lick here to view code image

```
In [3]: tweets = []

In [4]: counts = {'total_tweets':   0, 'locations': 0}
```

Creating the `LocationListener`

For this example, the `LocationListener` will collect 50 tweets about `'football'`:

lick here to view code image

```
In [5]: from locationlistener import LocationListener

In [6]: location_listener = LocationListener(api,   counts_dict=counts,
   ...:     tweets_list=tweets, topic='football', limit=50)
   ...:
```

The `LocationListener` will use our utility function `get_tweet_content` to extract the screen name, tweet text and location from each tweet, place that data in a dictionary.

Configure and Start the Stream of Tweets

Next, let's set up our `Stream` to look for English language `'football'` tweets:

lick here to view code image

```
In [7]: import tweepy

In [8]: stream = tweepy.Stream(auth=api.auth,   listener=location_listener)

In [9]: stream.filter(track=['football'],   languages=['en'], is_async=False
```

ow wait to receive the tweets. Though we do not show them here (to save space), the `LocationListener` displays each tweet's screen name and text so you can see the live stream. If you're not receiving any (perhaps because it is not football season), you might want to type *Ctrl + C* to terminate the previous snippet then try again with a different search term.

Displaying the Location Statistics

When the next `In []` prompt displays, we can check how many tweets we processed, how many had locations and the percentage that had locations:

lick here to view code image

```
In [10]: counts['total_tweets']
Out[10]: 63

In [11]: counts['locations']
Out[11]: 50

In [12]: print(f'{counts["locations"] / counts["total_tweets"]:.1%}')
79.4%
```

In this particular execution, 79.4% of the tweets contained location data.

Geocoding the Locations

Now, let's use our `get_geocodes` utility function from `tweetutilities.py` to geocode the location of each tweet stored in the list tweets:

lick here to view code image

```
In [13]: from tweetutilities import get_geocodes

In [14]: bad_locations = get_geocodes(tweets)
Getting coordinates for tweet locations...
OpenMapQuest service timed out. Waiting.
OpenMapQuest service timed out. Waiting.
Done geocoding
```

Sometimes the OpenMapQuest geocoding service times out, meaning that it cannot handle your request immediately and you need to try again. In that case, our function `get_geocodes` displays a message, waits for a short time, then retries the geocoding request.

As you'll soon see, for each tweet with a *valid* location, the `get_geocodes` function adds to the tweet's dictionary in the `tweets` list two new keys—`'latitude'` and `'longitude'`. For the corresponding values, the function uses the tweet's coordinates that OpenMapQuest returns.

Displaying the Bad Location Statistics

When the next `In []` prompt displays, we can check the percentage of tweets that had invalid location data:

lick here to view code image

C

```
In [15]: bad_locations
Out[15]: 7

In [16]: print(f'{bad_locations   / counts["locations"]:.1%}')
14.0%
```

In this case, of the 50 tweets with location data, 7 (14%) had invalid locations.

Cleaning the Data

Before we plot the tweet locations on a map, let's use a pandas `DataFrame` to clean the data. When you create a `DataFrame` from the `tweets` list, it will contain the value `NaN` for the `'latitude'` and `'longitude'` of any tweet that did not have a valid location. We can remove any such rows by calling the `DataFrame`'s **`dropna` method**:

lick here to view code image

```
In [17]: import pandas as pd

In [18]: df = pd.DataFrame(tweets)

In [19]: df = df.dropna()
```

Creating a Map with Folium

Now, let's create a folium **`Map`** on which we'll plot the tweet locations:

lick here to view code image

```
In [20]: import folium

In [21]: usmap = folium.Map(location=[39.8283, -98.5795],
    ...:                    tiles='Stamen   Terrain',
    ...:                    zoom_start=5, detect_retina=True)
    ...:
```

The `location` keyword argument specifies a sequence containing latitude and longitude coordinates for the map's center point. The values above are the geographic center of the continental United States (`ttp://bit.ly/CenterOfTheUS`). It's possible that some of the tweets we plot will be outside the U.S. In this case, you will not see them initially when you open the map. You can zoom in and out using the **+** and **-** buttons at the top-left of the map, or you can pan the map by dragging it with the mouse to see anywhere in the world.

The `zoom_start` keyword argument specifies the map's initial zoom level, lower values show more of the world and higher values show less. On our system, `5` displays the entire continental United States. The `detect_retina` keyword argument enables folium to detect high-resolution screens. When it does, it requests higher-resolution maps from OpenStreetMap.org and changes the zoom level accordingly.

Creating Popup Markers for the Tweet Locations

Next, let's iterate through the `DataFrame` and add to the `Map` folium `Popup` objects containing each tweet's text. In this case, we'll use method **`itertuples`** to create tuples from each row of the `DataFrame`. Each tuple will contain a property for each `DataFrame` column:

lick here to view code image

```
In [22]: for t in df.itertuples():
    ...:     text = ': '.join([t.screen_name,   t.text])
    ...:     popup = folium.Popup(text, parse_html=True)
    ...:     marker = folium.Marker((t.latitude,   t.longitude),
    ...:                            popup=popup)
    ...:     marker.add_to(usmap)
    ...:
```

First, we create a string (`text`) containing the user's `screen_name` and tweet `text` separated by a colon. This will be displayed on the map if you click the corresponding marker. The second statement creates a folium **`Popup`** to display the text. The third statement creates a folium **`Marker`** object using a tuple to specify the `Marker`'s latitude and longitude. The `popup` keyword argument associates the tweet's `Popup` object with the new `Marker`. Finally, the last statement calls the `Marker`'s **`add_to` method** to specify the `Map` that will display the `Marker`.

Saving the Map

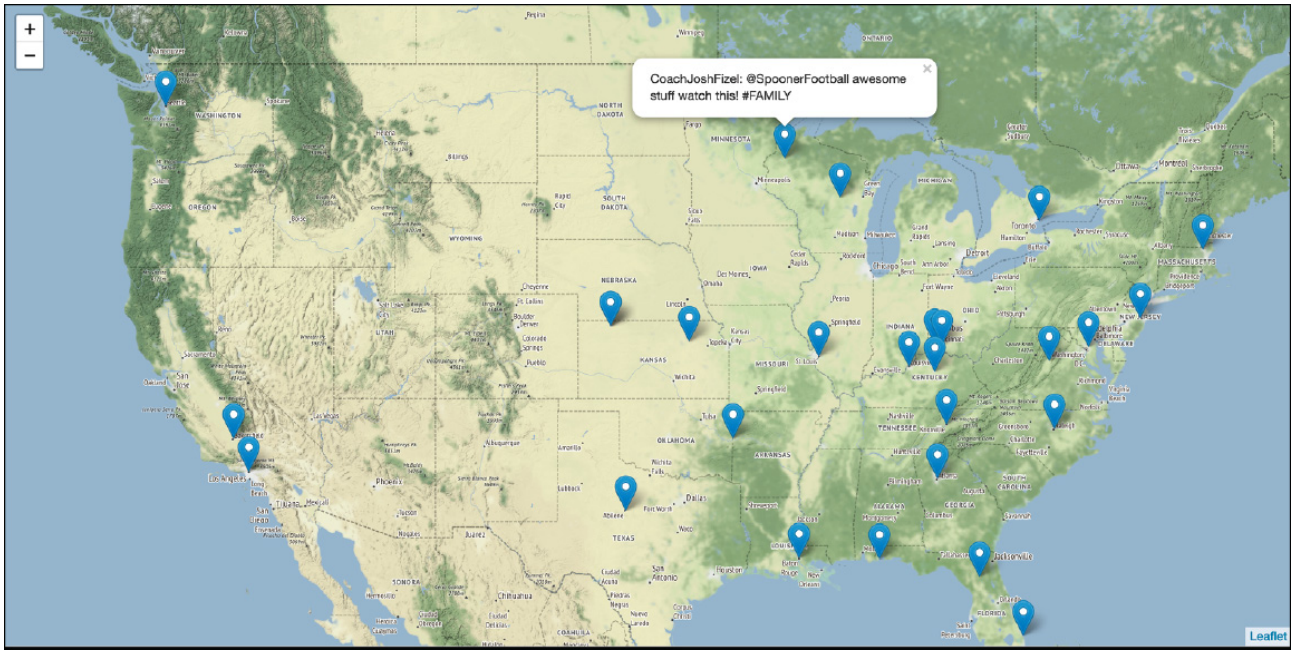
The last step is to call the `Map`'s **`save`** method to store the map in an HTML file, which you can then double click to open in your web browser:

lick here to view code image

```
In [23]: usmap.save('tweet_map.html')
```

The resulting map follows. The `Marker`s on your map will differ:

C



Map data © OpenStreetMap contributors.

The data is available under the Open Database License
ttp://www.openstreetmap.org/copyright.

12.15.2 Utility Functions in `tweetutilities.py`

Here we present the utility functions `get_tweet_content` and `get_geo_codes` used in the preceding section's IPython session. In each case, the line numbers start from 1 for discussion purposes. These are both defined in `tweetutilities.py`, which is included in the `ch12` examples folder.

`get_tweet_content` Utility Function

Function `get_tweet_content` receives a `Status` object (tweet) and creates a dictionary containing the tweet's `screen_name` (line 4), `text` (lines 7–10) and `location` (lines 12–13). The location is included only if the `location` keyword argument is `True`. For the tweet's text, we try to use the `full_text` property of an `extended_tweet`. If it's not available, we use the `text` property:

lick here to view code image

```
1 def get_tweet_content(tweet, location=False):
2     """Return dictionary with data from tweet   (a Status object)."""
3     fields = {}
4     fields['screen_name'] = tweet.user.screen_name
5
6     # get the tweet's text
7     try:
8         fields['text'] = tweet.extended_tweet.full_text
9     except:
10         fields['text'] = tweet.text
```

€

```
11
12      if location:
13          fields['location'] = tweet.user.location
14
15      return fields
```

`get_geocodes` Utility Function

Function `get_geocodes` receives a list of dictionaries containing tweets and geocodes their locations. If geocoding is successful for a tweet, the function adds the latitude and longitude to the tweet's dictionary in `tweet_list`. This code requires class **`OpenMapQuest`** from the `geopy` module, which we import into the file `tweetutilities.py` as follows:

```
from geopy import OpenMapQuest
```

lick here to view code image

```
1 def get_geocodes(tweet_list):
2     """Get the latitude and longitude for each   tweet's location.
3     Returns the number of tweets with invalid location   data."""
4     print('Getting coordinates for tweet locations...')
5     geo =   OpenMapQuest(api_key=keys.mapquest_key)   # geocoder
6     bad_locations   = 0
7
8     for tweet in tweet_list:
9         processed   = False
10         delay   = .1  # used if   OpenMapQuest times out to delay next c
11         while not processed:
12             try:  # get coordinates for   tweet['location']
13                 geo_location   = geo.geocode(tweet['location'])
14                 processed   = True
15             except:  # timed out, so wait   before trying again
16                 print('OpenMapQuest service timed out. Waiting.')
17                 time.sleep(delay)
18                 delay   += .1
19
20         if geo_location:
21             tweet['latitude'] = geo_location.latitude
22             tweet['longitude'] = geo_location.longitude
23         else:
24             bad_locations   += 1  #   tweet['location'] was invalid
25
26     print('Done   geocoding')
27     return bad_locations
```

The function operates as follows:

C

- Line 5 creates the `OpenMapQuest` object we'll use to geocode locations. The `api_key` keyword argument is loaded from the `keys.py` file you edited earlier.
- Line 6 initializes `bad_locations` which we use to keep track of the number of invalid locations in the tweet objects we collected.
- In the loop, lines 9–18 attempt to geocode the current tweet's location. Sometimes the OpenMapQuest geocoding service will time out, meaning that it's temporarily unavailable. This can happen if you make too many requests too quickly. So, the `while` loop continues executing as long as processed is `False`. In each iteration, this loop calls the `OpenMapQuest` object's **geocode method** with the tweet's location string as an argument. If successful, processed is set to `True` and the loop terminates. Otherwise, lines 16–18 display a time-out message, wait for `delay` seconds and increase the delay in case we get another time out. Line 17 calls the Python Standard Library `time` module's `sleep` method to pause the code execution.
- After the `while` loop terminates, lines 20–24 check whether location data was returned and, if so, add it to the tweet's dictionary. Otherwise, line 24 increments the `bad_locations` counter.
- Finally, the function prints a message that it's done geocoding and returns the `bad_locations` value.

