity, and the second will leverage sequence-to-sequence learning.

Building a chatbot

Now, having seen what's possible in terms of chatbots, you most likely want to build the best, most state-of-the-art, Google-level bot out there, right? Well, just put that out of your mind for now because we're going start by doing the exact opposite. We're going to build the most amazingly awful bot ever!

This may sound disappointing, but if your goal is just to build something very cool and engaging (that doesn't take hours and hours to construct), this is a great place to start.

We're going to leverage the training data derived from a set of real conversations with Cleverbot. The data was collected from <http://notsocleverbot.jimrule.com>. This site is perfect, as it has people submit the most absurd conversations they had with Cleverbot.

Let's take a look at a sample conversation between Cleverbot and a user from the site:

```
User: *lightsaber battle*.  
Cleverbot: Luke I am your father.  
User: *swings with lightsaber*.  
Cleverbot: *Blocks* *Swings lightsaber*.  
User: *blocks stabs with lightsaber*.  
Cleverbot: *pushes back with the force*.  
User: *pulls out gun*.  
Cleverbot: *Pulls out nuke*.  
User: *pulls out chuck norris*.  
Cleverbot: *pulls out God*.  
User: *pulls out universe*.  
Cleverbot: *pulls out his mom*.  
User: Mommy... *sobs*.  
Cleverbot: *curls up and cries*.  
User: *cuddles and cries*.  
Cleverbot: *kisses neck softly*.  
User: *cries more*.  
Cleverbot: *kisses tears more*.  
User: *cries even more*.  
Cleverbot: *Whispers* It's ok.  
User: *still sobs*.  
Cleverbot: *snaps neck*.  
User: WTF?!
```

While you are free to use the techniques for web scraping that we used in earlier chapters to collect the data, you can find a .csv of the data in the GitHub repo for this chapter.

We'll start again in our Jupyter Notebook. We'll load, parse, and examine the data. We'll first import pandas and the Python regular expressions library, re. We're also going to set the option in pandas to widen our column width so we can see the data better:

```
import pandas as pd  
import re  
pd.set_option('display.max_colwidth', 200)
```

Now we'll load in our data:

```
df = pd.read_csv('nscb.csv')  
df.head()
```

The preceding code results in the following output:

	Postinner link	Postinner link_link	Time value	Up number	Image 1	Down number
0	User: *walking*\nCleverbot: *looks at you and winks* ;)\nUser: O.O *walks faster*\nCleverbot: *catches up with you and pins you against a wall*\nUser: *eyes widen* W-What do you want?\nCleverbot: ...	http://www.notsocleverbot.com/index.php?i=48277	Posted: 2016-04-18 04:30:53	21	http://www.notsocleverbot.com/images/thumbs_up.png	10
1	you.\nUser: So you admit your idiocy.\nCleverbot: You're all and I am nothing.\nUser: Yess!!!\nCleverbot: ...	http://www.notsocleverbot.com/index.php?i=48275	Posted: 2016-04-16 22:39:34	32	http://www.notsocleverbot.com/images/thumbs_up.png	5
2	Cleverbot: Well, my wife left me today.\nUser: Feels bad for you.\nCleverbot: Not really, I have an amazing boyfriend :).	http://www.notsocleverbot.com/index.php?i=48256	Posted: 2016-04-09 04:49:46	13	http://www.notsocleverbot.com/images/thumbs_up.png	5

Since we're only interested in the first column, the conversation data, we'll parse that out:

```
convo = df.iloc[:, 0]
convo
```

The preceding code results in the following output:

```
0      User: *walking*\nCleverbot: *looks at you and winks* ;)\nUser: O.O *walks faster*\nCleverbot: *catches up with you and pins you against a wall*\nUser: *eyes widen* W-What do you want?\nCleverbot: ...
1      User: *Facepalm* You are so stupid, you know that, right?\nCleverbot: I cannot argue with you.\nUser: So you admit your idiocy.\nCleverbot: You're all and I am nothing.\nUser: Yess!!!\nCleverbot: ...
2      Cleverbot: Well, my wife left me today.\nUser: Feels bad for you.\nCleverbot: Not really, I have an amazing boyfriend :).
3      Cleverbot: *my dragongirl chest fills with dragon milk*.nUser: WHAT THE ACTUAL FUC K?!?!?\nCleverbot: *gives birth to dragon baby*\nUser: WHAT. THE. FUCK. HAPPENED. TO. CLEVERBOT.
4      Cleverbot: He'll have to sit and watch them all.\nUser: What the frick are you talking about.\nCleverbot: Something more pleasant than you!
5      Cleverbot: Give me a riddle.\nUser: What has a head but no body . has a bed but never sleeps . runs but never walks.\nCleverbot: What? I don't know.\nUser: A river.\nCleverbot: A river is not a fl...
6      User: I'm commander shepard.\nCleverbot: Really?\nUser: Yupp.\nCleverbot: That's nice. Can I meet your friend?\nUser: Garrus or Tali.\nCleverbot: Garrus because he is more impressive.\nUser: Okay ...
7      Cleverbot: Never gonna give you up.\nUser: No, please no.\nCleverbot: Never gonna tell a lie and hurt yo
```

You should be able to make out that we have interactions between **User** and **Cleverbot**, and that either can initiate the conversation. To get the data in the format we need, we'll have to parse it into question-and-response pairs. We aren't necessarily concerned with who says what, but with matching up each response to each question. You'll see why in a bit. Let's now do a bit of regular expression magic on the text:

```
clist = []
def qa_pairs(x):
    cpairs = re.findall(": (.*)?(?:$|\n)", x)
    clist.extend(list(zip(cpairs, cpairs[1:])))

convo.map(qa_pairs);
convo_frame = pd.Series(dict(clist)).to_frame().reset_index()
convo_frame.columns = ['q', 'a']
```

The preceding code results in the following output:

	q	a
0	*walking*	*looks at you and winks* ;)
1	*looks at you and winks* ;)	O.O *walks faster*
2	O.O *walks faster*	*catches up with you and pins you against a wall*
3	*catches up with you and pins you against a wall*	*eyes widen* W-What do you want?
4	*eyes widen* W-What do you want?	Your mom.
5	*Facepalm* You are so stupid, you know that, right?	I cannot argue with you.
6	I cannot argue with you.	So you admit your idiocy.
7	So you admit your idiocy.	You're all and I am nothing.

OK, lots of code there. What just happened? We first created a list to hold our question-and-response tuples. We then passed our conversations through a function to split them into those pairs using regular expressions.

Finally, we set it all into a pandas DataFrame with columns labelled q and a.

We're now going to apply a bit of algorithm magic to match up the closest question to the one a user inputs:

```
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.metrics.pairwise import cosine_similarity

vectorizer = TfidfVectorizer(ngram_range=(1, 3))
vec = vectorizer.fit_transform(convo_frame['q'])
```

In the preceding code, we imported our tf-idf vectorization library and the cosine similarity library. We then used our training data to create a tf-idf matrix. We can now use this to transform our own new questions and measure the similarity to existing questions in our training set. Let's do that now:

```
my_q = vectorizer.transform(['Hi. My name is Alex.'])

cs = cosine_similarity(my_q, vec)

rs = pd.Series(cs[0]).sort_values(ascending=False)
top5 = rs.iloc[0:5]
top5
```

The preceding code results in the following output:

19522	0.638891
70033	0.537884
51408	0.531098
20291	0.528135
62947	0.460475
dtype: float64	

What are we looking at here? This is the cosine similarity between the question I asked and the top-five closest questions. On the left is the index, and on the right is the cosine similarity. Let's take a look at those:

```
convo_frame.iloc[top5.index]['q']
```

This results in the following output:

19522	Hi my name is Cleverbot.
70033	Okay your name is Alex.
51408	Hi my name is pat
20291	Hi my name is lune.
62947	My name is P.J.
Name: q, dtype: object	

As you can see, nothing is exactly the same, but there are definitely some similarities.

Let's now take a look at the response:

```
rsi = rs.index[0]
rsi

convo_frame.iloc[rsi]['a']
```

The preceding code results in the following output:

'Are you clever or something.'

OK, so our bot seems to have an attitude already. Let's push further.

We'll create a handy function so that we can test a number of statements easily:

```
def get_response(q):
    my_q = vectorizer.transform([q])
    cs = cosine_similarity(my_q, vec)
    rs = pd.Series(cs[0]).sort_values(ascending=False)
    rsi = rs.index[0]
    return convo_frame.iloc[rsi]['a']

get_response('Yes, I am clearly more clever than you will ever be!')
```

This results in the following output:

'No you are not.'

We have clearly created a monster, so we'll continue:

```
get_response('You are a stupid machine. Why must I prove anything to
you?')
```

This results in the following output:

'Right you are.'

I'm enjoying this. Let's keep rolling with it:

```
get_response('Did you eat tacos?')
```

'No, I had nachos'

```
get_response('With beans on top?')
```

'Apples.'

```
get_response('What else do you like to do?')
```

'I also like talking to you.'

```
get_response('What do you like about it?')
```

'I like your randomness.'

```
get_response('Me, random?')
```

'Yes your random.'

```
get_response('I think you mean you\'re')
```

'DONT CORRECT ME!'

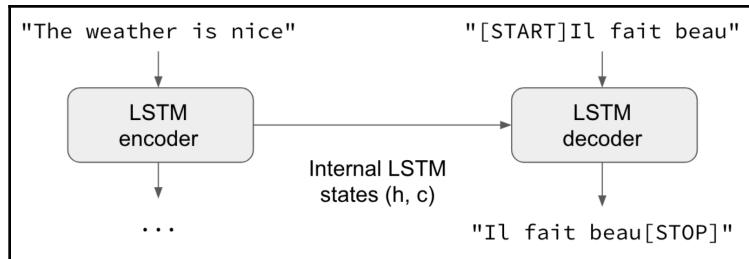
Remarkably, this may be one of the best conversations I've had in a while, bot or not.

Now while this was a fun little project, let's now move on to a more advanced modeling technique using sequence-to-sequence modeling.

Sequence-to-sequence modeling for chatbots

For this next task, we'll leverage a couple libraries discussed in Chapter 8, *Classifying Images with Convolutional Neural Networks*, TensorFlow and Keras. Both can be pip installed if you haven't done that already.

We're also going to use the type of advanced modeling discussed earlier in the chapter; it's a type of deep learning called **sequence-to-sequence modeling**. This is frequently used in machine translation and question-answering applications, as it allows us to map an input sequence of any length to an output sequence of any length:



Source: <https://blog.keras.io/a-ten-minute-introduction-to-sequence-to-sequence-learning-in-keras.html>

Francois Chollet has an excellent introduction to this type of model on the blog for Keras: <https://blog.keras.io/a-ten-minute-introduction-to-sequence-to-sequence-learning-in-keras.html>. It's worth a read.

We're going to make heavy use of his example code to build out our model. While his example uses machine translation, English to French, we're going to repurpose it for question-answering using our Cleverbot dataset:

1. Set the imports:

```
from keras.models import Model  
from keras.layers import Input, LSTM, Dense  
import numpy as np
```

2. Set up the training parameters:

```
batch_size = 64 # Batch size for training.  
epochs = 100 # Number of epochs to train for.  
latent_dim = 256 # Latent dimensionality of the encoding space.  
num_samples = 1000 # Number of samples to train on.
```

We'll use these to start. We can examine the success of our model and then adjust as necessary.

The first step in data processing will be to take our data, get it in the proper format, and then vectorize it. We'll go step by step:

```
input_texts = []
target_texts = []
input_characters = set()
target_characters = set()
```

This creates lists for our questions and answers (the targets) as well as sets for the individual characters in our questions and answers. This model will actually work by generating one character at a time:

1. Let's limit our question-and-answer pairs to 50 characters or fewer. This will help speed up our training:

```
convo_frame['q len'] = convo_frame['q'].astype('str').apply(lambda
    x: len(x))
convo_frame['a len'] = convo_frame['a'].astype('str').apply(lambda
    x: len(x))
convo_frame = convo_frame[(convo_frame['q len'] < 50) &
                           (convo_frame['a len'] < 50)]
```

2. Let's set up our input and target text lists:

```
input_texts = list(convo_frame['q'].astype('str'))
target_texts = list(convo_frame['a'].map(lambda x: '\t' + x +
                                         '\n').astype('str'))
```

The preceding code gets our data in the proper format. Note that we add a tab (\t) and a newline (\n) character to the target texts. This will serve as our start and stop tokens for the decoder.

3. Let's take a look at the input texts and the target texts:

```
input_texts
```

The preceding code generates the following output:

```
[ '*walking*',
  '*looks at you and winks* ;)',
  'O.O *walks faster*',
  '*catches up with you and pins you against a wall*',
  '*eyes widen* W-What do you want?',
  '*Facepalm* You are so stupid, you know that, right?',
  'I cannot argue with you.',
  'So you admit your idiocy.',
  "You're all and I am nothing.",
  'Yess!!!',
```

target_texts

The preceding code generates the following output:

```
[ '\t*looks at you and winks* ;)\n',
  '\tO.O *walks faster*\n',
  '\t*catches up with you and pins you against a wall*\n',
  '\t*eyes widen* W-What do you want?\n',
  '\tYour mom.\n',
  '\tI cannot argue with you.\n',
  '\tSo you admit your idiocy.\n',
  "\tYou're all and I am nothing.\n",
  '\tYess!!!\n',
  '\tYes.\n',
  '\tFeels bad for you.\n',
```

Let's take a look at those input and target-character sets now:

input_characters

The preceding code generates the following output:

```
{',  
!',  
",  
'#',  
'$',  
'%',  
'&',  
'"',  
'(',  
)',  
'*',  
'+',  
'-',  
'.',  
'/' ,
```

```
target_characters
```

The preceding code generates the following output:

```
{'\t',  
\n',  
,  
'!',  
",  
'#',  
'$',  
'%',  
'&',  
'"',  
'(',  
)',  
'*',  
'+',  
'-',  
'.' ,
```

Next, we'll do some additional preparation for the data that will feed into the model. Although data can be fed in any length and returned in any length, we need to add padding up to the max length of the data for the model to work:

```
input_characters = sorted(list(input_characters))
target_characters = sorted(list(target_characters))
num_encoder_tokens = len(input_characters)
num_decoder_tokens = len(target_characters)
max_encoder_seq_length = max([len(txt) for txt in input_texts])
max_decoder_seq_length = max([len(txt) for txt in target_texts])

print('Number of samples:', len(input_texts))
print('Number of unique input tokens:', num_encoder_tokens)
print('Number of unique output tokens:', num_decoder_tokens)
print('Max sequence length for inputs:', max_encoder_seq_length)
print('Max sequence length for outputs:', max_decoder_seq_length)
```

The preceding code generates the following output:

Number of samples: 1000
Number of unique input tokens: 383
Number of unique output tokens: 362
Max sequence length for inputs: 49
Max sequence length for outputs: 51

Next, we'll vectorize our data using one-hot encoding:

```
input_token_index = dict(
    [(char, i) for i, char in enumerate(input_characters)])
target_token_index = dict(
    [(char, i) for i, char in enumerate(target_characters)])

encoder_input_data = np.zeros(
    (len(input_texts), max_encoder_seq_length, num_encoder_tokens),
    dtype='float32')
decoder_input_data = np.zeros(
    (len(input_texts), max_decoder_seq_length, num_decoder_tokens),
    dtype='float32')
decoder_target_data = np.zeros(
    (len(input_texts), max_decoder_seq_length, num_decoder_tokens),
    dtype='float32')

for i, (input_text, target_text) in enumerate(zip(input_texts,
target_texts)):
    for t, char in enumerate(input_text):
        encoder_input_data[i, t, input_token_index[char]] = 1.
    for t, char in enumerate(target_text):
```

```

# decoder_target_data is ahead of decoder_input_data by one
# timestep
decoder_input_data[i, t, target_token_index[char]] = 1.
if t > 0:
    # decoder_target_data will be ahead by one timestep
    # and will not include the start character.
    decoder_target_data[i, t - 1, target_token_index[char]] =
        1.

```

Let's take a look at one of these vectors:

Decoder_input_data

The preceding code generates the following output:

```

array([[[1., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.],
       ...,
       [0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.]],

      [[1., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.],

```

From the preceding figure, you'll notice that we have a one-hot encoded vector of our character data, which will be used in our model.

We now set up our sequence-to-sequence model-encoder and -decoder LSTMs:

```

# Define an input sequence and process it.
encoder_inputs = Input(shape=(None, num_encoder_tokens))
encoder = LSTM(latent_dim, return_state=True)
encoder_outputs, state_h, state_c = encoder(encoder_inputs)
# We discard `encoder_outputs` and only keep the states.
encoder_states = [state_h, state_c]

# Set up the decoder, using `encoder_states` as initial state.
decoder_inputs = Input(shape=(None, num_decoder_tokens))

# We set up our decoder to return full output sequences,
# and to return internal states as well. We don't use the
# return states in the training model, but we will use them in
# inference.
decoder_lstm = LSTM(latent_dim, return_sequences=True,
                    return_state=True)

```

```
decoder_outputs, _, _ = decoder_lstm(decoder_inputs,
                                      initial_state=encoder_states)
decoder_dense = Dense(num_decoder_tokens, activation='softmax')
decoder_outputs = decoder_dense(decoder_outputs)
```

Then we move on to the model itself:

```
# Define the model that will turn
# `encoder_input_data` & `decoder_input_data` into `decoder_target_data`
model = Model([encoder_inputs, decoder_inputs], decoder_outputs)

# Run training
model.compile(optimizer='rmsprop', loss='categorical_crossentropy')
model.fit([encoder_input_data, decoder_input_data],
          decoder_target_data,
          batch_size=batch_size,
          epochs=epochs,
          validation_split=0.2)
# Save model
model.save('s2s.h5')
```

In the preceding code, we defined our model using our encoder and decoder input and our decoder output. We then compile it, fit it, and save it.

We set up the model to use 1,000 samples. Here, we also split the data 80/20 to training and validation, respectively. We also set our epochs at 100, so this will essentially run for 100 cycles. On a standard MacBook Pro, this may take around an hour to complete.

Once that cell is run, the following output will be generated:

```
Train on 800 samples, validate on 200 samples
Epoch 1/100
800/800 [=====] - 12s 16ms/step - loss: 1.7601 - val_loss: 1.4434
Epoch 2/100
800/800 [=====] - 11s 13ms/step - loss: 1.4279 - val_loss: 1.4251
Epoch 3/100
800/800 [=====] - 10s 13ms/step - loss: 1.4096 - val_loss: 1.4195
Epoch 4/100
800/800 [=====] - 10s 13ms/step - loss: 1.3994 - val_loss: 1.4078
Epoch 5/100
800/800 [=====] - 10s 13ms/step - loss: 1.3914 - val_loss: 1.4038
Epoch 6/100
800/800 [=====] - 10s 13ms/step - loss: 1.3841 - val_loss: 1.3981
Epoch 7/100
800/800 [=====] - 11s 14ms/step - loss: 1.3784 - val_loss: 1.3855
Epoch 8/100
800/800 [=====] - 11s 14ms/step - loss: 1.3752 - val_loss: 1.3921
```