12.15.3 Class `LocationListener`

Class `LocationListener` performs many of the same tasks we demonstrated in the prior streaming examples, so we'll focus on just a few lines in this class:

lick here to view code image

```
1 # locationlistener.py
2 """Receives tweets matching a search string   and stores a list of
3 dictionaries containing each tweet's   screen_name/text/location."""
4 import tweepy
5 from tweetutilities import   get_tweet_content
6
7 class LocationListener(tweepy.StreamListener):
8     """Handles incoming Tweet stream to get   location data."""
9
10     def __init__(self, api, counts_dict, tweets_list, topic,   limit=10):
11         """Configure the   LocationListener."""
12         self.tweets_list   = tweets_list
13         self.counts_dict   = counts_dict
14         self.topic   = topic
15         self.TWEET_LIMIT   = limit
```

C

```
        super().__init__(api)    # call superclass's init

    def on_status(self, status):
        """Called when Twitter pushes a new   tweet to you."""
        # get each tweet's screen_name, text and location
        tweet_data   = get_tweet_content(status, location=True)

        # ignore retweets and tweets that do not contain   the topic
        if (tweet_data['text'].startswith('RT') or
            self.topic.lower()   not in tweet_data['text'].lower()):
            return

        self.counts_dict['total_tweets'] += 1  # original tweet

        # ignore tweets with no location
        if not status.user.location:
            return

        self.counts_dict['locations'] += 1  # tweet with location
        self.tweets_list.append(tweet_data)  # store the tweet
        print(f'{status.user.screen_name}: {tweet_data["text"]}\n')

        # if TWEET_LIMIT is reached, return False to   terminate streamir
        return self.counts_dict['locations']   != self.TWEET_LIMIT
```

n this case, the `__init__` method receives a `counts` dictionary that we use to keep track of the total number of tweets processed and a `tweet_list` in which we store the dictionaries returned by the `get_tweet_content` utility function.

Method `on_status`:

- Calls `get_tweet_content` to get the screen name, text and location of each tweet.
- Ignores the tweet if it is a retweet or if the text does not include the topic we're searching for—we'll use only original tweets containing the search string.
- Adds 1 to the value of the `'total_tweets'` key in the `counts` dictionary to track the number of original tweets we process.
- Ignores tweets that have no location data.
- Adds 1 to the value of the `'locations'` key in the `counts` dictionary to indicate that we found a tweet with a location.
- Appends to the `tweets_list` the `tweet_data` dictionary that `get_tweet_content` returned.

I

- Displays the tweet's screen name and tweet text so you can see that the app is making progress.
- Checks whether the `TWEET_LIMIT` has been reached and, if so, returns `False` to terminate the stream.

12.16 WAYS TO STORE TWEETS

For analysis, you'll commonly store tweets in:

- CSV files—A file format that we introduced in the "Files and Exceptions" chapter.
- pandas `DataFrame`s in memory—CSV files can be loaded easily into `DataFrame`s for cleaning and manipulation.
- SQL databases—Such as MySQL, a free and open source relational database management system (RDBMS).
- NoSQL databases—Twitter returns tweets as JSON documents, so the natural way to store them is in a NoSQL JSON document database, such as MongoDB. Tweepy generally hides the JSON from the developer. If you'd like to manipulate the JSON directly, use the techniques we present in the " ig Data: Hadoop, Spark, NoSQL and oT" chapter, where we'll look at the PyMongo library.

12.17 TWITTER AND TIME SERIES

A time series is a sequence of values with timestamps. Some examples are daily closing stock prices, daily high temperatures at a given location, monthly U.S. job-creation numbers, quarterly earnings for a given company and more. Tweets are natural for time-series analysis because they're time stamped. In the "Machine Learning" chapter, we'll use a technique called simple linear regression to make predictions with time series. We'll take another look at time series in the "Deep Learning" chapter when we discuss recurrent neural networks.

12.18 WRAP-UP

In this chapter, we explored data mining Twitter, perhaps the most open and accessible of all the social media sites, and one of the most commonly used big-data sources. You created a Twitter developer account and connected to Twitter using your account credentials. We discussed Twitter's rate limits and some additional rules, and the importance of conforming to them.

B

e looked at the JSON representation of a tweet. We used Tweepy—one of the most widely used Twitter API clients—to authenticate with Twitter and access its APIs. We saw that tweets returned by the Twitter APIs contain much metadata in addition to a tweet's text. We determined an account's followers and whom an account follows, and looked at a user's recent tweets.

We used Tweepy `Cursor`s to conveniently request successive pages of results from various Twitter APIs. We used Twitter's Search API to download past tweets that met specified criteria. We used Twitter's Streaming API to tap into the flow of live tweets as they happened. We used the Twitter Trends API to determine trending topics for various locations and created a word cloud from trending topics.

We used the tweet-preprocessor library to clean and preprocess tweets to prepare them for analysis, and performed sentiment analysis on tweets. We used the folium library to create a map of tweet locations and interacted with it to see the tweets at particular locations. We enumerated common ways to store tweets and noted that tweets are a natural form of time series data. In the next chapter, we'll present IBM's Watson and its cognitive computing capabilities.

W

13. IBM Watson and Cognitive Computing

Objectives

In this chapter, you'll:

- See Watson's range of services and use their Lite tier to become familiar with them at no charge.
- Try lots of demos of Watson services.
- Understand what cognitive computing is and how you can incorporate it into your applications.
- Register for an IBM Cloud account and get credentials to use various services.
- Install the Watson Developer Cloud Python SDK to interact with Watson services.
- Develop a traveler's companion language translator app by using Python to weave together a mashup of the Watson Speech to Text, Language Translator and Text to Speech services.
- Check out additional resources, such as IBM Watson Redbooks that will help you jump start your custom Watson application development.

Outline

13.1 INTRODUCTION: IBM WATSON AND COGNITIVE COMPUTING

In hapter 1, we discussed some key IBM artificial-intelligence accomplishments, including beating the two best human Jeopardy! players in a $1 million match. Watson won the competition and IBM donated the prize money to charity. Watson simultaneously executed hundreds of language-analysis algorithms to locate correct answers in 200 million pages of content (including all of Wikipedia) requiring four terabytes of storage. [1] [2] IBM researchers trained Watson using machine-learning and reinforcement-learning techniques–we discuss machine learning in the next chapter. [3]

[1] ttps://www.techrepublic.com/article/ibm-watson-the-inside-story-of-how-the-jeopardy--winning-supercomputer-was-born-and-hat-it-wants-to-do-next/.

[2] ttps://en.wikipedia.org/wiki/Watson_(computer).

[3] ttps://www.aaai.org/Magazine/Watson/watson.php, *AI Magazine*, Fall 2010.

Early in our research for this book, we recognized the rapidly growing importance of Watson, so we placed Google Alerts on Watson and related topics. Through those alerts and the newsletters and blogs we follow, we accumulated 900+ current Watson-related articles, documentation pieces and videos. We investigated many competitive services and found Watson's "no credit card required" policy and free *Lite tier* services [4] to be among friendliest to people who'd like to experiment with Watson's services at no

harge.

[4] Always check the latest terms on IBMs website as the terms and services may change.

IBM Watson is a cloud-based cognitive-computing platform being employed across a wide range of real-world scenarios. Cognitive-computing systems simulate the pattern-recognition and decision-making capabilities of the human brain to "learn" as they consume more data. [5] [6] [7] We overview Watson's broad range of web services and provide a hands-on Watson treatment, demonstrating many Watson capabilities. The table on the next page shows just a few of the ways in which organizations are using Watson.

[5] ttp://whatis.techtarget.com/definition/cognitive-computing.

[6] ttps://en.wikipedia.org/wiki/Cognitive_computing.

[7] ttps://www.forbes.com/sites/bernardmarr/2016/03/23/what-veryone-should-know-about-cognitive-computing.

Watson offers an intriguing set of capabilities that you can incorporate into your applications. In this chapter, you'll set up an IBM Cloud account [8] and use the *Lite tier* and IBM's Watson demos to experiment with various web services, such as natural language translation, speech-to-text, text-to-speech, natural language understanding, chatbots, analyzing- text for tone and visual object recognition in images and video. We'll briefly overview some additional Watson services and tools.

[8] IBM Cloud previously was called Bluemix. Youll still see `bluemix` in many of this chapters URLs.

Watson use cases

ad targeting	fraud prevention	personal assistants
artificial intelligence	game playing	predictive maintenance
augmented intelligence	genetics	product recommendations
augmented reality	healthcare	
	image processing	

chatbots	IoT (Internet of Things)	robots and drones
closed captioning	language translation	self-driving cars
cognitive computing	machine learning	sentiment and mood analysis
conversational interfaces	malware detection	smart homes
crime prevention	medical diagnosis and treatment	sports
customer support	medical imaging	supply-chain management
detecting cyberbullying	music	threat detection
drug development	natural language processing	virtual reality
education	natural language understanding	voice analysis
facial recognition	object recognition	weather forecasting
finance		workplace safety

You'll install the Watson Developer Cloud Python Software Development Kit (SDK) for programmatic access to Watson services from your Python code. Then, in our hands-on implementation case study, you'll develop a traveler's companion translation app by quickly and conveniently *mashing up* several Watson services. The app enables English-only and Spanish-only speakers to communicate with one another verbally, despite the language barrier. You'll transcribe English and Spanish audio recordings to text, translate the text to the other language, then synthesize and play English and Spanish audio from the translated text.

Watson is a dynamic and evolving set of capabilities. During the time we worked on this book, new services were added and existing services were updated and/or removed multiple times. The descriptions of the Watson services and the steps we present were accurate as of the time of this writing. We'll post updates as necessary on the book's web page at `ww.deitel.com`.

w

3.2 IBM CLOUD ACCOUNT AND CLOUD CONSOLE

You'll need a free IBM Cloud account to access Watson's Lite tier services. Each service's description web page lists the service's tiered offerings and what you get with each tier. Though the Lite tier services limit your use, they typically offer what you'll need to familiarize yourself with Watson features and begin using them to develop apps. The limits are subject to change, so rather than list them here, we point you to each service's web page. IBM increased the limits significantly on some services while we were writing this book. Paid tiers are available for use in commercial-grade applications.

To get a free IBM Cloud account, follow the instructions at:

ttps://console.bluemix.net/docs/services/watson/index.html#about

You'll receive an e-mail. Follow its instructions to confirm your account. Then you can log in to the IBM Cloud console. Once there, you can go to the **Watson dashboard** at:

ttps://console.bluemix.net/developer/watson/dashboard

where you can:

- Browse the Watson services.
- Link to the services you've already registered to use.
- Look at the developer resources, including the Watson documentation, SDKs and various resources for learning Watson.
- View the apps you've created with Watson.

Later, you'll register for and get your credentials to use various Watson services. You can view and manage your list of services and your credentials in the **IBM Cloud dashboard** at:

ttps://console.bluemix.net/dashboard/apps

You can also click **Existing Services** in the Watson dashboard to get to this list.

13.3 WATSON SERVICES

This section overviews many of Watson's services and provides links to the details for

ach. Be sure to run the demos to see the services in action. For links to each Watson service's documentation and API reference, visit:

ttps://console.bluemix.net/developer/watson/documentation

We provide footnotes with links to each service's details. When you're ready to use a particular service, click the **Create** button on its details page to set up your credentials.