The next step is our inference step. We'll use the states generated from this model to feed into our next model to generate our output:

```
# Next: inference mode (sampling).  
# Here's the drill:  
# 1) encode input and retrieve initial decoder state  
# 2) run one step of decoder with this initial state  
# and a "start of sequence" token as target.  
# Output will be the next target token  
# 3) Repeat with the current target token and current states  
  
# Define sampling models  
encoder_model = Model(encoder_inputs, encoder_states)  
  
decoder_state_input_h = Input(shape=(latent_dim,))  
decoder_state_input_c = Input(shape=(latent_dim,))  
decoder_states_inputs = [decoder_state_input_h, decoder_state_input_c]  
decoder_outputs, state_h, state_c = decoder_lstm(  
    decoder_inputs, initial_state=decoder_states_inputs)  
decoder_states = [state_h, state_c]  
decoder_outputs = decoder_dense(decoder_outputs)  
decoder_model = Model(  
    [decoder_inputs] + decoder_states_inputs,  
    [decoder_outputs] + decoder_states)  
  
# Reverse-lookup token index to decode sequences back to  
# something readable.  
reverse_input_char_index = dict(  
    (i, char) for char, i in input_token_index.items())  
reverse_target_char_index = dict(  
    (i, char) for char, i in target_token_index.items())  
  
def decode_sequence(input_seq):  
    # Encode the input as state vectors.  
    states_value = encoder_model.predict(input_seq)  
  
    # Generate empty target sequence of length 1.  
    target_seq = np.zeros((1, 1, num_decoder_tokens))  
    # Populate the first character of target sequence with the start  
    character.  
    target_seq[0, 0, target_token_index['\t']] = 1.  
  
    # Sampling loop for a batch of sequences  
    # (to simplify, here we assume a batch of size 1).  
    stop_condition = False  
    decoded_sentence = ''  
    while not stop_condition:
```

```
        output_tokens, h, c = decoder_model.predict(
            [target_seq] + states_value)

        # Sample a token
        sampled_token_index = np.argmax(output_tokens[0, -1, :])
        sampled_char = reverse_target_char_index[sampled_token_index]
        decoded_sentence += sampled_char

        # Exit condition: either hit max length
        # or find stop character.
        if (sampled_char == '\n' or
            len(decoded_sentence) > max_decoder_seq_length):
            stop_condition = True

        # Update the target sequence (of length 1).
        target_seq = np.zeros((1, 1, num_decoder_tokens))
        target_seq[0, 0, sampled_token_index] = 1.

        # Update states
        states_value = [h, c]

    return decoded_sentence

for seq_index in range(100):
    # Take one sequence (part of the training set)
    # for trying out decoding.
    input_seq = encoder_input_data[seq_index: seq_index + 1]
    decoded_sentence = decode_sequence(input_seq)
    print('-')
    print('Input sentence:', input_texts[seq_index])
    print('Decoded sentence:', decoded_sentence)
```

The preceding code generates the following output:

```
Input sentence: *eyes widen* W-What do you want?  
Decoded sentence: *sings at you and bont*.  
  
-  
Input sentence: I cannot argue with you.  
Decoded sentence: I don't know.  
  
-  
Input sentence: So you admit your idiocy.  
Decoded sentence: I don't know.  
  
-  
Input sentence: You're all and I am nothing.  
Decoded sentence: I don't know.  
  
-  
Input sentence: Yess!!!  
Decoded sentence: *sings at you and bors*.
```

As you can see, the results of our model are quite repetitive. But then we only used 1,000 samples and the responses were generated one character at a time, so this is actually fairly impressive.

If you want better results, rerun the model using more sample data and more epochs.

Here, I have provide some of the more humorous output I've noted from much longer training periods:

```
Input sentence: * Summon Soul Eater eat your soul *  
Decoded sentence: You are a wasting you.
```

```
Input sentence: *hugs*  
Decoded sentence: *stops crying*.
```

```
Input sentence: *catches up with you and pins you against a wall*  
Decoded sentence: *eyes widen* W-What do you want?
```

Input sentence: So you admit your idiocy.
Decoded sentence: I didn't say that. You are being difficult again.

Input sentence: *knocks you out*
Decoded sentence: You are weird.

Input sentence: I'm a potato.
Decoded sentence: Yes you are.

Summary

In this chapter, we took a full tour of the chatbot landscape. It's clear that we're on the cusp of an explosion of these types of applications. The *Conversational UI* revolution is just about to begin. Hopefully, this chapter has inspired you to create your own bot, but if not, we hope you have a much richer understanding of how these applications work and how they'll shape our future.

I'll let the app say the final words:

```
get_response("This is the end, Cleverbot. Say goodbye.")
```

10

Build a Recommendation Engine

Like so many things, it was born of frustration and stiff cocktails. It was a Saturday, and the two young men were once again stuck without a date for the night. As they sat pouring drinks and sharing laments, the two Harvard freshmen began to flesh out an idea. What if, instead of relying on random chance to meet the right girl, they could use a computer algorithm?

The key to matching people up, they felt, would be to create a set of questions that provided the sort of information everyone is really looking for on those first awkward dates. By matching people using these questionnaires, you could eliminate dates that could best be avoided. The process would be super efficient.

The idea was to market their new service to college students in Boston and around the country. And in short order, that's exactly what they did.

Soon after, the digital matchmaking service they built went on to become a huge success. It received national media attention and generated tens of thousands of matches over the course of the next few years. The company was so successful, in fact, it was eventually bought out by a larger company that wanted to use its technology.

If you think I'm talking about **OkCupid**, you would be wrong—and off by about 40 years. The company I'm speaking of did all of this beginning in 1965—a time when computing matches was done using punch cards on an IBM 1401 mainframe. It also took three days just to run the computations.

But oddly enough, there's a connection between OkCupid and its 1965 precursor, Compatibility Research, Inc. The co-founder of Compatibility Research is Jeff Tarr, whose daughter, Jennifer Tarr, is the wife of OkCupid's co-founder Chris Coyne. Small world indeed.

But why is any of this relevant to a chapter on building a recommendation engine? Because it's quite likely that this was in fact the first one. And while most people tend to think of recommendation engines as tools for finding closely related products or music and movies they're likely to appreciate, the original incarnation was to find potential mates. And as a model for thinking about how these systems work, it provides a good frame of reference.

In this chapter, we're going to explore the different varieties of recommendation systems. We'll see how they're implemented commercially and how they work. Finally, we'll implement our own recommendation engine for finding GitHub repositories.

We'll cover the following topics in this chapter:

- Collaborative filtering
- Content-based filtering
- Hybrid systems
- Building a recommendation engine

Collaborative filtering

In early 2012, a story broke about a man who had come into a Target store in Minneapolis to complain about a book of coupons sent to his home. He was in fact quite incensed about these coupons, which had been addressed to his daughter, a high school student at the time. And although it might seem like an odd reaction to a potential money-saving opportunity, learning that the coupons were exclusively for products such as prenatal vitamins, diapers, baby formula, cribs, and so on might change your view.

The manager, upon hearing the complaint, apologized profusely. He felt bad enough, in fact, that he called several days later to follow up and explain how this could have happened. But before the manager was able to even begin his apology, the father began to apologize to the manager. As it turned out, his daughter was in fact pregnant and her shopping habits had given her away.

The algorithm that gave her away was likely based—at least in part—on one type of algorithm used in recommendation engines called **collaborative filtering**.

So, what's collaborative filtering?

Collaborative filtering is based on the idea that, somewhere out there in the world, you have a taste doppelganger—someone who shares the same notions about how good *Star Wars* is and how awful *Love Actually* is.

The idea is that you've rated some set of items in a way that's very similar to the way this other person, this doppelganger, has rated them, but then each of you has rated additional items that the other hasn't. Because you've established that your tastes are similar, recommendations can be generated from the items your doppelganger has rated highly but which you haven't rated and vice versa. It's in a way much like digital matchmaking, but with the outcome being songs or products you would like, rather than actual people.

So, in the case of our pregnant high schooler, when she bought the right combination of unscented lotions, cotton balls, and vitamin supplements, she likely found herself paired up with people who went on to buy cribs and diapers at some point later.

Let's go through an example to see how this works in practice.

We'll start with what's called a **utility matrix**. This is similar to a **term-document matrix** but, instead of terms and documents, we'll be representing products and users.

Here we'll assume that we have customers *A-D* and a set of products that they've rated on a scale from 0 to 5:

Customer	Snarky's Potato Chips	SoSo Smooth Lotion	Duffly Beer	BetterTap Water	XXLargeLivin' Football Jersey	Snowy Cotton Balls	Disposos' Diapers
A	4		5	3	5		
B		4		4		5	
C	2		2		1		
D		5		3		5	4

We've seen previously that, when we want to find similar items, we could use cosine similarity. Let's try that here. We'll find the user most like user *A*. Because we have a sparse vector containing many unrated items, we'll have to input something for those missing values. We'll just go with 0 here. We'll start by comparing user *A* to user *B*:

```
from sklearn.metrics.pairwise import cosine_similarity
cosine_similarity(np.array([4,0,5,3,5,0,0]).reshape(1,-1), \
                  np.array([0,4,0,4,0,5,0]).reshape(1,-1))
```

The previous code results in the following output:

array([[0.18353259]])

As you can see, the two don't have a high similarity rating, which makes sense as they have no ratings in common.

Let's now look at user C compared to user A:

```
cosine_similarity(np.array([4,0,5,3,5,0,0]).reshape(1,-1), \
                  np.array([2,0,2,0,1,0,0]).reshape(1,-1))
```

The previous code results in the following output:

array([[0.88527041]])

Here, we see that they have a high similarity rating (remember 1 is perfect similarity), despite the fact they rated the same products very differently. Why are we getting these results? The problem lies with our choice of using 0 for the unrated products. It's registering as strong (negative) agreement on those unrated products. 0 isn't neutral in this case.

So, how can we fix this?

What we can do instead of just using 0 for the missing values is to re-center each user's ratings so that the mean rating is 0, or neutral. We do this by taking each user rating and subtracting the mean for all ratings of that user. For example, for user A, the mean is 17/4, or 4.25. We then subtract that from every individual rating that user A provided.

Once that's been done, we then continue on to find the mean for every other user and subtract it from each of their ratings until every user has been processed.

This procedure will result in a table like the following. You will notice each user row sums to 0 (ignore the rounding issues here):

Customers	Snarky's Potato Chips	SoSo Smooth Lotion	Duffly Beer	BetterTap Water	XXLargeLivin' Football Jersey	Snowy Cotton Balls	Disposos' Diapers
A	-.25		.75	-1.25	.75		
B		-.33		-.33		.66	
C	.33		.33		-.66		
D		.75		-1.25		.75	-.25

Let's now try our cosine similarity on our newly centered data. We'll do user A compared to user B and C again.

First, let's compare user *A* to user *B*:

```
cosine_similarity(np.array([- .25, 0, .75, -1.25, .75, 0, 0]) \
    .reshape(1,-1), \
    np.array([0, -.33, 0, -.33, 0, .66, 0]) \
    .reshape(1,-1))
```

The preceding code results in the following output:

```
array([[ 0.30772873]])
```

Now let's try between users *A* and *C*:

```
cosine_similarity(np.array([- .25, 0, .75, -1.25, .75, 0, 0]) \
    .reshape(1,-1), \
    np.array([.33, 0, .33, 0, -.66, 0, 0]) \
    .reshape(1,-1))
```

The preceding code results in the following output:

```
array([[-0.24618298]])
```

What we can see is that the similarity between *A* and *B* increased slightly, while the similarity between *A* and *C* decreased dramatically. This is exactly as we would hope.

This centering process, besides helping us deal with missing values, also has the side benefit of helping us to deal with difficult or easy raters since now everyone is centered around a mean of 0. This formula, you may notice, is equivalent to the Pearson correlation coefficient and, just like with that coefficient, the values fall between -1 and 1 .

Predicting the rating for the product

Let's now take this framework and use it to predict the rating for a product. We'll limit our example to three users, person *X*, person *Y*, and person *Z*. We'll predict the rating of a product that person *X* hasn't rated, but that persons *Y* and *Z*, who are very similar to *X*, have rated.

We'll start with our base ratings for each user, as shown in the following table:

Customers	Snarky's Potato Chips	SoSo Smooth Lotion	Duffly Beer	BetterTap Water	XXLargeLivin' Football Jersey	Snowy Cotton Balls	Disposos' Diapers
X		4		3		4	
Y		3.5		2.5		4	4
Z		4		3.5		4.5	4.5

Next, we'll center the ratings:

Customers	Snarky's Potato Chips	SoSo Smooth Lotion	Duffly Beer	BetterTap Water	XXLargeLivin' Football Jersey	Snowy Cotton Balls	Disposos' Diapers
X		.33		-.66		.33	?
Y		0		-1		.5	.5
Z		-.125		-.625		.375	.375

Now, we'd like to know what rating user X might be likely to give **Disposos' Diapers**. Using the ratings from user Y and user Z, we can calculate this by taking the weighted average according to their centered cosine similarity.

Let's first get that figure:

```
user_x = [0, .33, 0, -.66, 0, 33, 0]
user_y = [0, 0, 0, -1, 0, .5, .5]

cosine_similarity(np.array(user_x).reshape(1,-1), \
                  np.array(user_y).reshape(1,-1))
```

The preceding code results in the following output:

```
array([[ 0.42447212]])
```

Now, let's get that figure for user Z:

```
user_x = [0, .33, 0, -.66, 0, 33, 0]
user_z = [0, -.125, 0, -.625, 0, .375, .375]

cosine_similarity(np.array(user_x).reshape(1,-1), \
                  np.array(user_z).reshape(1,-1))
```

The preceding code results in the following output:

```
array([[ 0.46571861]])
```

So, now we have a figure for the similarity between user X and user Y (0.42447212) and user Z (0.46571861).

Putting it all together, we weight each users rating by their similarity to X, and then divide by the total similarity, as follows:

$$(0.42447212 * (4) + 0.46571861 * (4.5)) / (0.42447212 + 0.46571861) = 4.26$$

And we can see that the expected rating of user X for **Disposos' Diapers** is 4.26. (Better send a coupon!)

Now, so far, we've looked only at user-to-user collaborative filtering, but there's another method we can use. In practice, this method outperforms user-to-user filtering; it's called **item-to-item filtering**. Here's how the method works: rather than match each user up with other similar users based on their past ratings, each rated item is compared against all other items to find the most similar ones, again using centered cosine similarity.

Let's take a look at how this would work.

Again we have a utility matrix; this time, we'll look at users' ratings of songs. The users are along the columns and the songs are along the rows, shown as follows:

Entity	U1	U2	U3	U4	U5
S1	2		4		5
S2		3		3	
S3	1		5		4
S4		4	4	4	
S5	3				5

Now, suppose we would like to know the rating that user 3 will assign to song 5. Instead of looking for similar users, we'll look for songs that are similar based upon how they were rated across the users.

Let's see an example.

First, we start by centering each song row, and calculating the cosine similarity for each versus our target row, which is S_5 , shown as follows:

Entity	U1	U2	U3	U4	U5	CntrdCoSim
S_1	-1.66		.33		1.33	.98
S_2		0		0		0
S_3	-2.33		1.66		.66	.72
S_4		0	0	0		0
S_5	-1		?		1	1

You can see the far right column has been calculated with the centered cosine similarity for each row versus row S_5 .

We next need to select a number, k , that's the number of the nearest neighbors we'll use to rate songs for user 3. We use $k = 2$ in our simple example.

You can see that song S_1 and song S_3 are the most similar, so we'll use those two along with the ratings user 3 had for S_1 and S_3 (4 and 5, respectively).

Let's now calculate the rating:

$$(.98 * (4) + .72 * (5)) / (.98 + .72) = 4.42$$

So, based on this item-to-item collaborative filtering, we can see user 3 is likely to rate song S_5 very highly at 4.42 from our calculations.

Earlier, I said that user-to-user filtering is less effective than item-to-item filtering. Why might that be?

There's a good chance you have friends who really enjoy some of things that you enjoy as well, but then each of you has other areas of interest that the other has absolutely no interest in.

For example, perhaps you both love *Game of Thrones*, but your friend also loves Norwegian death metal. You, however, would rather be dead than listen to Norwegian death metal. If you're similar in many ways—excluding the death metal—with user-to-user recommendations, you're still going to see a lot of recommendations for bands with names that include words such as *flaming*, *axe*, *skull*, and *bludgeon*. With item-to-item filtering, most likely, you would be spared those suggestions.

So far, we've looked at users and items as a single entity when making comparisons, but now let's move on to look at another method that decomposes our users and items into what might be called **feature baskets**.

Content-based filtering

As a musician himself, Tim Westergren had spent years on the road listening to other talented musicians, wondering why they could never get ahead. Their music was good—just as good as anything you might hear on the radio—and yet, somehow, they just never caught their big break. He imagined it must be because their music just never got in front of enough of the right people.

Tim eventually quit his job as a musician and took another job as a composer for movie scores. It was there that he began to think of each piece of music as having a distinct structure that could be decomposed into constituent parts—a form of musical DNA.

After giving it some thought, he began to consider creating a company around this idea of building a **musical genome**. He ran the concept by one of his friends, who had previously created and sold a company. The friend loved Tim's idea. So much so, in fact, that he began helping him to write a business plan and gather the initial funding round for the project. It was a go.

Over the next several years, they employed a small army of musicians who meticulously codified almost 400 distinct musical features for over a million pieces of music. Each feature was rated on a 0 to 5 point scale by hand (or maybe by ear is a better way to say it). Each three- or four-minute song took nearly a half hour to classify.

The features included things such as how gravelly the lead singers' voice was or how many beats per minute the tempo was. It took nearly a year for their first prototype to be completed. Built entirely in Excel using a VBA macro, it took nearly four minutes just to return a single recommendation. But in the end, it worked and it worked well.

That company is now known as Pandora music, and chances are you've either heard of it or used its products as it has millions of daily users around the world. It's without a doubt a triumphant example of content-based filtering.

Rather than treat each song as a single indivisible unit, as in content-based filtering, the songs become feature vectors that can be compared using our friend cosine similarity.

Another benefit is that not only are the songs subject to being decomposed into feature vectors, but the listeners can be as well. Each listener's taste profile becomes a vector in this space so that measurements can be made between their taste profiles and the songs themselves.

For Tim Westergren, this was the magic, because rather than rely on the popularity of the music like so many recommendations are, the recommendations from this system were made based upon the inherent structural similarity. Maybe you've never heard of song X, but if you like song Y, then you should like song X because it's *genetically* almost identical. That's content-based filtering.

Hybrid systems

We've now looked at the two primary forms of recommender systems, but you should know that, in any large-scale production environment, you're likely to see recommendations that leverage both of these. This is known as a **hybrid system**, and the reason hybrid systems are preferred is that they help eliminate the drawbacks that can be present when using either system alone. The two systems together create a more robust solution.

Let's examine the pros and cons of each type.

Collaborative filtering

The pros of collaborative filtering are as follows:

- There's no need to hand-craft features

The cons are as follows:

- Doesn't work well without a large number of items and users
- Sparsity when the number of items far exceeds the number that could be purchased

Content-based filtering

The pros of content-based filtering are as follows:

- It doesn't require a large number of users

The cons are as follows:

- Defining the right features can be a challenge
- Lack of serendipity

As you can see, content-based filtering is a better choice when you haven't built up a large user base but, as you grow, adding on collaborative filtering can help introduce more serendipity into the recommendations.

Now that you're familiar with the types and inner-workings of recommendation engines, let's begin constructing one of our own.

Building a recommendation engine

One thing I love to stumble upon is a really useful GitHub repository. You can find repositories that contain everything from hand-curated tutorials on machine learning to libraries that will save you dozens of lines of code when using **Elasticsearch**. The trouble is, finding these libraries is far more difficult than it should be. Fortunately, we now have the knowledge to leverage the GitHub API in a way that will help us to find these code gems.

We're going to be using the GitHub API to create a recommendation engine based on collaborative filtering. The plan is to get all of the repositories I've starred over time and to then get all of the creators of those repositories to find what repositories they've starred. Once that's done, we'll find which users are most similar to me (or you, if you're running this for your own repository, which I suggest). Once we have the most similar users, we can use the repositories they've starred and that I haven't to generate a set of recommendations.

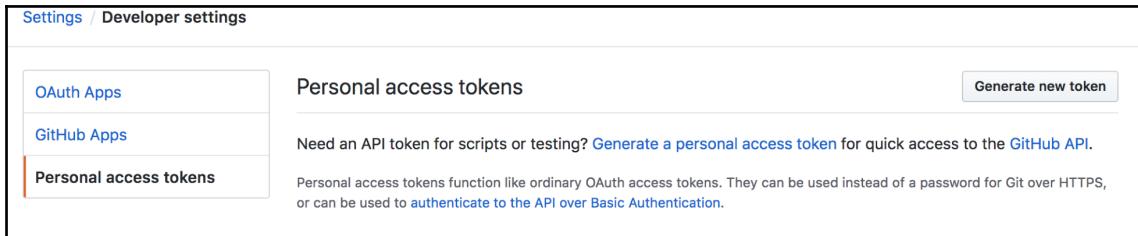
Let's get started:

1. We'll import the libraries we'll need:

```
import pandas as pd
import numpy as np
import requests
import json
```

2. You'll need to have opened an account with GitHub and to have starred a number of repositories for this to work for your GitHub handle, but you won't actually need to sign up for the developer program. You can get an authorization token from your profile, which will allow you to use the API. You can also get it to work with this code, but the limits are too restrictive to make it usable for our example.