Watson Assistant

The **Watson Assistant service** [9] helps you build chatbots and virtual assistants that enable users to interact via natural language text. IBM provides a web interface that you can use to *train* the Watson Assistant service for specific scenarios associated with your app. For example, a weather chatbot could be trained to respond to questions like, "What is the weather forecast for New York City?" In a customer service scenario, you could create chatbots that answer customer questions and route customers to the correct department, if necessary. Try the demo at the following site to see some sample interactions:

[9] ttps://console.bluemix.net/catalog/services/watson-assistant-ormerly-conversation.

ttps://www.ibm.com/watson/services/conversation/demo/index.html#demo

Visual Recognition

The **Visual Recognition service** [0] enables apps to locate and understand information in images and video, including colors, objects, faces, text, food and inappropriate content. IBM provides predefined models (used in the service's demo), or you can train and use your own (as you'll do in the "Deep Learning" chapter). Try the following demo with the images provided and upload some of your own:

[0] ttps://console.bluemix.net/catalog/services/visual-ecognition.

ttps://watson-visual-recognition-duo-dev.ng.bluemix.net/

Speech to Text

The **Speech to Text service**,[1] which we'll use in building this chapter's app, converts speech audio files to text transcriptions of the audio. You can give the service keywords to "listen" for, and it tells you whether it found them, what the likelihood of a match

h

as and where the match occurred in the audio. The service can distinguish among multiple speakers. You could use this service to help implement voice-controlled apps, transcribe live audio and more. Try the following demo with its sample audio clips or upload your own:

[1] ttps://console.bluemix.net/catalog/services/speech-to-text.

ttps://speech-to-text-demo.ng.bluemix.net/

Text to Speech

The **Text to Speech service**, [2] which we'll also use in building this chapter's app, enables you to synthesize speech from text. You can use **Speech Synthesis Markup Language (SSML)** to embed instructions in the text for control over voice inflection, cadence, pitch and more. Currently, this service supports English (U.S. and U.K.), French, German, Italian, Spanish, Portuguese and Japanese. Try the following demo with its plain sample text, its sample text that includes SSML and text that you provide:

[2] ttps://console.bluemix.net/catalog/services/text-to-speech.

ttps://text-to-speech-demo.ng.bluemix.net/

Language Translator

The **Language Translator service**, [3] which we'll also use in building in this chapter's app, has two key components:

[3] ttps://console.bluemix.net/catalog/services/language-ranslator.

- translating text between languages and
- identifying text as being written in one of over 60 languages.

Translation is supported to and from English and many languages, as well as between other languages. Try translating text into various languages with the following demo:

ttps://language-translator-demo.ng.bluemix.net/

Natural Language Understanding

The **Natural Language Understanding service** [4] analyzes text and produces

nformation including the text's overall sentiment and emotion and keywords ranked by their relevance. Among other things, the service can identify

[4] ttps://console.bluemix.net/catalog/services/natural-language-nderstanding.

- people, places, job titles, organizations, companies and quantities.
- categories and concepts like sports, government and politics.
- parts of speech like subjects and verbs.

You also can train the service for industry- and application-specific domains with Watson Knowledge Studio (discussed shortly). Try the following demo with its sample text, with text that you paste in or by providing a link to an article or document online:

ttps://natural-language-understanding-demo.ng.bluemix.net/

Discovery

The **Watson Discovery service**[5] shares many features with the Natural Language Understanding service but also enables enterprises to store and manage documents. So, for example, organizations can use Watson Discovery to store all their text documents and be able to use natural language understanding across the entire collection. Try this service's demo, which enables you to search recent news articles for companies:

[5] ttps://console.bluemix.net/catalog/services/discovery.

ttps://discovery-news-demo.ng.bluemix.net/

Personality Insights

The **Personality Insights service**[6] analyzes text for personality traits. According to the service description, it can help you "gain insight into how and why people think, act, and feel the way they do. This service applies linguistic analytics and personality theory to infer attributes from a person's unstructured text." This information could be used to target product advertising at the people most likely to purchase those products. Try the following demo with tweets from various Twitter accounts or documents built into the demo, with text documents that you paste into the demo or with your own Twitter account:

[6] ttps://console.bluemix.net/catalog/services/personality-

h

nsights.

ttps://personality-insights-livedemo.ng.bluemix.net/

Tone Analyzer

The **Tone Analyzer service** [7] analyzes text for its tone in three categories:

[7] ttps://console.bluemix.net/catalog/services/tone-analyzer.

- emotions—anger, disgust, fear, joy, sadness.
- social propensities—openness, conscientiousness, extroversion, agreeableness and emotional range.
- language style—analytical, confident, tentative.

Try the following demo with sample tweets, a sample product review, a sample e-mail or text you provide. You'll see the tone analyses at both the document and sentence levels:

ttps://tone-analyzer-demo.ng.bluemix.net/

Natural Language Classifier

You *train* the **Natural Language Classifier service** [8] with sentences and phrases that are specific to your application and classify each sentence or phrase. For example, you might classify "I need help with your product" as "tech support" and "My bill is incorrect" as "billing." Once you've trained your classifier, the service can receive sentences and phrases, then use Watson's cognitive computing capabilities and your classifier to return the best matching classifications and their match probabilities. You might then use the returned classifications and probabilities to determine the next steps in your app. For example, in a customer service app where someone is calling in with a question about a particular product, you might use Speech to Text to convert a question into text, use the Natural Language Classifier service to classify the text, then route the call to the appropriate person or department. This service *does not offer a Lite tier*. In the following demo, enter a question about the weather—the service will respond by indicating whether your question was about the temperature or the weather conditions:

[8] ttps://console.bluemix.net/catalog/services/natural-language-

lassifier.

ttps://natural-language-classifier-demo.ng.bluemix.net/

Synchronous and Asynchronous Capabilities

Many of the APIs we discuss throughout the book are **synchronous**—when you call a function or method, the program *waits* for the function or method to return before moving on to the next task. **Asynchronous** programs can start a task, continue doing other things, then be *notified* when the original task completes and returns its results. Many Watson services offer both synchronous and asynchronous APIs.

The Speech to Text demo is a good example of asynchronous APIs. The demo processes sample audio of two people speaking. As the service transcribes the audio, it returns intermediate transcription results, even if it has not yet been able to distinguish among the speakers. The demo displays these intermediate results in parallel with the service's continued work. Sometimes the demo displays "Detecting speakers" while the service figures out who is speaking. Eventually, the service sends updated transcription results for distinguishing among the speakers, and the demo then replaces the prior transcription results.

With today's multi-core computers and multi-computer clusters, the asynchronous APIs can help you improve program performance. However, programming with them can be more complicated than programming with synchronous APIs. When we discuss installing the Watson Developer Cloud Python SDK, we provide a link to the SDK's code examples on GitHub, where you can see examples that use synchronous and asynchronous versions of several services. Each service's API reference provides the complete details.

13.4 ADDITIONAL SERVICES AND TOOLS

In this section, we overview several Watson advanced services and tools.

Watson Studio

Watson Studio[9] is the new Watson interface for creating and managing your Watson projects and for collaborating with your team members on those projects. You can add data, prepare your data for analysis, create Jupyter Notebooks for interacting with your data, create and train models and work with Watson's deep-learning capabilities. Watson Studio offers a single-user Lite tier. Once you've set up your Watson Studio Lite access by clicking **Create** on the service's details web page

h

[9] ttps://console.bluemix.net/catalog/services/data-science-xperience.

ttps://console.bluemix.net/catalog/services/data-science-experience

you can access Watson Studio at

ttps://dataplatform.cloud.ibm.com/

Watson Studio contains preconfigured projects.[0] Click **Create a project** to view them:

[0] ttps://dataplatform.cloud.ibm.com/.

- Standard—"Work with any type of asset. Add services for analytical assets as you need them."
- Data Science—"Analyze data to discover insights and share your findings with others."
- Visual Recognition—"Tag and classify visual content using the Watson Visual Recognition service."
- Deep Learning—"Build neural networks and deploy deep learning models."
- Modeler—"Build modeler flows to train SPSS models or design deep neural networks."
- Business Analytics—"Create visual dashboards from your data to gain insights faster."
- Data Engineering—"Combine, cleanse, analyze, and shape data using Data Refinery."
- Streams Flow—"Ingest and analyze streaming data using the Streaming Analytics service."

Knowledge Studio

Various Watson services work with *predefined* models, but also allow you to provide custom models that are trained for specific industries or applications. Watson's **Knowledge Studio**[1] helps you build custom models. It allows enterprise teams to

ork together to create and train new models, which can then be deployed for use by Watson services.

[1] ttps://console.bluemix.net/catalog/services/knowledge-studio.

Machine Learning

The **Watson Machine Learning service** [2] enables you to add predictive capabilities to your apps via popular machine-learning frameworks, including Tensorflow, Keras, scikit-learn and others. You'll use scikit-learn and Keras in the next two chapters.

[2] ttps://console.bluemix.net/catalog/services/machine-learning.

Knowledge Catalog

The **Watson Knowledge Catalog** [3, 4] is an advanced enterprise-level tool for securely managing, finding and sharing your organization's data. The tool offers:

[3] ttps://medium.com/ibm-watson/introducing-ibm-watson-knowledge-atalog-cf42c13032c1.

[4] ttps://dataplatform.cloud.ibm.com/docs/content/catalog/overview-kc.html.

- Central access to an enterprise's local and cloud-based data and machine learning models.
- Watson Studio support so users can find and access data, then easily use it in machine-learning projects.
- Security policies that ensure only the people who should have access to specific data actually do.
- Support for over 100 data cleaning and wrangling operations.
- And more.

Cognos Analytics

The IBM **Cognos Analytics** [5] service, which has a 30-day free trial, uses AI and machine learning to discover and visualize information in your data, without any programming on your part. It also provides a natural-language interface that enables you to ask questions which Cognos Analytics answers based on the knowledge it gathers

from your data.

[5] ttps://www.ibm.com/products/cognos-analytics.

13.5 WATSON DEVELOPER CLOUD PYTHON SDK

In this section, you'll install the modules required for the next section's full-implementation Watson case study. For your coding convenience, IBM provides the **Watson Developer Cloud Python SDK** (software development kit). Its **`watson_developer_cloud` module** contains classes that you'll use to interact with Watson services. You'll create objects for each service you need, then interact with the service by calling the object's methods.

To install the SDK[6] open an Anaconda Prompt (Windows; open as Administrator), Terminal (macOS/Linux) or shell (Linux), then execute the following command[7]:

[6] For detailed installation instructions and troubleshooting tips, see ttps://github.com/watson-developer-cloud/python-dk/blob/develop/README.md.

[7] Windows users might need to install Microsofts C++ build tools from ttps://visualstudio.microsoft.com/visual-cpp-build-tools/, then install the `watson-developer-cloud` module.

```
pip install --upgrade watson-developer-cloud
```

Modules We'll Need for Audio Recording and Playback

You'll also need two additional modules for audio recording (PyAudio) and playback (PyDub). To install these, use the following commands[8]:

[8] Mac users might need to first execute `conda install -c conda-forge portaudio`.

```
pip install pyaudio
pip install pydub
```

SDK Examples

On GitHub, IBM provides sample code demonstrating how to access Watson services using the Watson Developer Cloud Python SDK's classes. You can find the examples at:

ttps://github.com/watson-developer-cloud/python-sdk/tree/master/examples

13.6 CASE STUDY: TRAVELER'S COMPANION TRANSLATION APP

Suppose you're traveling in a Spanish-speaking country, but you do not speak Spanish, and you need to communicate with someone who does not speak English. You could use a translation app to speak in English, and the app could translate that, then speak it in Spanish. The Spanish-speaking person could then respond, and the app could translate that and speak it to you in English.

Here, you'll use three powerful IBM Watson services to implement such a traveler's companion translation app,[9] enabling people who speak different languages to converse in near real time. Combining services like this is known as creating a **mashup**. This app also uses simple file-processing capabilities that we introduced in the "Files and Exceptions" chapter.

[9] These services could change in the future. If they do, well post updates on the books web page at `ttp://www.deitel.com/books/IntroToPython`.

13.6.1 Before You Run the App

You'll build this app using the Lite (free) tiers of several IBM Watson services. Before executing the app, make sure that you've registered for an IBM Cloud account, as we discussed earlier in the chapter, so you can get credentials for each of the three services the app uses. Once you have your credentials (described below), you'll insert them in our `keys.py` file (located in the `ch13` examples folder) that we import into the example. Never share your credentials.

As you configure the services below, each service's credentials page also shows you the service's URL. These are the default URLs used by the Watson Developer Cloud Python SDK, so you do not need to copy them. In ection 13.6.3, we present the `SimpleLanguageTranslator.py` script and a detailed walkthrough of the code.