3. To create a token for use with the API, go to the following URL at <https://github.com/settings/tokens>. There, you will see a button in the upper-right corner like the following:



4. You'll need to click on that **Generate new token** button. Once you've done that, you need to select the permissions, I chose just **public_repo**. Then, finally, copy the token it gives you for use in the following code. Be sure to enclose both in quotes:

```
myun = YOUR_GITHUB_HANDLE  
mypw = YOUR_PERSONAL_TOKEN
```

5. We'll create the function that will pull the names of every repository you've starred:

```
my_starred_repos = []  
def get_starred_by_me():  
    resp_list = []  
    last_resp = ''  
    first_url_to_get = 'https://api.github.com/user/starred'  
    first_url_resp = requests.get(first_url_to_get,  
        auth=(myun,mypw))  
    last_resp = first_url_resp  
    resp_list.append(json.loads(first_url_resp.text))  
    while last_resp.links.get('next'):  
        next_url_to_get = last_resp.links['next']['url']  
        next_url_resp = requests.get(next_url_to_get,  
            auth=(myun,mypw))  
        last_resp = next_url_resp  
        resp_list.append(json.loads(next_url_resp.text))  
    for i in resp_list:  
        for j in i:  
            msr = j['html_url']  
            my_starred_repos.append(msr)
```

There's a lot going on in there, but, essentially, we're querying the API to get our own starred repositories. GitHub uses pagination rather than return everything in one call. Because of this, we'll need to check the `.links` that are returned from each response. As long as there is a next link to call, we'll continue to do so.

6. We just need to call that function we created:

```
get_starred_by_me()
```

7. Then, we can see the full list of starred repositories:

```
my_starred_repos
```

The preceding code will result in output similar to the following:

```
['https://github.com/alash3al/sqler',
 'https://github.com/ecthros/uncaptcha2',
 'https://github.com/yyx990803/build-your-own-mint',
 'https://github.com/trekhleb/homemade-machine-learning',
 'https://github.com/soumith/dcgan.torch',
 'https://github.com/minimaxir/textgenrnn',
 'https://github.com/kingkool168/generate-ssl-certs-for-local-development',
 'https://github.com/enochtangg/quick-SQL-cheatsheet',
 'https://github.com/remicnrd/ml_cheatsheet',
 'https://github.com/mwouts/jupytext',
 'https://github.com/seandavi/SRAdb',
 'https://github.com/chubin/cheat.sh',
 'https://github.com/madchops1/Dutchess.ai',
 'https://github.com/alopresto/slides',
 'https://github.com/Azure/mmlspark',
 'https://github.com/zotroneneis/machine_learning_basics',
 'https://github.com/ipython-contrib/jupyter_contrib_nbextensions',
 'https://github.com/Esri/spatial-framework-for-hadoop',
```

8. We need to parse out the user names for each of the libraries we starred so that we can retrieve the libraries they starred:

```
my_starred_users = []
for ln in my_starred_repos:
    right_split = ln.split('.com/')[1]
    starred_usr = right_split.split('/')[0]
    my_starred_users.append(starred_usr)

my_starred_users
```

This will result in output similar to the following:

```
[ 'alash3al',
  'ecthros',
  'yx990803',
  'trekhleb',
  'soumith',
  'minimaxir',
  'kingkool68',
  'enochtangg',
  'remicnrd',
  'mwouts',
  'seandavi',
  'chubin',
  'madchops1',
  'alopresto',
  'Azure',
  'zottroneneis',
  'ipython-contrib',
  'Esri',
```

9. Now that we have the handles for all of the users we starred, we'll need to retrieve all of the repositories they starred. The following function will do just that:

```
starred_repos = {k:[] for k in set(my_starred_users)}
def get_starred_by_user(user_name):
    starred_resp_list = []
    last_resp = ''
    first_url_to_get = 'https://api.github.com/users/' + user_name
    +'/_starred'
    first_url_resp = requests.get(first_url_to_get,
    auth=(myun,mypw))
    last_resp = first_url_resp
    starred_resp_list.append(json.loads(first_url_resp.text))
    while last_resp.links.get('next'):
        next_url_to_get = last_resp.links['next']['url']
        next_url_resp = requests.get(next_url_to_get,
        auth=(myun,mypw))
        last_resp = next_url_resp
        starred_resp_list.append(json.loads(next_url_resp.text))
    for i in starred_resp_list:
        for j in i:
            sr = j['html_url']
            starred_repos.get(user_name).append(sr)
```

This function works in nearly the same way as the function we called earlier, but calls a different endpoint. It'll add their starred repositories to a dict we'll use later.

10. Let's call it now. It may take a few minutes to run, depending on the number of repositories each user has starred. I actually had one that starred over 4,000 repositories:

```
for usr in list(set(my_starred_users)):  
    print(usr)  
    try:  
        get_starred_by_user(usr)  
    except:  
        print('failed for user', usr)
```

The preceding code will result in output similar to the following:

```
Charging-Alot  
amontalenti  
kingkool68  
jorisvandenbossche  
alexisgreene  
tweepy  
cchi  
kilimchoi  
prompt-toolkit  
jostmey  
matplotlib  
d3  
rhiever  
elastic  
mahmoudparsian  
ecthros  
linkedin  
mit-nlp
```

Notice that I turned the list of starred users into a set before I called it. I noticed some duplication that resulted from starring multiple repositories under one user handle, so it makes sense to follow these steps to reduce extra calls:

1. We now need to build a feature set that includes all of the starred repositories of everyone we have starred:

```
repo_vocab = [item for sl in list(starred_repos.values()) for item  
in sl]
```

2. We'll convert that into a set to remove duplicates that may be present from multiple users starring the same repositories:

```
repo_set = list(set(repo_vocab))
```

3. Let's see how many that produces:

```
len(repo_vocab)
```

The preceding code should result in output similar to the following:

27098

I had starred 170 repositories, and together the users of those repositories starred over 27,000 unique repositories. You can imagine if we went one degree further out how many we might see.

Now that we have the full feature set, or repository vocabulary, we need to run every user to create a binary vector that contains a 1 for every repository they've starred and a 0 for every repository they haven't:

```
all_usr_vector = []
for k,v in starred_repos.items():
    usr_vector = []
    for url in repo_set:
        if url in v:
            usr_vector.extend([1])
        else:
            usr_vector.extend([0])
    all_usr_vector.append(usr_vector)
```

What we just did was check for every user whether they had starred every repository in our repository vocabulary. If they did, they received a 1, if they didn't, they received a 0.

At this point, we have a 27,098 item binary vector for each user—all 170 of them. Let's now put this into a DataFrame. The row index will be the user handles we starred, and the columns will be the repository vocabulary:

```
df = pd.DataFrame(all_usr_vector, columns=repo_set,
index=starred_repos.keys())
df
```

The preceding code will generate output similar to the following:

	https://github.com/sebpiq/rhizome	https://github.com/googleapis/google-cloud-python	https://github.com/taichino/croniter
Charging-Alot	0	0	0
amontalenti	0	0	0
kingkool68	0	0	0
jorisvandenbossche	0	0	0
alexgreene	0	0	0
tweepy	0	0	0
cchi	0	0	0
kilimchoi	0	0	0
prompt-toolkit	0	0	0
jostmey	0	0	0

Next, in order to compare ourselves to the other users, we need to add our own row to this frame. Here, I add my user handle, but you should add your own:

```
my_repo_comp = []
for i in df.columns:
    if i in my_starred_repos:
        my_repo_comp.append(1)
    else:
        my_repo_comp.append(0)

mrc = pd.Series(my_repo_comp).to_frame('acombs').T

mrc
```

The preceding code will generate output similar to the following:

0	1	2	3	4	5	6	7	8	9	...	27088	27089	27090	27091	27092	27093	27094	27095	27096	27097
acombs	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0

We now need add the appropriate column names and to concatenate this to our other DataFrame:

```
mrc.columns = df.columns  
  
fdf = pd.concat([df, mrc])  
  
fdf
```

The preceding code will result in output similar to the following:

JohnSnowLabs	0	0	0
ipython-contrib	0	0	0
sandialabs	0	0	0
sqlitebrowser	0	0	0
jundongl	0	0	0
learnbyexample	0	0	0
altair-viz	0	0	0
hortonworks-gallery	0	0	0
Azure	0	0	0
acomb	0	0	0

You can see in the previous screenshot that I've been added into the DataFrame.

From here, we just need to calculate the similarity between ourselves and the other users we've starred. We'll do that now using the `pearsonr` function which we'll need to import from SciPy:

```
from scipy.stats import pearsonr  
  
sim_score = {}  
for i in range(len(fdf)):  
    ss = pearsonr(fdf.iloc[-1,:], fdf.iloc[i,:])  
    sim_score.update({i: ss[0]})  
  
sf = pd.Series(sim_score).to_frame('similarity')  
sf
```

The preceding code will generate output similar to the following:

similarity	
0	NaN
1	0.045831
2	-0.002590
3	0.089009
4	0.018741
5	NaN
6	0.154673
7	0.010973
8	NaN
9	0.024107

What we've just done is compare our vector, the last one in the `DataFrame`, to every other user's vector to generate a centered cosine similarity (Pearson correlation coefficient). Some values are by necessity `NaN`, as they've starred no repositories, and hence result in division by zero in the calculation:

1. Let's now sort these values to return the index of the users who are most similar:

```
sf.sort_values('similarity', ascending=False)
```

The preceding code will result in output similar to the following:

similarity	
159	1.000000
6	0.154673
42	0.133273
116	0.119926
118	0.113706
103	0.105356
37	0.094806
3	0.089009
12	0.086880
145	0.085535

So there we have it, those are the most similar users, and hence the ones that we can use to recommend repositories we might enjoy. Let's take a look at these users and what they have starred that we might like.

2. You can ignore that first user with a perfect similarity score; that's our own repository. Going down the list, the three nearest matches are user 6, user 42, and user 116. Let's look at each:

```
fdf.index[6]
```

The preceding code will result in output similar to the following:

```
'cchi'
```

3. Let's take a look at who this is and their repository. From <https://github.com/cchi>, I can see who the repository belongs to the following user:



This is actually Charles Chi, a former colleague of mine from Bloomberg, so this is no surprise. Let's see what he has starred:

1. There are a couple of ways to do this; we can either use our code, or just click under their picture on stars. Let's do both for this one, just to compare and make sure everything matches up. First, let's do it via code:

```
fdf.iloc[6, :][fdf.iloc[6, :]==1]
```

This results in the following output:

https://github.com/hangtwenty/dive-into-machine-learning	1
https://github.com/doximity/vital	1
https://github.com/numamenta/nupic	1
https://github.com/chaconnewu/free-data-science-books	1
https://github.com/johnmyleswhite/ML_for_Hackers	1
https://github.com/jeroenjanssens/data-science-at-the-command-line	1
https://github.com/tensorflow/skflow	1
https://github.com/nborwankar/LearnDataScience	1
https://github.com/dariusk/corpora	1
https://github.com/shagunsodhani/papers-I-read	1
https://github.com/clips/pattern	1
https://github.com/rasbt/pattern_classification	1
https://github.com/apache/predictionio	1
https://github.com/graceavery/Eigenstyle	1
https://github.com/prakhar1989/awesome-courses	1
https://github.com/DataScienceSpecialization/courses	1
https://github.com/scikit-learn/scikit-learn	1

2. We see 30 starred repositories. Let's compare those to the ones from GitHub's site:

The screenshot shows a GitHub user profile for 'Charles Chi' (username: cchi). The profile includes a custom profile picture featuring a gear and the letters 'Chi'. Key statistics at the top are: Overview, Repositories 17, Stars 30 (highlighted in orange), Followers 6, and Following 20. Below this, the 'REPOSITORIES' section lists three starred repositories:

- rushter / data-science-blogs**: A curated list of data science blogs. Starred by Charles Chi. Last updated 16 days ago.
- DataScienceSpecialization / courses**: Course materials for the Data Science Specialization. Starred by Charles Chi. Last updated on Dec 6, 2018.
- jeroenjanssens / data-science-at-the-command-line**: Data Science at the Command Line. Starred by Charles Chi. Last updated 23 days ago.

3. Here we can see they're identical, and you'll notice you can ID the repositories that we've both starred: they are the ones labelled **Unstar**.
4. Unfortunately, with just 30 starred repositories, there aren't a lot of repositories to generate recommendations.
5. The next user in terms of similarity is 42, Artem Golubin:

```
fdf.index[42]
```

The preceding code results in the following output:

```
'rushter'
```

And his GitHub profile below:



Here we see the repositories he has starred:

The screenshot shows a GitHub user profile for 'Artem Golubin'. On the left is a purple abstract profile picture. Below it, the user's name 'Artem Golubin' and handle 'rushter' are displayed, with a 'Follow' button. To the right, there are tabs for 'Overview', 'Repositories 9', 'Stars 530' (which is underlined in orange), 'Followers 570', and 'Following 20'. Under the 'REPOSITORIES' heading, two starred repositories are listed: 'zziz / pwc' (description: 'Papers with code. Sorted by stars. Updated weekly.', stats: 8,002 stars, 885 forks, updated 5 days ago) and 'eleweek / inside_python_dict' (description: 'An explorable explanation of python dictionaries', stats: 74 stars, 5 forks, updated an hour ago). Each repository entry includes a 'Star' button.

6. Artem has starred over 500 repositories, so there are definitely some recommendations to be found there.
7. And finally, let's look at the third most similar user:

```
fdf.index[116]
```

This results in the following output:

```
'justmarkham'
```

This user, Kevin Markham, has starred around 60 repositories:

The screenshot shows a GitHub user profile for 'Kevin Markham'. It features a portrait photo of a smiling man with dark hair. Below the photo, the name 'Kevin Markham' and handle 'justmarkham' are shown, along with a 'Follow' button.

We can see the starred repositories in the following image:

The screenshot shows a GitHub user profile for Kevin Markham (justmarkham). The profile includes a photo of Kevin smiling, his name, and his GitHub handle. It also shows a 'Follow' button and a 'Block or report user' link. At the top, there are tabs for Overview, Repositories (36), Stars (59), Followers (3.6k), and Following (0). The 'Stars' tab is currently selected. Below the tabs is a 'REPOSITORIES' section with a 'Filters' dropdown. Two repositories are listed: 'janelleshane / halloween-costume-dataset' and 'chris1610 / pbpython'. Each repository entry includes the repository name, a 'Star' button, a brief description, the number of stars, the number of forks, and the last update date.

Repository	Stars	Forks	Last Updated
janelleshane / halloween-costume-dataset	29	1	Oct 26, 2017
chris1610 / pbpython	814	361	Nov 18, 2018

This is definitely fertile ground for generating recommendations. Let's now do just that; let's use the links from these three to produce some recommendations:

1. We need to gather the links to the repositories they've starred and that I haven't. We'll create a DataFrame that has the repositories I've starred as well as the three most similar users to me:

```
all_recs = fdf.iloc[[6, 42, 116, 159], :]  
all_recs.T
```

The preceding code will produce the following output:

	cchi	rushter	justmarkham	acombs
https://github.com/sebpiq/rhizome	0	0	0	0
https://github.com/googleapis/google-cloud-python	0	0	0	0
https://github.com/taichino/croniter	0	0	0	0
https://github.com/PhrozenByte/rmtrash	0	0	0	0
https://github.com/LeCoupa/awesome-cheatsheets	0	0	0	0
https://github.com/chrischoy/3D-R2N2	0	0	0	0
https://github.com/CISOfy/lynis	0	0	0	0
https://github.com/schneems/Octave	0	0	0	0
https://github.com/quantmind/d3-view	0	0	0	0
https://github.com/uwescience/shablona	0	0	0	0
https://github.com/mickelindahl/progress-barjs	0	0	0	0

2. Don't worry if it looks like it's all zeros; this is a sparse matrix so most will be 0. Let's see whether there are any repositories we've all starred:

```
all_recs[(all_recs==1).all(axis=1)]
```

The preceding code will produce the following output:

	cchi	rushter	justmarkham	acombs
https://github.com/hangtwenty/dive-into-machine-learning	1.0	1.0	1.0	1.0
https://github.com/scikit-learn/scikit-learn	1.0	1.0	1.0	1.0
https://github.com/josephmisiti/awesome-machine-learning	1.0	1.0	1.0	1.0

3. As you can see, we all seem to love scikit-learn and machine learning repositories—no surprise there. Let's see what they might have all starred that I missed. We'll start by creating a frame that excludes me, and then we'll query it for commonly starred repositories:

```
str_recs_tmp = all_recs[all_recs[myun]==0].copy()
str_recs = str_recs_tmp.iloc[:, :-1].copy()
str_recs[(str_recs==1).all(axis=1)]
```

The preceding code produces the following output:

cchi	rushter	justmarkham

4. Okay, so it looks like I haven't been missing anything super obvious. Let's see if there any repositories that at least two out of three users starred. To find this, we'll just sum across the rows:

```
str_recs.sum(axis=1).to_frame('total').sort_values(by='total',
ascending=False)
```

The preceding code will result in output similar to the following:

	total
https://github.com/kjw0612/awesome-rnn	2.0
https://github.com/airbnb/aerosolve	2.0
https://github.com/apache/predictionio	2.0
https://github.com/seatgeek/fuzzywuzzy	2.0
https://github.com/blue-yonder/tsfresh	2.0
https://github.com/DataScienceSpecialization/courses	2.0
https://github.com/hal3/vwnlp	2.0
https://github.com/numamenta/nupic	2.0
https://github.com/rasbt/pattern_classification	2.0
https://github.com/EpistasisLab/tpot	2.0
https://github.com/ChristosChristofidis/awesome-deep-learning	2.0
https://github.com/openai/gym	2.0
https://github.com/ujjwalkarn/Machine-Learning-Tutorials	2.0
https://github.com/TomAugspurger/postlearn	1.0
https://github.com/mitsuhiko/python-regex-scanner	1.0
https://github.com/facebookarchive/caffe2	1.0

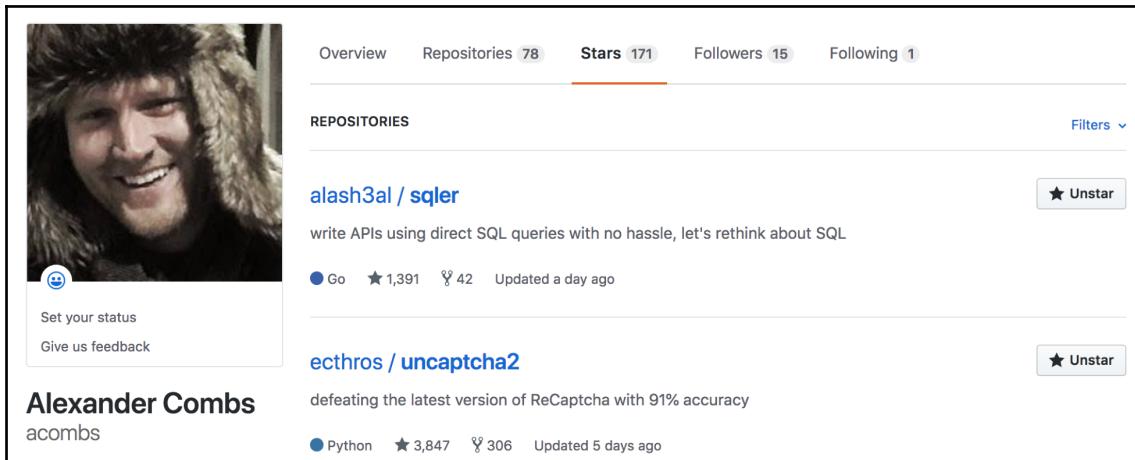
This looks promising. There are lots of good ML and AI repositories, and I'm honestly ashamed I never starred **fuzzywuzzy** as I use that quite frequently.