Registering for the Speech to Text Service

This app uses the Watson Speech to Text service to transcribe English and Spanish audio files to English and Spanish text, respectively. To interact with the service, you must get a username and password. To do so:

1. ***Create a Service Instance:*** Go to

`ttps://console.bluemix.net/catalog/services-/speech-to-text` and click the **Create** button on the bottom of the page. This auto-generates an API key for you and takes you to a tutorial for working with the Speech to Text service.

2. ***Get Your Service Credentials:*** To see your API key, click **Manage** at the top-left of the page. To the right of **Credentials**, click **Show credentials**, then copy the **API Key**, and paste it into the variable `speech_to_text_key`'s string in the `keys.py` file provided in this chapter's `ch13` examples folder.

Registering for the Text to Speech Service

In this app, you'll use the Watson Text to Speech service to synthesize speech from text. This service also requires you to get a username and password. To do so:

1. ***Create a Service Instance:*** Go to `ttps://console.bluemix.net/catalog/services/text-to-speech` and click the **Create** button on the bottom of the page. This auto-generates an API key for you and takes you to a tutorial for working with the Text to Speech service.

2. ***Get Your Service Credentials:*** To see your API key, click **Manage** at the top-left of the page. To the right of **Credentials**, click **Show credentials**, then copy the **API Key** and paste it into the variable `text_to_speech_key`'s string in the `keys.py` file provided in this chapter's `ch13` examples folder.

Registering for the Language Translator Service

In this app, you'll use the Watson Language Translator service to pass text to Watson and receive back the text translated into another language. This service requires you to get an API key. To do so:

1. ***Create a Service Instance:*** Go to `ttps://console.bluemix.net/catalog/services-/language-ranslator` and click the **Create** button on the bottom of the page. This auto-generates an API key for you and takes you to a page to manage your instance of the service.

2. ***Get Your Service Credentials:*** To the right of **Credentials**, click **Show credentials**, then copy the **API Key** and paste it into the variable `translate_key`'s string in the `keys.py` file provided in this chapter's `ch13` examples folder.

h

Retrieving Your Credentials

To view your credentials at any time, click the appropriate service instance at:

ttps://console.bluemix.net/dashboard/apps

13.6.2 Test-Driving the App

Once you've added your credentials to the script, open an Anaconda Prompt (Windows), a Terminal (macOS/Linux) or a shell (Linux). Run the script[o] by executing the following command from the `ch13` examples folder:

[o]The `pydub.playback` module we use in this app issues a warning when you run our script. The warning has to do with module features we dont use and can be ignored. To eliminate this warning, you can install `ffmpeg` for Windows, macOS or Linux from `ttps://www.ffmpeg.org`.

```
ipython SimpleLanguageTranslator.py
```

Processing the Question

The app performs 10 steps, which we point out via comments in the code. When the app begins executing:

Step 1 prompts for and records a question. First, the app displays:

```
Press Enter then ask your question in English
```

and waits for you to press *Enter*. When you do, the app displays:

```
Recording 5 seconds of audio
```

Speak your question. We said, "Where is the closest bathroom?" After five seconds, the app displays:

```
Recording complete
```

Step 2 interacts with Watson's Speech to Text service to transcribe your audio to text and displays the result:

h

```
English: where is the closest bathroom
```

Step 3 then uses Watson's Language Translator service to translate the English text to Spanish and displays the translated text returned by Watson:

```
Spanish: ¿Dónde está el baño más cercano?
```

Step 4 passes this Spanish text to Watson's Text to Speech service to convert the text to an audio file.

Step 5 plays the resulting Spanish audio file.

Processing the Response

At this point, we're ready to process the Spanish speaker's response.

Step 6 displays:

```
Press Enter then speak the Spanish answer
```

and waits for you to press *Enter*. When you do, the app displays:

```
Recording 5 seconds of audio
```

and the Spanish speaker records a response. We do not speak Spanish, so we used Watson's Text to Speech service to *prerecord* Watson saying the Spanish response "El baño más cercano está en el restaurante," then played that audio loud enough for our computer's microphone to record it. We provided this prerecorded audio for you as `SpokenResponse.wav` in the `ch13` folder. If you use this file, play it quickly after pressing *Enter* above as the app records for only 5 seconds.[1] To ensure that the audio loads and plays quickly, you might want to play it once before you press *Enter* to begin recording. After five seconds, the app displays:

[1]For simplicity, we set the app to record five seconds of audio. You can control the duration with the variable `SECONDS` in function `record_audio`. Its possible to create a recorder that begins recording once it detects sound and stops recording after a period of silence, but the code is more complicated.

```
Recording complete
```

Step 7 interacts with Watson's Speech to Text service to transcribe the Spanish audio to text and displays the result:

```
Spanish response: el baño más cercano está en el restaurante
```

Step 8 then uses Watson's Language Translator service to translate the Spanish text to English and displays the result:

```
English response: The nearest bathroom is in the restaurant
```

Step 9 passes the English text to Watson's Text to Speech service to convert the text to an audio file.

Step 10 then plays the resulting English audio.

13.6.3 `SimpleLanguageTranslator.py` Script Walkthrough

In this section, we present the `SimpleLanguageTranslator.py` script's source code, which we've divided into small consecutively numbered pieces. Let's use a top-down approach as we did in the "Control Statements" chapter. Here's the top:

Create a translator app that enables English and Spanish speakers to communicate.

The first refinement is:

Translate a question spoken in English into Spanish speech.

Translate the answer spoken in Spanish into English speech.

We can break the first line of the second refinement into five steps:

Step 1: Prompt for then record English speech into an audio file.

Step 2: Transcribe the English speech to English text.

Step 3: Translate the English text into Spanish text.

Step 4: Synthesize the Spanish text into Spanish speech and save it into an audio file.

Step 5: Play the Spanish audio file.

e can break the second line of the second refinement into five steps:

Step 6: Prompt for then record Spanish speech into an audio file.

Step 7: Transcribe the Spanish speech to Spanish text.

Step 8: Translate the Spanish text into English text.

Step 9: Synthesize the English text into English speech and save it into an audio file.

Step 10: Play the English audio.

This top-down development makes the benefits of the divide-and-conquer approach clear, focusing our attention on small pieces of a more significant problem.

In this section's script, we implement the 10 steps specified in the second refinement. **Steps 2** and **7** use the Watson Speech to Text service, **Steps 3** and **8** use the Watson Language Translator service, and **Steps 4** and **9** use the Watson Text to Speech service.

Importing Watson SDK Classes

Lines 4–6 import classes from the `watson_developer_cloud` module that was installed with the Watson Developer Cloud Python SDK. Each of these classes uses the Watson credentials you obtained earlier to interact with a corresponding Watson service:

- Class **`SpeechToTextV1`**[2] enables you to pass an audio file to the Watson Speech to Text service and receive a JSON[3] document containing the text transcription.

  [2]The `V1` in the class name indicates the services version number. As IBM revises its services, it adds new classes to the `watson_developer_cloud` module, rather than modifying the existing classes. This ensures that existing apps do not break when the services are updated. The Speech to Text and Text to Speech services are each Version 1 (`V1`) and the Language Translator service is Version 3 (`V3`) at the time of this writing.

  [3]We introduced JSON in the previous chapter, Data Mining Twitter.

- Class **`LanguageTranslatorV3`** enables you to pass text to the Watson Language Translator service and receive a JSON document containing the translated text.

- Class **`TextToSpeechV1`** enables you to pass text to the Watson Text to Speech

service and receive audio of the text spoken in a specified language.

lick here to view code image

```
1 # SimpleLanguageTranslator.py
2 """Use IBM Watson Speech to Text, Language   Translator and Text to Spe
3    APIs to enable English and Spanish speakers to   communicate."""
4 from watson_developer_cloud import   SpeechToTextV1
5 from watson_developer_cloud import   LanguageTranslatorV3
6 from watson_developer_cloud import   TextToSpeechV1
```

Other Imported Modules

Line 7 imports the `keys.py` file containing your Watson credentials. Lines 8–11 import modules that support this app's audio-processing capabilities:

- The `pyaudio` module enables us to record audio from the microphone.
- `pydub` and `pydub.playback` modules enable us to load and play audio files.
- The Python Standard Library's `wave` module enables us to save WAV (Waveform Audio File Format) files. WAV is a popular audio format originally developed by Microsoft and IBM. This app uses the `wave` module to save the recorded audio to a `.wav` file that we send to Watson's Speech to Text service for transcription.

lick here to view code image

```
7 import keys  # contains your API   keys for accessing Watson services
8 import pyaudio  # used to record   from mic
9 import pydub  # used to load a   WAV file
10 import pydub.playback  # used to   play a WAV file
11 import wave  # used to save a   WAV file
12
```

Main Program: Function `run_translator`

Let's look at the main part of the program defined in function `run_translator` (lines 13–54), which calls the functions defined later in the script. For discussion purposes, we broke `run_translator` into the 10 steps it performs. In **Step 1** (lines 15–17), we prompt in English for the user to press *Enter*, then speak a question. Function

C

`record_audio` then records audio for five seconds and stores it in the file `english.wav`:

lick here to view code image

```
13 def run_translator():
14     """Calls the functions that interact with   Watson services."""
15     # Step 1: Prompt for then record English speech into an   audio fi
16     input('Press Enter then ask your question in English')
17     record_audio('english.wav')
18
```

In **Step 2**, we call function `speech_to_text`, passing the file `english.wav` for transcription and telling the Speech to Text service to transcribe the text using its *predefined* model `'en-US_BroadbandModel'`.[4] We then display the transcribed text:

[4]For most languages, the Watson Speech to Text service supports *broadband* and *narrowband* models. Each has to do with the audio quality. For audio captured at 16 kHZ and higher, IBM recommends using the broadband models. In this app, we capture the audio at 44.1 kHZ.

lick here to view code image

```
19     # Step 2: Transcribe the English speech to English text
20     english =   speech_to_text(
21         file_name='english.wav', model_id='en-US_BroadbandModel')
22     print('English:', english)
23
```

In **Step 3**, we call function `translate`, passing the transcribed text from **Step 2** as the text to translate. Here we tell the Language Translator service to translate the text using its *predefined* model `'en-es'` to translate from English (`en`) to Spanish (`es`). We then display the Spanish translation:

lick here to view code image

```
24     # Step 3: Translate the English text into Spanish text
25     spanish =   translate(text_to_translate=english, model='en-es')
26     print('Spanish:', spanish)
27
```

C

In **Step 4**, we call function `text_to_speech`, passing the Spanish text from **Step 3** for the Text to Speech service to speak using its voice `'es-US_SofiaVoice'`. We also specify the file in which the audio should be saved:

lick here to view code image

```
28      # Step 4: Synthesize the Spanish text into Spanish speech
29      text_to_speech(text_to_speak=spanish,   voice_to_use='es-US_SofiaV
30          file_name='spanish.wav')
31
```

In **Step 5**, we call function `play_audio` to play the file `'spanish.wav'`, which contains the Spanish audio for the text we translated in **Step 3**.

lick here to view code image

```
32      # Step 5: Play the Spanish audio file
33      play_audio(file_name='spanish.wav')
34
```

Finally, **Steps 6–10** repeat what we did in **Steps 1–5**, but for Spanish speech to English speech:

- **Step 6** records the *Spanish* audio.
- **Step 7** transcribes the Spanish audio to Spanish text using the Speech to Text service's predefined model `'es-ES_BroadbandModel'`.
- **Step 8** translates the Spanish text to English text using the Language Translator Service's `'es-en'` (Spanish-to-English) model.
- **Step 9** creates the English audio using the Text to Speech Service's voice `'en-US_AllisonVoice'`.
- **Step 10** plays the English audio.

lick here to view code image

```
35      # Step 6: Prompt for then record Spanish speech into an   audio fi
36      input('Press Enter then speak the Spanish answer')
37      record_audio('spanishresponse.wav')
```

C

```
38
39      # Step 7: Transcribe the Spanish speech to Spanish text
40      spanish =   speech_to_text(
41          file_name='spanishresponse.wav', model_id='es-ES_BroadbandMode
42      print('Spanish response:', spanish)
43
44      # Step 8: Translate the Spanish text into English text
45      english =   translate(text_to_translate=spanish, model='es-en')
46      print('English response:', english)
47
48      # Step 9: Synthesize the English text into English speech
49      text_to_speech(text_to_speak=english,
50          voice_to_use='en-US_AllisonVoice',
51          file_name='englishresponse.wav')
52
53      # Step 10: Play the English audio
54      play_audio(file_name='englishresponse.wav')
55
```

ow let's implement the functions we call from ***Steps 1*** through ***10***.

Function `speech_to_text`

To access Watson's Speech to Text service, function `speech_to_text` (lines 56–87) creates a `SpeechToTextV1` object named `stt` (short for speech-to-text), passing as the argument the API key you set up earlier. The `with` statement (lines 62–65) opens the audio file specified by the `file_name` parameter and assigns the resulting file object to `audio_file`. The open mode `'rb'` indicates that we'll read (`r`) binary data (`b`)—audio files are stored as bytes in binary format. Next, lines 64–65 use the `SpeechToTextV1` object's **`recognize` method** to invoke the Speech to Text service. The method receives three keyword arguments:

- `audio` is the file (`audio_file`) to pass to the Speech to Text service.
- `content_type` is the media type of the file's contents—`'audio/wav'` indicates that this is an audio file stored in WAV format.[5]

[5] Media types were formerly known as **MIME (Multipurpose Internet Mail Extensions) types** a standard that specifies data formats, which programs can use to interpret data correctly.

- `model` indicates which spoken language model the service will use to recognize the speech and transcribe it to text. This app uses predefined models—either `'en-US_BroadbandModel'` (for English) or `'es-ES_BroadbandModel'` (for

Spanish).

lick here to view code image

```
def speech_to_text(file_name, model_id):
    """Use Watson Speech to Text to convert   audio file to text."""
    # create Watson Speech to Text client
    stt =   SpeechToTextV1(iam_apikey=keys.speech_to_text_key)

    # open the audio file
    with open(file_name, 'rb') as audio_file:
        # pass the file to Watson for transcription
        result   = stt.recognize(audio=audio_file,
            content_type='audio/wav', model=model_id).get_result()

    # Get the 'results' list. This may contain intermediate   and fina
    # results, depending on method recognize's arguments.   We asked
    # for only final results, so this list contains one   element.
    results_list   = result['results']

    # Get the final speech recognition result--the list's   only eleme
    speech_recognition_result    = results_list[0]

    # Get the 'alternatives' list. This may contain   multiple alterna
    # transcriptions, depending on method recognize's   arguments. We
    # not ask for alternatives, so this list contains one   element.
    alternatives_list   = speech_recognition_result['alternatives']

    # Get the only alternative transcription from   alternatives_list.
    first_alternative   = alternatives_list[0]

    # Get the 'transcript' key's value, which contains the   audio's
    # text transcription.
    transcript   = first_alternative['transcript']

    return transcript  # return the   audio's text transcription

```

The `recognize` method returns a `DetailedResponse` object. Its `getResult` method returns a JSON document containing the transcribed text, which we store in `result`. The JSON will look similar to the following but depends on the question you ask:

C

```
{
  "results": [                                          Line 70
    {                                                   Line 73
      "alternatives": [                                 Line 78
        {                                               Line 81
          "confidence": 0.983,
          "transcript": "where is the closest bathroom " Line 85
        }
      ],
      "final": true
    }
  ],
  "result_index": 0
}
```

The JSON contains *nested* dictionaries and lists. To simplify navigating this data structure, lines 70–85 use separate small statements to "pick off" one piece at a time until we get the transcribed text—`"where is the closest bathroom "`, which we then return. The boxes around portions of the JSON and the line numbers in each box correspond to the statements in lines 70–85. The statements operate as follows:

- Line 70 assigns to `results_list` the list associated with the key `'results'`:

  lick here to view code image

  ```
  results_list = result['results']
  ```

  Depending on the arguments you pass to method `recognize`, this list may contain intermediate and final results. Intermediate results might be useful, for example, if you were transcribing live audio, such as a newscast. We asked for only final results, so this list contains one element.[6]

  [6]For method `recognizes` arguments and JSON response details, see `ttps://www.ibm.com/watson/developercloud/speech-to-ext/api/v1/python.html?python#recognize-sessionless`.

- Line 73 assigns to `speech_recognition_result` the final speech-recognition result—the only element in `results_list`:

  lick here to view code image

  ```
  speech_recognition_result  = results_list[0]
  ```

- Line 78

lick here to view code image

```
alternatives_list = speech_recognition_result['alternatives']
```

assigns to `alternatives_list` the list associated with the key `'alternatives'`. This list may contain multiple alternative transcriptions, depending on method `recognize`'s arguments. The arguments we passed result in a one-element list.

- Line 81 assigns to `first_alternative` the only element in `alternatives_list`:

  lick here to view code image

  ```
  first_alternative = alternatives_list[0]
  ```

- Line 85 assigns to `transcript` the `'transcript'` key's value, which contains the audio's text transcription:

  lick here to view code image

  ```
  transcript = first_alternative['transcript']
  ```

- Finally, line 87 returns the audio's text transcription.

Lines 70–85 could be replaced with the denser statement

lick here to view code image

```
return result['results'][0]['alternatives'][0]['transcript']
```

but we prefer the separate simpler statements.

Function `translate`

To access the Watson Language Translator service, function `translate` (lines 89–111) first creates a `LanguageTranslatorV3` object named `language_translator`, passing as arguments the service version (`'2018-05-31'`[7]), the API Key you set up earlier and the service's URL. Lines 93–94 use the `LanguageTranslatorV3` object's

`translate` method to invoke the Language Translator service, passing two keyword arguments:

[7]According to the Language Translator services API reference, `'2018-05-31'` is the current version string at the time of this writing. IBM changes the version string only if they make API changes that are not backward compatible. Even when they do, the service will respond to your calls using the API version you specify in the version string. For more details, see `ttps://www.ibm.com/watson-/developercloud/language-translator/api/v3/python.html? ython#versioning`.

- `text` is the string to translate to another language.
- `model_id` is the predefined model that the Language Translator service will use to understand the original text and translate it into the appropriate language. In this app, `model` will be one of IBM's *predefined* translation models—`'en-es'` (for English to Spanish) or `'es-en'` (for Spanish to English).

lick here to view code image

```
89 def translate(text_to_translate, model):
90     """Use Watson Language Translator to   translate English to Spanis
91        (en-es) or Spanish to English (es-en) as specified by   model."
92     # create Watson Translator client
93     language_translator   = LanguageTranslatorV3(version='2018-05-31',
94         iam_apikey=keys.translate_key)
95
96     # perform the translation
97     translated_text   = language_translator.translate(
98         text=text_to_translate,   model_id=model).get_result()
99
100     # Get 'translations' list. If method translate's text   argument
101     # multiple strings, the list will have multiple   entries. We pas
102     # one string, so the list contains only one element.
103     translations_list   = translated_text['translations']
104
105     # get translations_list's only element
106     first_translation   = translations_list[0]
107
108     # get 'translation' key's value, which is the   translated text
109     translation   = first_translation['translation']
110
111     return translation  # return the   translated string
112
```

The method returns a `DetailedResponse`. That object's `getResult` method returns a JSON document, like:

```
{
  "translations": [                                    Line 103
    {                                                  Line 106
      "translation": "¿Dónde está el baño más cercano? "  Line 109
    }
  ],
  "word_count": 5,
  "character_count": 30
}
```

The JSON you get as a response depends on the question you asked and, again, contains nested dictionaries and lists. Lines 103–109 use small statements to pick off the translated text `"¿Dónde está el baño más cercano? "`. The boxes around portions of the JSON and the line numbers in each box correspond to the statements in lines 103–109. The statements operate as follows:

- Line 103 gets the `'translations'` list:

  lick here to view code image

  ```
  translations_list = translated_text['translations']
  ```

  If method `translate`'s `text` argument has multiple strings, the list will have multiple entries. We passed only one string, so the list contains only one element.

- Line 106 gets `translations_list`'s only element:

  lick here to view code image

  ```
  first_translation = translations_list[0]
  ```

- Line 109 gets the `'translation'` key's value, which is the translated text:

  lick here to view code image

  ```
  translation = first_translation['translation']
  ```

- Line 111 returns the translated string.

C

Lines 103–109 could be replaced with the more concise statement

lick here to view code image

```
return translated_text['translations'][0]['translation']
```

but again, we prefer the separate simpler statements.

Function `text_to_speech`

To access the Watson Text to Speech service, function `text_to_speech` (lines 113–122) creates a `TextToSpeechV1` object named `tts` (short for text-to-speech), passing as the argument the API key you set up earlier. The `with` statement opens the file specified by `file_name` and associates the file with the name `audio_file`. The mode `'wb'` opens the file for writing (`w`) in binary (`b`) format. We'll write into that file the contents of the audio returned by the Speech to Text service.

lick here to view code image

```
113 def text_to_speech(text_to_speak, voice_to_use,   file_name):
114     """Use Watson Text to Speech to convert   text to specified voice
115        and save to a WAV file."""
116     # create Text to Speech client
117     tts =   TextToSpeechV1(iam_apikey=keys.text_to_speech_key)
118
119     # open file and write the synthesized audio content   into the fi
120     with open(file_name, 'wb') as audio_file:
121         audio_file.write(tts.synthesize(text_to_speak,
122             accept='audio/wav', voice=voice_to_use).get_result().cont
123
```

Lines 121–122 call two methods. First, we invoke the Speech to Text service by calling the `TextToSpeechV1` object's **`synthesize` method**, passing three arguments:

- `text_to_speak` is the string to speak.
- the keyword argument `accept` is the media type indicating the audio format the Speech to Text service should return—again, `'audio/wav'` indicates an audio file in WAV format.
- the keyword argument `voice` is one of the Speech to Text service's predefined voices. In this app, we'll use `'en-US_AllisonVoice'` to speak English text and

C

`'es-US_SofiaVoice'` to speak Spanish text. Watson provides many male and female voices across various languages.[8]

[8] or a complete list, see `ttps://www.ibm.com/watson/developercloud/text-to-peech/api/v1/python.html?python#get-voice`. Try experimenting with other voices.

Watson's `DetailedResponse` contains the spoken text audio file, accessible via `get_result`. We access the returned file's `content` attribute to get the bytes of the audio and pass them to the `audio_file` object's `write` method to output the bytes to a `.wav` file.

Function `record_audio`

The `pyaudio` module enables you to record audio from the microphone. The function `record_audio` (lines 124–154) defines several constants (lines 126–130) used to configure the stream of audio information coming from your computer's microphone. We used the settings from the `pyaudio` module's online documentation:

- `FRAME_RATE`—`44100` frames-per-second represents 44.1 kHz, which is common for CD-quality audio.
- `CHUNK`—`1024` is the number of frames streamed into the program at a time.
- `FORMAT`—`pyaudio.paInt16` is the size of each frame (in this case, 16-bit or 2-byte integers).
- `CHANNELS`—`2` is the number of samples per frame.
- `SECONDS`—`5` is the number of seconds for which we'll record audio in this app.

lick here to view code image

```
124 def record_audio(file_name):
125     """Use pyaudio to record 5 seconds of   audio to a WAV file."""
126     FRAME_RATE = 44100  # number of frames per second
127     CHUNK = 1024  # number of frames read at a time
128     FORMAT = pyaudio.paInt16    # each frame is a 16-bit (2-byte) int
129     CHANNELS = 2  # 2 samples per frame
130     SECONDS = 5  # total recording time
131
```

```
132     recorder    = pyaudio.PyAudio()  # opens/closes audio streams
133
134     # configure and open audio stream for recording   (input=True)
135     audio_stream    = recorder.open(format=FORMAT, channels=CHANNELS,
136         rate=FRAME_RATE, input=True,   frames_per_buffer=CHUNK)
137     audio_frames    = []  # stores raw bytes of mic input
138     print('Recording 5 seconds of audio')
139
140     # read 5 seconds of audio in CHUNK-sized pieces
141     for i in range(0, int(FRAME_RATE * SECONDS / CHUNK)):
142         audio_frames.append(audio_stream.read(CHUNK))
143
144     print('Recording complete')
145     audio_stream.stop_stream()    # stop recording
146     audio_stream.close()
147     recorder.terminate()    # release underlying resources used by Py
148
149     # save audio_frames to a WAV file
150     with wave.open(file_name, 'wb')   as output_file:
151         output_file.setnchannels(CHANNELS)
152         output_file.setsampwidth(recorder.get_sample_size(FORMAT))
153         output_file.setframerate(FRAME_RATE)
154         output_file.writeframes(b''.join(audio_frames))
155
```

ine 132 creates the **`PyAudio`** object from which we'll obtain the input stream to record audio from the microphone. Lines 135–136 use the `PyAudio` object's **`open` method** to open the input stream, using the constants `FORMAT`, `CHANNELS`, `FRAME_RATE` and `CHUNK` to configure the stream. Setting the `input` keyword argument to `True` indicates that the stream will be used to *receive* audio input. The open method returns a `pyaudio` **`Stream`** object for interacting with the stream.