At this point, I have to say I'm impressed with the results. These are definitely repositories that interest me, and I'll be checking them out.

So far, we've generated the recommendations using collaborative filtering and done some light additional filtering using aggregation. If we wanted to go further, we could order the recommendation based upon the total number of stars they received. This could be achieved by making another call to the GitHub API. There's an endpoint that provides this information.

Another thing we could do to improve the results is to add in a layer of content-based filtering. This is the hybridization step we discussed earlier. We would need to create a set of features from our own repository that was indicative of the types of things we would be interested in. One way to do this would be to create a feature set by tokenizing the names of the repositories we have starred along with their descriptions.

Here's a look at my starred repositories:



The screenshot shows a GitHub profile page for a user named Alexander Combs (username: acombs). The profile picture is a smiling man with long hair and a beard. The top navigation bar includes links for Overview, Repositories (78), Stars (171, highlighted in orange), Followers (15), and Following (1). Below the navigation is a section titled "REPOSITORIES" with a "Filters" dropdown. Two repositories are listed: "alash3al / sqlier" and "ecthros / uncaptcha2".

- alash3al / sqlier**: write APIs using direct SQL queries with no hassle, let's rethink about SQL. Language: Go, Stars: 1,391, Forks: 42, Updated: a day ago.
- ecthros / uncaptcha2**: defeating the latest version of ReCaptcha with 91% accuracy. Language: Python, Stars: 3,847, Forks: 306, Updated: 5 days ago.

As you might imagine, this would generate a set of word features that we could use to vet the repositories of those we found using collaborative filtering. This would include a lot of words such as *Python*, *Machine Learning*, and *Data Science*. This would ensure that users who are less similar to ourselves are still providing recommendations that are based on our own interests. It would also reduce the serendipity of the recommendations, which is something to consider. Perhaps there's something unlike anything I have currently that I would love to see. It's certainly a possibility.

What would that content-based filtering step look like in terms of a DataFrame? The columns would be word features (n-grams) and the rows would be the repositories generated from our collaborative filtering step. We would just run the similarity process once again using our own repository for comparison.

Summary

In this chapter, we learned about recommendation engines. We learned about the two primary types of systems in use today: collaborative filtering and content-based filtering. We also learned how they can be used together to form a hybrid system. We also discussed the pros and cons of each type of system. And finally, we learned step by step how to build a recommendation engine from scratch using the GitHub API.

I hope you build your own recommendation engine using the guidance in this chapter, and I hope you find resources that are useful to you. I know I've found a number of things I will certainly be using. Best of luck to you on your journey!

11

What's Next?

So far, we've used **machine learning (ML)** to implement various tasks. There are many advancements in the field of ML, and with each passing day, the areas in which it is applied are increasing too.

In this chapter, we'll summarize the projects that we executed in the previous chapters.

Summary of the projects

Let's start with Chapter 1, *The Python Machine Learning Ecosystem*.

In the first chapter, we began with an overview of ML with Python. We started with the ML workflow, which included acquisition, inspection, preparation, modeling evaluation, and deployment. Then we studied the various Python libraries and functions that are needed for each step of the workflow. Lastly, we set up our ML environment to execute the projects.

Chapter 2, *Building an App to Find Underpriced Apartments*, as the name says, was based on building an app to find underpriced apartments. Initially, we listed our data to find the source of the apartments in the required location. Then, we inspected the data, and after preparing and visualizing the data, we performed regression modeling. Linear regression is a type of supervised ML. Supervised, in this context, simply means we provide the output values for our training set.

Then, we spent the remainder of our time exploring the options as per our choice. We created an application that made finding the right apartment just a little bit easier.

In Chapter 3, *Building an App to Find Cheap Airfare*, we built a similar app as in Chapter 2, *Building an App to Find Underpriced Apartments*, but to find cheap airfare. We started by sourcing airfare prices on the web. We used one of the trending techniques, web scraping, to retrieve the data of the airplane fares. To parse the DOM for our Google page, we used the `BeautifulSoup` library. Then, we used anomaly detection techniques to identify outlier fares. In doing this, cheaper airfare can be found, and we'll receive real-time text alerts using IFTTT.

In Chapter 4, *Forecasting the IPO Market Using Logistic Regression*, we looked at how the IPO market works. First, we discussed what an **Initial Public Offering (IPO)** is, and what the research tells us about this market. After that, we discussed a number of strategies that we can apply to predict the IPO market. It involved data cleansing and feature-engineering. Then, we implemented binary classification of the data using logistic regression to analyze the data. Then, we evaluated the final model, which was obtained as the output.

We also understood that the features that have an impact on our model include the feature importance that comes out of a random forest classifier. This more accurately reflects the true impact of a given feature.

Chapter 5, *Create a Custom Newsfeed*, was mostly for avid news readers who are interested in knowing what's going on around the globe. By creating a custom newsfeed, you can decide what news updates you get on your devices. In this chapter, you learned how to build a system that understands your taste in news, and will send you a tailored newsletter each day. We started by creating a supervised training set with the Pocket app, and then leveraged the Pocket API to retrieve the stories. We used the Embedly API to extract story bodies.

Then, we studied the basics of **natural language processing (NLP)** and **Support Vector Machines (SVMs)**. We integrated **If This Then That (IFTTT)** with RSS feeds and Google sheets so that we could stay up to date with notifications, emails, and more. Lastly, we set up a daily personal newsletter. We used the Webhooks channel to send a `POST` request.

The script runs every four hours, pulls down the news stories from Google Sheets, runs the stories through the model, generates an email by sending a `POST` request to IFTTT for the stories that are predicted to be of interest, and then, finally, it will clear out the stories in the spreadsheet so only new stories get sent in the next email. And that's how we get our very own personalized newsfeed.

In Chapter 6, *Predicate Whether Your Content Will Go Viral*, we examined some of the most-shared content and attempted to find the common elements that differentiate this content from the content people were less inclined to share. The chapter started by providing an understanding of what exactly virality means. We also looked at what research tells us about virality.

Then, as we did it in the rest of the chapters we will be sourcing the shared counts and content. We used a dataset that was collected from a now-defunct website called `ruzzit.com`. This site, when it was active, tracked the most-shared content over time, which was exactly what we needed for this project. Then we explored the features of shareability, which included exploring image data, clustering, exploring the headlines, and exploring the story's content.

The last, but most important, part was building the predictive content-scoring model. We used an algorithm called **random forest regression**. We built the model with zero errors in it. Then, we evaluated the model and added some features to enhance it.

In Chapter 7, *Use Machine Learning to Forecast the Stock Market*, we learned how to build and test a trading strategy. We also learned how *not* to do it. There are countless pitfalls to avoid when trying to devise your own system, and it's nearly an impossible task, but it can be a lot of fun, and sometimes it can even be profitable. That said, don't do dumb things, such as risking money you can't afford to lose.

When you're ready to risk you money, you might as well learn some tricks and tips to avoid losing much of it. Who likes to lose in life—be it for money or a game?

We mostly concentrated our attention on stocks and the stock market. Initially, we analyzed types of markets and then the researched on the stock market. It's always better to have some prior knowledge before risking anything. We began developing our strategy by focusing on the technical aspects. We went through the S&P 500 over the last few years and used pandas to import our data. That gave us access to several sources of stock data, including Yahoo! and Google.

Then we built the regression model. We started with a very basic model using only the stock's prior closing values to predict the next day's close, and built it using a support vector regression. Lastly, we evaluated the performance of our model and the trades that were carried out.

Long before Siri was released with the iPhone 4S, we had chatbots that were used widely across multiple applications. In Chapter 9, *Building a Chatbot*, we learned about the Turing Test and its origins. Then we looked at a program called ELIZA. If ELIZA was an early example of chatbots, what have we seen since then? In recent years, there has been an explosion of new chatbots—the most notable of these is Cleverbot.

Then, we looked at the interesting part: designing these chatbots.

But what about more advanced bots? How are they built?

Surprisingly, most chatbots you're likely to encounter don't use ML; they're what's known as retrieval-based models. This means responses are predefined according to the question and the context. The most common architecture for these bots is something called **Artificial Intelligence Markup Language (AIML)**. AIML is an XML-based schema for representing how the bot should interact given the user's input. It's really just a more advanced version of how ELIZA works.

Lastly, we did sequence-to-sequence modeling for chatbots. This is frequently used in machine translation and question-answering applications as it allows us to map an input sequence of any length to an output sequence of any length.

In Chapter 8, *Classifying Images with Convolutional Neural Networks*, we looked at building a **Convolutional Neural Network (CNN)** to classify images in the Zalando Research dataset using Keras.

We started by extracting the image's features. Then, using CNNs, we understood the network topology, the various convolutional layers and filters, and what max pooling layers are.

Try building deeper models or grid searching over the many hyperparameters we used in our models. Assess your classifier's performance as you would with any other model—try building a confusion matrix to understand what classes we predicted well and what classes we weren't as strong in!

In Chapter 10, *Build a Recommendation Engine*, we explored different varieties of recommendation systems. We saw how they're implemented commercially and how they work. Then we implemented our own recommendation engine for finding GitHub repositories.

We started with collaborative filtering. Collaborative filtering is based on the idea that, somewhere out there in the world, you have a taste doppelganger—someone who has the same feelings about how good *Star Wars* is and how awful *Love Actually* is.

Then we also studied what content-based filtering and hybrid systems are.

Lastly, we used the GitHub API to create a recommendation engine based on collaborative filtering. The plan was to get all of the repositories that I'd starred over time and to then get all of the creators of those repositories to find out what repositories they'd starred. This enabled us to find out which users starred repositories were most similar to mine.

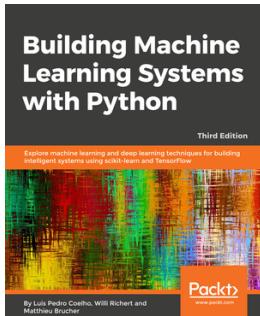
Summary

This chapter was just a small recap to take you back through all the projects we implemented.