Lines 141–142 use the `Stream` object's **`read` method** to get `1024` (that is, `CHUNK`) frames at a time from the input stream, which we then append to the `audio_frames` list. To determine the total number of loop iterations required to produce 5 seconds of audio using `CHUNK` frames at a time, we multiply the `FRAME_RATE` by `SECONDS`, then divide the result by `CHUNK`. Once reading is complete, line 145 calls the `Stream` object's **`stop_stream` method** to terminate recording, line 146 calls the `Stream` object's **`close` method** to close the `Stream`, and line 147 calls the `PyAudio` object's **`terminate` method** to release the underlying audio resources that were being used to manage the audio stream.

The `with` statement in lines 150–154 uses the `wave` module's `open` function to open the WAV file specified by `file_name` for writing in binary format (`'wb'`). Lines 151–153 configure the WAV file's number of channels, sample width (obtained from the

L

`PyAudio` object's **`get_sample_size` method**) and frame rate. Then line 154 writes the audio content to the file. The expression `b''.join(audio_frames)` concatenates all the frames' bytes into a **byte string**. Prepending a string with `b` indicates that it's a string of bytes rather than a string of characters.

Function `play_audio`

To play the audio files returned by Watson's Text to Speech service, we use features of the `pydub` and `pydub.playback` modules. First, from the `pydub` module, line 158 uses the **`AudioSegment` class's `from_wav` method** to load a WAV file. The method returns a new `AudioSegment` object representing the audio file. To play the `AudioSegment`, line 159 calls the `pydub.playback` module's **`play` function**, passing the `AudioSegment` as an argument.

lick here to view code image

```
156 def play_audio(file_name):
157     """Use the pydub module (pip install   pydub) to play a WAV file.
158     sound =   pydub.AudioSegment.from_wav(file_name)
159     pydub.playback.play(sound)
160
```

Executing the run_translator Function

We call the `run_translator` function when you execute `SimpleLanguage-Translator.py` as a script:

lick here to view code image

```
161 if __name__ == '__main__':
162     run_translator()
```

Hopefully, the fact that we took a divide-and-conquer approach on this substantial case study script made it manageable. Many of the steps matched up nicely with some key Watson services, enabling us to quickly create a powerful mashup application.

13.7 WATSON RESOURCES

IBM provides a wide range of developer resources to help you familiarize yourself with their services and begin using them to build applications.

C

atson Services Documentation

The Watson Services documentation is at:

ttps://console.bluemix.net/developer/watson/documentation

For each service, there are documentation and API reference links. Each service's documentation typically includes some or all of the following:

- a getting started tutorial.
- a video overview of the service.
- a link to a service demo.
- links to more specific how-to and tutorial documents.
- sample apps.
- additional resources, such as more advanced tutorials, videos, blog posts and more.

Each service's API reference shows all the details of interacting with the service using any of several languages, including Python. Click the **Python** tab to see the Python-specific documentation and corresponding code samples for the Watson Developer Cloud Python SDK. The API reference explains all the options for invoking a given service, the kinds of responses it can return, sample responses, and more.

Watson SDKs

We used the Watson Developer Cloud Python SDK to develop this chapter's script. There are SDKs for many other languages and platforms. The complete list is located at:

ttps://console.bluemix.net/developer/watson/sdks-and-tools

Learning Resources

On the Learning Resources page

ttps://console.bluemix.net/developer/watson/learning-resources

you'll find links to:

- Blog posts on Watson features and how Watson and AI are being used in industry.

- Watson's GitHub repository (developer tools, SDKs and sample code).
- The Watson YouTube channel (discussed below).
- Code patterns, which IBM refers to as "roadmaps for solving complex programming challenges." Some are implemented in Python, but you may still find the other code patterns helpful in designing and implementing your Python apps.

Watson Videos

The Watson YouTube channel

ttps://www.youtube.com/user/IBMWatsonSolutions/

contains hundreds of videos showing you how to use all aspects of Watson. There are also spotlight videos showing how Watson is being used.

IBM Redbooks

The following IBM Redbooks publications cover IBM Cloud and Watson services in detail, helping you develop your Watson skills.

- Essentials of Application Development on IBM Cloud: `ttp://www.redbooks.ibm.com/abstracts/sg248374.html`
- Building Cognitive Applications with IBM Watson Services: Volume 1 **Getting Started**: `ttp://www.redbooks.ibm.com/abstracts/sg248387.html`
- Building Cognitive Applications with IBM Watson Services: Volume 2 **Conversation** (now called Watson Assistant): `ttp://www.redbooks.ibm.com/abstracts/sg248394.html`
- Building Cognitive Applications with IBM Watson Services: Volume 3 **Visual Recognition**: `ttp://www.redbooks.ibm.com/abstracts/sg248393.html`
- Building Cognitive Applications with IBM Watson Services: Volume 4 **Natural Language Classifier**: `ttp://www.redbooks.ibm.com/abstracts/sg248391.html`
- Building Cognitive Applications with IBM Watson Services: Volume 5 **Language Translator**: `ttp://www.redbooks.ibm.com/abstracts/sg248392.html`

h

- Building Cognitive Applications with IBM Watson Services: Volume 6 **Speech to Text and Text to Speech**:
  `ttp://www.redbooks.ibm.com/abstracts/sg248388.html`

- Building Cognitive Applications with IBM Watson Services: Volume 7 **Natural Language Understanding**:
  `ttp://www.redbooks.ibm.com/abstracts/sg248398.html`

13.8 WRAP-UP

In this chapter, we introduced IBM's Watson cognitive-computing platform and overviewed its broad range of services. You saw that Watson offers intriguing capabilities that you can integrate into your applications. IBM encourages learning and experimentation via its free Lite tiers. To take advantage of that, you set up an IBM Cloud account. You tried Watson demos to experiment with various services, such as natural language translation, speech-to-text, text-to-speech, natural language understanding, chatbots, analyzing text for tone and visual object recognition in images and video.

You installed the Watson Developer Cloud Python SDK for programmatic access to Watson services from your Python code. In the traveler's companion translation app, we mashed up several Watson services to enable English-only and Spanish-only speakers to communicate easily with one another verbally. We transcribed English and Spanish audio recordings to text, translated the text to the other language, then synthesized English and Spanish audio from the translated text. Finally, we discussed various Watson resources, including documentation, blogs, the Watson GitHub repository, the Watson YouTube channel, code patterns implemented in Python (and other languages) and IBM Redbooks.

h

14. Machine Learning: Classification, Regression and Clustering

Objectives

In this chapter you'll:

- Use scikit-learn with popular datasets to perform machine learning studies.
- Use Seaborn and Matplotlib to visualize and explore data.
- Perform supervised machine learning with k-nearest neighbors classification and linear regression.
- Perform multi-classification with Digits dataset.
- Divide a dataset into training, test and validation sets.
- Tune model hyperparameters with k-fold cross-validation.
- Measure model performance.
- Display a confusion matrix showing classification prediction hits and misses.
- Perform multiple linear regression with the California Housing dataset.
- Perform dimensionality reduction with PCA and t-SNE on the Iris and Digits datasets to prepare them for two-dimensional visualizations.
- Perform unsupervised machine learning with k-means clustering and the Iris dataset.

Outline

14.1 INTRODUCTION TO MACHINE LEARNING

In this chapter and the next, we'll present machine learning—one of the most exciting and promising subfields of artificial intelligence. You'll see how to quickly solve challenging and intriguing problems that novices and most experienced programmers probably would not have attempted just a few years ago. Machine learning is a big, complex topic that raises lots of subtle issues. Our goal here is to give you a friendly, hands-on introduction to a few of the simpler machine-learning techniques.

What Is Machine Learning?

Can we really make our machines (that is, our computers) learn? In this and the next

hapter, we'll show exactly how that magic happens. What's the "secret sauce" of this new application-development style? It's data and lots of it. Rather than programming expertise into our applications, we program them to learn from data. We'll present many Python-based code examples that build working machine-learning- models then use them to make remarkably accurate predictions.

Prediction

Wouldn't it be fantastic if you could improve weather forecasting to save lives, minimize injuries and property damage? What if we could improve cancer diagnoses and treatment regimens to save lives, or improve business forecasts to maximize profits and secure people's jobs? What about detecting fraudulent credit-card purchases and insurance claims? How about predicting customer "churn," what prices houses are likely to sell for, ticket sales of new movies, and anticipated revenue of new products and services? How about predicting the best strategies for coaches and players to use to win more games and championships? All of these kinds of predictions are happening today with machine learning.

Machine Learning Applications

Here's a table of some popular machine-learning applications:

Machine learning applications		
Anomaly detection		
Chatbots	Detecting objects in scenes	
Classifying emails as spam or not spam	Detecting patterns in data	Recommender systems ("people who bought this product also bought ")
Classifying news articles as sports, financial, politics, etc.	Diagnostic medicine	Self-Driving cars (more generally, autonomous vehicles)
Computer vision and image	Facial recognition	Sentiment analysis (like classifying movie reviews as
	Insurance fraud detection	

c

classification	Intrusion detection in computer networks	positive, negative or neutral)
Credit-card fraud detection	Handwriting recognition	Spam filtering
Customer churn prediction	Marketing: Divide customers into clusters	Time series predictions like stock-price forecasting and weather forecasting
Data compression	Natural language translation (English to Spanish, French to Japanese, etc.)	Voice recognition
Data exploration		
Data mining social media (like Facebook, Twitter, LinkedIn)	Predict mortgage loan defaults	

14.1.1 Scikit-Learn

We'll use the popular *scikit-learn machine learning library*. Scikit-learn, also called sklearn, conveniently packages the most effective machine-learning algorithms as *estimators*. Each is encapsulated, so you don't see the intricate details and heavy mathematics of how these algorithms work. You should feel comfortable with this—you drive your car without knowing the intricate details of how engines, transmissions, braking systems and steering systems work. Think about this the next time you step into an elevator and select your destination floor, or turn on your television and select the program you'd like to watch. Do you really understand the internal workings of your smart phone's hardware and software?

With scikit-learn and a small amount of Python code, you'll create powerful models quickly for analyzing data, extracting insights from the data and most importantly making predictions. You'll use scikit-learn to *train* each model on a subset of your data, then *test* each model on the rest to see how well your model works. Once your models are trained, you'll put them to work making predictions based on data they have not seen. You'll often be amazed at the results. All of a sudden your computer that you've used mostly on rote chores will take on characteristics of intelligence.

Scikit-learn has tools that automate training and testing your models. Although you can

pecify parameters to customize the models and possibly improve their performance, in this chapter, we'll typically use the models' *default parameters*, yet still obtain impressive results. There also are tools like auto-sklearn (`ttps://automl.github.io/auto-sklearn`), which automates many of the tasks you perform with scikit-learn.

Which Scikit-Learn Estimator Should You Choose for Your Project

It's difficult to know in advance which model(s) will perform best on your data, so you typically try many models and pick the one that performs best. As you'll see, scikit-learn makes this convenient for you. A popular approach is to run many models and pick the best one(s). How do we evaluate which model performed best?