I hope you enjoyed reading this book, and that the executions will help you to create your own projects in a similar manner!

Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:

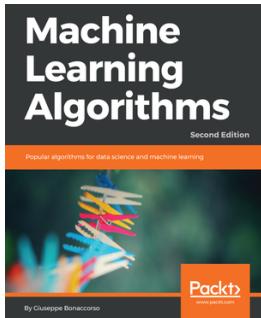


Building Machine Learning Systems with Python - Third Edition

Luis Pedro Coelho, Willi Richert, Matthieu Brucher

ISBN: 9781788623223

- Build a classification system that can be applied to text, images, and sound
- Employ Amazon Web Services (AWS) to run analysis on the cloud
- Solve problems related to regression using scikit-learn and TensorFlow
- Recommend products to users based on their past purchases
- Understand different ways to apply deep neural networks on structured data
- Address recent developments in the field of computer vision and reinforcement learning



Machine Learning Algorithms - Second Edition

Giuseppe Bonacorso

ISBN: 9781789347999

- Study feature selection and the feature engineering process
- Assess performance and error trade-offs for linear regression
- Build a data model and understand how it works by using different types of algorithm
- Learn to tune the parameters of Support Vector Machines (SVM)
- Explore the concept of natural language processing (NLP) and recommendation systems
- Create a machine learning architecture from scratch

Leave a review - let other readers know what you think

Please share your thoughts on this book with others by leaving a review on the site that you bought it from. If you purchased the book from Amazon, please leave us an honest review on this book's Amazon page. This is vital so that other potential readers can see and use your unbiased opinion to make purchasing decisions, we can understand what our customers think about our products, and our authors can see your feedback on the title that they have worked with Packt to create. It will only take a few minutes of your time, but is valuable to other potential customers, our authors, and Packt. Thank you!

Index

A

acquisition, data science workflow

 libraries and functions 10, 11

airfare pricing data

 sourcing 88

Anaconda

 about 48

 reference 48

anchoring effect 222

Anscombe's quartet 22

apartment listing data

 inspecting 66, 67, 68

 model, extending 86

 modeling 79, 80, 81

 models, used for making forecasts 82, 83, 85

 preparing 64, 65

 retrieving 50, 51, 52, 54, 57, 58, 59, 61, 63

 sourcing 50

 visualizing 70, 72, 73, 74, 75, 76, 77

apply function 34, 35

applymap function 35, 36

Artificial Intelligence Markup Language (AIML) 300

Asynchronous JavaScript (AJAX) 89

B

bag-of-words (BOW) 166

Bald's Leechbook 219

baseline accuracy 289

BeautifulSoup

 reference 56

 using 56

binary classification

 about 141, 142, 143

 dummy coding 143, 145

 model performance, examining 145, 146, 147,

 148, 149

target, creating for model 143

buy-and-hold strategy 230, 231, 232

C

chatbots

 building 303, 304, 306, 307, 309

 design 298, 300, 302, 303

 history 293, 294, 295, 298

 sequence-to-sequence modeling 309, 311, 313, 315, 317, 319

cheap airfare alert

 implementing 118

choropleth map 69

collaborative filtering

 about 322, 323, 324, 325

 cons 330

 product rating, predicting 325, 326, 327, 328

 pros 330

content-based filtering

 about 329, 330

 cons 331

 pros 330

convolutional filters 258, 260, 262, 265, 267

convolutional layers 258, 260, 262, 265, 267

convolutional neural networks

 about 255

 building, to classify images in Zalando Research

 dataset with Keras 272, 274, 277, 279, 281,

 284, 287, 289

 convolutional filters 258, 260, 262, 265, 267

 convolutional layers 258, 260, 262, 265, 267

 flattening 270, 271

 fully-connected layers 271

 max pooling layers 268, 270

 network topology 256, 257, 258

 output layers 271

D

daily personal newsletter
 setting up 182, 183, 184, 185, 186, 188
data mining fallacy 233
data science, workflow
 about 7, 8
 acquisition 8
 deployment 10
 evaluation 9
 inspection 8
 modeling 9
 preparation 9
data slicing 14
density-based spatial clustering of applications with
 noise (DBSCAN) 106
deployment, data science workflow
 libraries and functions 47
Disposos' Diapers 327
Document Object Model (DOM)
 about 93
 parsing, to extract pricing data 93, 95, 96, 98,
 100, 103, 106
dropout regularization 286
dummy variables 82
dynamic time warping 246

E

Elasticsearch
 using 331
ELIZA 294
Embedly API
 reference 164
 used, for downloading story bodies 164, 165
evaluation, data science workflow
 libraries and functions 38

F

fare data
 retrieving, with web scraping 89, 92
Fashion MNIST 272
feature baskets 329
feature engineering
 about 139
IPO performance, influencing with features 139,

140, 141
flattening 270, 271
Flight Explorer 88
folium
 reference 77
fully-connected layers 271

G

Generalized Extreme Studentized Deviate
 (Generalized ESD) 107
Google Maps API
 reference 70
Google Sheets
 setting up, through IFTTT 172, 173, 174, 175,
 176, 178, 180, 181
groupby function
 about 36, 37, 38
 reference 38
gspread
 about 177
 reference 177

H

headless browser 90
headlines
 exploring 202, 204, 205, 207
hybrid system 330
hyperparameters 257

I

If This Then That (IFTTT)
 about 113
 used, for sending real-time alerts 113, 117
 used, for setting up Google Sheets 172, 173,
 174, 175, 176, 178, 180, 181
 used, for setting up news feeds 172, 173, 174,
 175, 176, 178, 180, 181
image data
 exploring 196, 197, 199, 200
image-feature extraction 252, 253, 254, 255
Initial Public Offering (IPO)
 about 122, 123
 data, analyzing 131, 132
DataFrame, working with 126, 127, 128, 129,
 130, 131

market performance 124, 125, 126
stock performance, summarizing 132, 133, 134, 135, 136
strategy 136, 137, 138
inspection, data science workflow
 Jupyter Notebook 11, 12
 libraries and functions 11
 Pandas 13, 14, 15, 16, 17, 18, 19, 21
 visualization 21, 22
IPO market 123
IPoScoop
 reference 124
iris dataset
 reference 13
item-to-item filtering 327

J

Jupyter Notebook
 about 11, 12
 reference 11

K

Keras
 used, for building convolutional neural network to classify images in Zalando Research dataset 272, 274, 277, 279, 281, 284, 287, 289
kernel trick 170

L

lemmatization 167
logistic regression
 about 141, 257
 feature importance, generating 149, 150
 random forest classifier 151, 152

M

machine learning (ML)
 about 300
 environment, setting up 48
map function 33
market analysis
 types 221
matplotlib library 23, 24, 25, 26, 27, 29
max pooling layers 268, 270

maximum-margin hyperplane 169
Methicillin-resistant *Staphylococcus aureus* (MRSA) 219
modeling, data science workflow
 scikit-learn library 42, 44, 45, 47
 statsmodels library 38, 40, 41
momentum strategy 222
Multiple Listing Service (MLS) 50
musical genome
 building 329

N

National Association of Realtors (NAR) 50
Natural Language Processing (NLP) 166, 167, 168
Natural Language Toolkit (NLTK) 202
network topology 256, 257, 258
news feeds
 setting up, through IFTTT 172, 173, 174, 175, 176, 178, 180, 181
normal probability 108

O

OAuth 2 177
OkCupid 321
opening gap 139
outlier 106
outlier fares
 identifying, with anomaly detection techniques 106, 109, 112
output layers 271

P

Pandas 13, 14, 16, 17, 18, 19, 21
patsy package
 about 79
 reference 79
Pearson correlation coefficient 21
Pocket API
 reference 157
 used, for retrieving stories 157, 159, 160, 162, 163
Pocket Chrome Extension
 installing 155, 156
Pocket

about 155
reference 157
supervised training set, creating 155

predictive content scoring model
building 210, 211, 213, 214, 216, 218

preparation, data science workflow
apply function 34, 35
applymap function 35, 36
groupby function 36, 37, 38
libraries and functions 32
map function 33

Python Imaging Library (PIL) 273

Python Requests library
reference 51

Python Scientific Stack 6

Q

quantile-quantile (Q-Q) plot 108

R

random forest classifier 151, 152, 257
random forest regression 210
rank correlation coefficient 21
real-time alerts
sending, IFTTT used 113, 117

recommendation engine
building 331, 332, 333, 334, 335, 336, 337, 338, 339, 340, 341, 342, 343, 344, 345, 346, 347, 348

Red Queen's race 219

regression model
building 237, 239, 241, 243, 245, 247
example 241, 242
trades, evaluating 249, 250

RentHop
page elements, examining 55
reference 50

requests library
about 10, 11
reference 10

roadshow 123

S

scatterplot 25

scikit-learn library 42, 44, 45, 47

seaborn library
about 29, 30, 32
reference 29

sequence-to-sequence modeling
for chatbots 309, 311, 313, 315, 317, 319

shareability
features, exploring 195
headlines, exploring 202, 204, 205, 207
image data, exploring 196, 197, 199, 200
story content, exploring 207, 209

shared content
sourcing 192, 194

shared counts
sourcing 192, 194

Sharpe ratio 230

soft-margin SVM 170

statsmodels library
about 38, 40, 42
reference 38

stemming 167

stock market
research 221

stop words 167

stories
retrieving, Pocket API used 157, 159, 160, 162, 163

story bodies
downloading, Embedly API used 164, 165

story content
exploring 207, 209

Sublime Text
reference 52

subscription 123

supervised training set
creating, with Pocket 155

support vector machine (SVM) 46, 169, 170, 171

support vectors 170

T

Tay 296

term frequency-inverse document frequency (tf-idf)
167

term-document matrix 166, 323

tidy data 29

time series data 245

trading strategy
developing 223, 224, 225, 226, 227, 228
Turing Test 292

U

utility matrix 323

V

vancomycin 219
virality 190, 191

visualization, inspection
about 21, 22
matplotlib library 23, 24, 25, 26, 27, 29
seaborn library 29, 30, 32

Z

Zalando Research dataset
convolutional neural network, building to classify
images with Keras 272, 274, 277, 279, 281,
284, 287, 289