You'll want to experiment with lots of different models on different kinds of datasets. You'll rarely get to know the details of the complex mathematical algorithms in the sklearn estimators, but with experience, you'll become familiar with which algorithms may be best for particular types of datasets and problems. Even with that experience, it's unlikely that you'll be able to intuit the best model for each new dataset. So scikit-learn makes it easy for you to "try 'em all." It takes at most a few lines of code for you to create and use each model. The models report their performance so you can compare the results and pick the model(s) with the best performance.

14.1.2 Types of Machine Learning

We'll present the two main types of machine learning—*supervised machine learning*, which works with *labeled data*, and *unsupervised machine learning*, which works with *unlabeled data*.

If, for example, you're developing a computer vision application to recognize dogs and cats, you'll train your model on lots of dog photos labeled "dog" and cat photos labeled "cat." If your model is effective, when you put it to work processing unlabeled photos it will recognize dogs and cats it has never seen before. The more photos you train with, the greater the chance that your model will accurately predict which new photos are dogs and which are cats. In this era of big data and massive, economical computer power, you should be able to build some pretty accurate models with the techniques you're about to see.

How can looking at unlabeled data be useful? Online booksellers sell lots of books. They record enormous amounts of (unlabeled) book purchase transaction data. They noticed early on that people who bought certain books were likely to purchase other books on the same or similar topics. That led to their *recommendation systems*. Now, when you

rowse a bookseller site for a particular book, you're likely to see recommendations like, "people who bought this book also bought these other books." Recommendation systems are big business today, helping to maximize product sales of all kinds.

Supervised Machine Learning

Supervised machine learning falls into two categories—*classification* and *regression*. You train machine-learning models on datasets that consist of rows and columns. Each row represents a data *sample*. Each column represents a *feature* of that sample. In supervised machine learning, each sample has an associated label called a *target* (like "dog" or "cat"). This is the value you're trying to predict for new data that you present to your models.

Datasets

You'll work with some "toy" datasets, each with a small number of samples with a limited number of features. You'll also work with several richly featured real-world datasets, one containing a few thousand samples and one containing tens of thousands of samples. In the world of big data, datasets commonly have, millions and billions of samples, or even more.

There's an enormous number of free and open datasets available for data science studies. Libraries like scikit-learn package up popular datasets for you to experiment with and provide mechanisms for loading datasets from various repositories (such as `openml.org`). Governments, businesses and other organizations worldwide offer datasets on a vast range of subjects. You'll work with several popular free datasets, using a variety of machine learning techniques.

Classification

We'll use one of the simplest classification algorithms, *k-nearest neighbors*, to analyze the Digits dataset bundled with scikit-learn. Classification algorithms predict the discrete classes (categories) to which samples belong. Binary classification uses two classes, such as "spam" or "not spam" in an email classification application. Multi-classification uses more than two classes, such as the 10 classes, 0 through 9, in the Digits dataset. A classification scheme looking at movie descriptions might try to classify them as "action," "adventure," "fantasy," "romance," "history" and the like.

Regression

Regression models predict a *continuous output*, such as the predicted temperature output in the weather *time series* analysis from hapter 10's Intro to Data Science

ection. In this chapter, we'll revisit that *simple linear regression* example, this time implementing it using scikit-learn's `LinearRegression` estimator. Next, we use a `LinearRegression` estimator to perform *multiple linear regression* with the California Housing dataset that's bundled with scikit-learn. We'll predict the median house value of a U. S. census block of homes, considering eight features per block, such as the average number of rooms, median house age, average number of bedrooms and median income. The `LinearRegression` estimator, by default, uses *all* the numerical features in a dataset to make more sophisticated predictions than you can with a single-feature simple linear regression.

Unsupervised Machine Learning

Next, we'll introduce unsupervised machine learning with *clustering* algorithms. We'll use *dimensionality reduction* (with scikit-learn's `TSNE` estimator) to *compress* the Digits dataset's 64 features down to two for visualization purposes. This will enable us to see how nicely the Digits data "cluster up." This dataset contains handwritten digits like those the post office's computers must recognize to route each letter to its designated zip code. This is a challenging computer-vision problem, given that each person's handwriting is unique. Yet, we'll build this clustering model with just a few lines of code and achieve impressive results. And we'll do this without having to understand the inner workings of the clustering algorithm. This is the beauty of object-based programming. We'll see this kind of convenient object-based programming again in the next chapter, where we'll build powerful deep learning models using the open source Keras library.

K-Means Clustering and the Iris Dataset

We'll present the simplest unsupervised machine-learning algorithm, *k-means clustering*, and use it on the Iris dataset that's also bundled with scikit-learn. We'll use dimensionality reduction (with scikit-learn's `PCA` estimator) to compress the Iris dataset's four features to two for visualization purposes. We'll show the clustering of the three *Iris* species in the dataset and graph each cluster's *centroid,* which is the cluster's center point. Finally, we'll run multiple clustering estimators to compare their ability to divide the Iris dataset's samples effectively into three clusters.

You normally specify the desired number of clusters, *k*. K-means works through the data trying to divide it into that many clusters. As with many machine learning algorithms, k-means is *iterative* and gradually zeros in on the clusters to match the number you specify.

K-means clustering can find similarities in unlabeled data. This can ultimately help

s

ith assigning labels to that data so that supervised learning estimators can then process it. Given that it's tedious and error-prone for humans to have to assign labels to unlabeled data, and given that the vast majority of the world's data is unlabeled, unsupervised machine learning is an important tool.

Big Data and Big Computer Processing Power

The amount of data that's available today is already enormous and continues to grow exponentially. The data produced in the world in the last few years equals the amount produced up to that point since the dawn of civilization. We commonly talk about big data, but "big" may not be a strong enough term to describe truly how huge data is getting.

People used to say "I'm drowning in data and I don't know what to do with it." With machine learning, we now say, "Flood me with big data so I can use machine-learning technology to extract insights and make predictions from it."

This is occurring at a time when computing power is *exploding* and computer memory and secondary storage are *exploding* in capacity while costs dramatically decline. All of this enables us to think differently about the solution approaches. We now can program computers to *learn* from data, and lots of it. It's now all about predicting from data.

14.1.3 Datasets Bundled with Scikit-Learn

The following table lists scikit-learn's bundled datasets. [1] It also provides capabilities for loading datasets from other sources, such as the 20,000+ datasets available at `openml.org`.

[1] `ttp://scikit-learn.org/stable/datasets/index.html`.

Datasets bundled with scikit-learn

"Toy" datasets	*Real-world datasets*
Boston house prices	Olivetti faces
Iris plants	20 newsgroups text
Diabetes	Labeled Faces in the Wild face recognition

w

Optical recognition of handwritten digits	Forest cover types
Linnerrud	RCV1
Wine recognition	Kddcup 99
Breast cancer Wisconsin (diagnostic)	California Housing

14.1.4 Steps in a Typical Data Science Study

We'll perform the steps of a typical machine-learning case study, including:

- loading the dataset
- exploring the data with pandas and visualizations
- transforming your data (converting non-numeric data to numeric data because scikit-learn requires numeric data; in the chapter, we use datasets that are "ready to go," but we'll discuss the issue again in the "Deep Learning" chapter)
- splitting the data for training and testing
- creating the model
- training and testing the model
- tuning the model and evaluating its accuracy
- making predictions on live data that the model hasn't seen before.

In the "Array-Oriented Programming with NumPy" and "Strings: A Deeper Look" chapters' Intro to Data Science sections, we discussed using pandas to deal with missing and erroneous values. These are important steps in cleaning your data before using it for machine learning.

14.2 CASE STUDY: CLASSIFICATION WITH K-NEAREST NEIGHBORS AND THE DIGITS DATASET, PART 1

o process mail efficiently and route each letter to the correct destination, postal service computers must be able to scan handwritten names, addresses and zip codes and recognize the letters and digits. As you'll see in this chapter, powerful libraries like scikit-learn enable even novice programmers to make such machine-learning problems manageable. In the next chapter, we'll use even more powerful computer-vision capabilities when we present the deep learning technology of convolutional neural networks.

Classification Problems

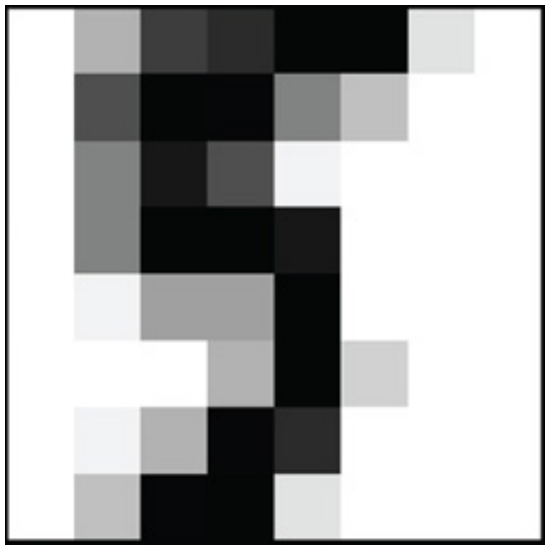
In this section, we'll look at **classification** in supervised machine learning, which attempts to predict the distinct class [2] to which a sample belongs. For example, if you have images of dogs and images of cats, you can classify each image as a "dog" or a "cat." This is a **binary classification problem** because there are *two* classes.

[2] Note that the term class in this case means category, not the Python concept of a class.

We'll use the **Digits dataset** [3] bundled with scikit-learn, which consists of 8-by-8 pixel images representing 1797 hand-written digits (0 through 9). Our goal is to predict which digit an image represents. Since there are 10 possible digits (the classes), this is a **multi-classification problem**. You train a classification model using **labeled data** —we know in advance each digit's class. In this case study, we'll use one of the simplest machine-learning classification algorithms, *k-nearest neighbors (k-NN)*, to recognize handwritten digits.

[3] `ttp://scikit-learn.org/stable/datasets/index.html#optical-ecognition-of-handwritten-digits-dataset`.

The following low-resolution digit visualization of a 5 was produced with Matplotlib from one digit's 8-by-8 pixel raw data. We'll show how to display images like this with Matplotlib momentarily:



Researchers created the images in this dataset from the MNIST database's tens of thousands of 32-by-32 pixel images that were produced in the early 1990s. At today's high-definition camera and scanner resolutions, such images can be captured with much higher resolutions.

Our Approach

We'll cover this case study over two sections. In this section, we'll begin with the basic steps of a machine learning case study:

- Decide the data from which to train a model.
- Load and explore the data.
- Split the data for training and testing.
- Select and build the model.
- Train the model.
- Make predictions.

As you'll see, in scikit-learn each of these steps requires at most a few lines of code. In the next section, we'll

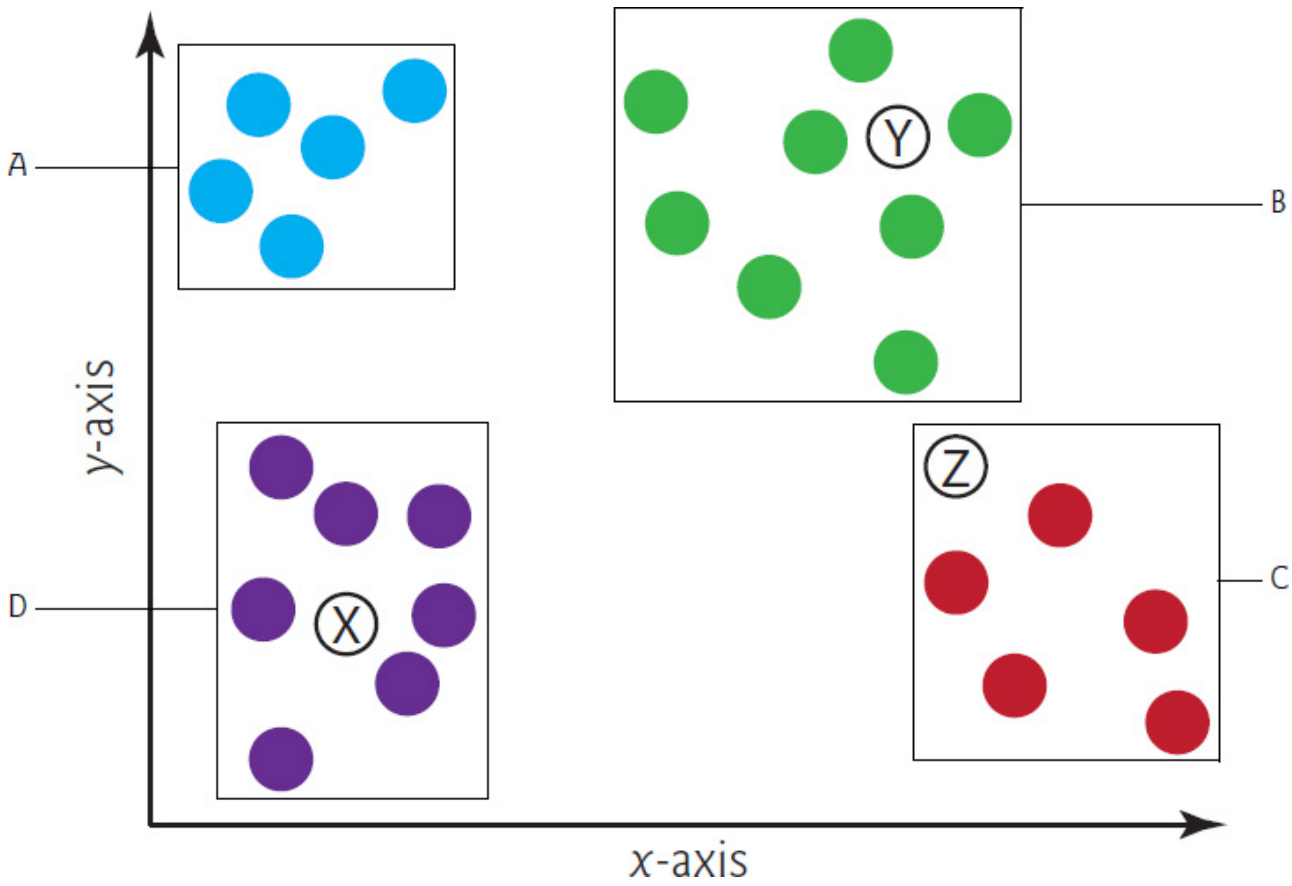
- Evaluate the results.
- Tune the model.
- Run several classification models to choose the best one(s).

We'll visualize the data using Matplotlib and Seaborn, so launch IPython with Matplotlib support:

```
ipython --matplotlib
```

14.2.1 k-Nearest Neighbors Algorithm

Scikit-learn supports many **classification algorithms**, including the simplest—**k-nearest neighbors (k-NN)**. This algorithm attempts to predict a test sample's class by looking at the *k* training samples that are nearest (in distance) to the test sample. For example, consider the following diagram in which the filled dots represent four sample classes—A, B, C and D. For this discussion, we'll use these letters as the class names:



We want to predict the classes to which the new samples **X**, **Y** and **Z** belong. Let's assume we'd like to make these predictions using each sample's *three* nearest neighbors —*three* is *k* in the k-nearest neighbors algorithm:

- Sample **X**'s three nearest neighbors are all class D dots, so we'd predict that **X**'s class is D.
- Sample **Y**'s three nearest neighbors are all class B dots, so we'd predict that **Y**'s class is B.

- For **Z**, the choice is not as clear, because it appears *between* the B *and* C dots. Of the three nearest neighbors, one is class B and two are class C. In the k-nearest neighbors algorithm, the class with the most "votes" wins. So, based on two C votes to one B vote, we'd predict that **Z**'s class is C. Picking an odd *k* value in the kNN algorithm avoids ties by ensuring there's never an equal number of votes.

Hyperparameters and Hyperparameter Tuning

In machine learning, a **model** implements a machine-learning algorithm. In scikit-learn, models are called **estimators**. There are two parameter types in machine learning:

- those the estimator calculates as it learns from the data you provide and
- those you specify in advance when you create the scikit-learn estimator object that represents the model.

The parameters specified in advance are called **hyperparameters**.

In the k-nearest neighbors algorithm, *k* is a hyperparameter. For simplicity, we'll use scikit-learn's *default* hyperparameter values. In real-world machine-learning studies, you'll want to experiment with different values of *k* to produce the best possible models for your studies. This process is called **hyperparameter tuning**. Later we'll use hyperparameter tuning to choose the value of *k* that enables the k-nearest neighbors algorithm to make the best predictions for the Digits dataset. Scikit-learn also has *automated* hyperparameter tuning capabilities.

14.2.2 Loading the Dataset

The `load_digits` function from the `sklearn.datasets` **module** returns a scikit-learn `Bunch` object containing the digits data and information *about* the Digits dataset (called **metadata**):

lick here to view code image

```
In [1]: from sklearn.datasets import load_digits

In [2]: digits = load_digits()
```

`Bunch` is a subclass of `dict` that has additional attributes for interacting with the dataset.

C

Displaying the Description

The Digits dataset bundled with scikit-learn is a subset of the **UCI (University of California Irvine) ML hand-written digits dataset** at:

ttp://archive.ics.uci.edu/ml/datasets/Optical+Recognition+of+Handwritten+Digits

The original UCI dataset contains 5620 samples—3823 for training and 1797 for testing. The version of the dataset bundled with scikit-learn contains only the *1797 testing samples*. A `Bunch`'s **`DESCR` attribute** contains a description of the dataset. According to the Digits dataset's description [4] , each sample has `64` features (as specified by `Number of Attributes`) that represent an 8-by-8 image with pixel values in the range `0–16` (specified by `Attribute Information`). This dataset has *no missing values* (as specified by `Missing Attribute Values`). The 64 features may seem like a lot, but real-world datasets can sometimes have hundreds, thousands or even millions of features.

[4] e highlighted some key information in bold.

lick here to view code image

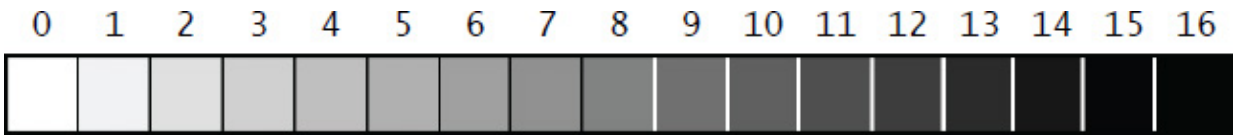
```
 n [3]: print(digits.DESCR)
.. _digits_dataset:

Optical recognition of handwritten digits dataset
--------------------------------------------------

**Data Set Characteristics:**

    :Number of Instances: 5620
    :Number of Attributes: 64
    :Attribute Information: 8x8 image of   integer pixels in the range
        0..16.
    :Missing Attribute Values: None
    :Creator: E. Alpaydin (alpaydin '@'   boun.edu.tr)
    :Date: July; 1998

This is a copy of the test set of the UCI ML hand-written digits   dataset:
http://archive.ics.uci.edu/ml/datasets/
      Optical+Recognition+of+Handwritten+Digits-

The data set contains images of hand-written digits: 10   classes where
each class refers to a digit.

Preprocessing programs made available by NIST were used to   extract
normalized bitmaps of handwritten digits from a preprinted   form. From a
total of 43 people, 30 contributed to the training set and   different 13
to the test set. 32x32 bitmaps are divided into   nonoverlapping blocks of
```

```
4x4 and the number of on pixels are counted in each block.  This generates
an input matrix of 8x8 where each element is an integer in  the range
0..16. This reduces dimensionality and gives invariance to  small
distortions.

For info on NIST preprocessing routines, see M. D. Garris,  J. L. Blue, G
T. Candela, D. L. Dimmick, J. Geist, P. J. Grother, S. A.  Janet, and C.
L. Wilson, NIST Form-Based Handprint Recognition System,  NISTIR 5469,
1994.

.. topic:: References

  - C. Kaynak (1995) Methods of Combining Multiple  Classifiers and Their
    Applications to Handwritten Digit Recognition, MSc  Thesis, Institute
    of Graduate Studies in Science and Engineering,  Bogazici University.
  - E. Alpaydin, C. Kaynak (1998) Cascading Classifiers,  Kybernetika.
  - Ken Tang and Ponnuthurai N. Suganthan and Xi Yao and A.  Kai Qin.
    Linear dimensionality reduction using relevance  weighted LDA. School
    of Electrical and Electronic Engineering Nanyang  Technological
    University. 2005.
  - Claudio Gentile. A New Approximate Maximal Margin  Classification
    Algorithm. NIPS. 2000.
```

hecking the Sample and Target Sizes

The `Bunch` object's **`data`** and **`target` attributes** are NumPy `array`s:

- The `data` array contains the 1797 samples (the digit images), each with 64 features, having values in the range 0–16, representing *pixel intensities*. With Matplotlib, we'll visualize these intensities in grayscale shades from white (0) to black (16):



- The `target` array contains the images' labels—that is, the classes indicating which digit each image represents. The array is called `target` because, when you make predictions, you're aiming to "hit the target" values. To see labels of samples throughout the dataset, let's display the `target` values of every 100th sample:

lick here to view code image

```
In [4]: digits.target[::100]
Out[4]: array([0, 4, 1, 7, 4, 8, 2, 2, 4, 4, 1, 9, 7, 3, 2,   1, 2, 5])
```

C

We can confirm the number of samples and features (per sample) by looking at the `data` array's `shape` attribute, which shows that there are 1797 rows (samples) and 64 columns (features):

lick here to view code image

```
In [5]: digits.data.shape
Out[5]: (1797, 64)
```

You can confirm that the number of target values matches the number of samples by looking at the `target` array's `shape`:

lick here to view code image

```
In [6]: digits.target.shape
Out[6]: (1797,)
```

A Sample Digit Image

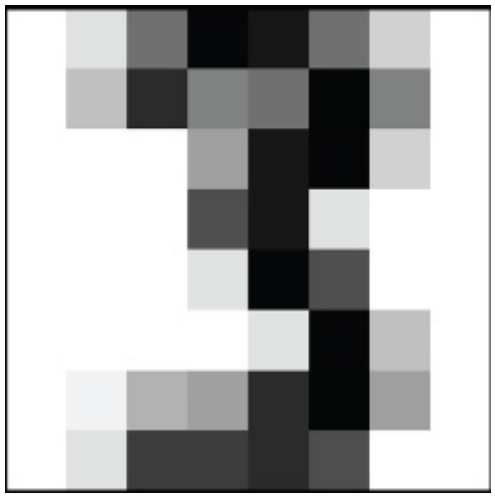
Each image is two-dimensional—it has a width and a height in pixels. The `Bunch` object returned by `load_digits` contains an `images` attribute—an array in which each element is a two-dimensional 8-by-8 array representing a digit image's pixel intensities. Though the original dataset represents each pixel as an integer value from 0–16, scikit-learn stores these values as *floating-point* values (NumPy type `float64`). For example, here's the two-dimensional array representing the sample image at index `13`:

lick here to view code image

```
In [7]: digits.images[13]
Out[7]:
array([[ 0.,  2.,  9., 15., 14.,  9.,  3.,  0.],
       [ 0.,  4., 13.,  8.,  9., 16.,  8.,  0.],
       [ 0.,  0.,  0.,  6., 14., 15.,  3.,  0.],
       [ 0.,  0.,  0., 11., 14.,  2.,  0.,  0.],
       [ 0.,  0.,  0.,  2., 15., 11.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  2., 15.,  4.,  0.],
       [ 0.,  1.,  5.,  6., 13., 16.,  6.,  0.],
       [ 0.,  2., 12., 12., 13., 11.,  0.,  0.]])
```

and here's the image represented by this two-dimensional array—we'll soon show the code for displaying this image:

C



Preparing the Data for Use with Scikit-Learn

Scikit-learn's machine-learning algorithms require samples to be stored in a *two-dimensional array* of *floating-point values* (or two-dimensional *array-like* collection, such as a list of lists or a pandas `DataFrame`**):**

- Each row represents one *sample*.
- Each column in a given row represents one *feature* for that sample.

To represent every sample as one row, multi-dimensional data like the two-dimensional image array shown in snippet `[7]` must be *flattened* into a one-dimensional array.

If you were working with a data containing **categorical features** (typically represented as strings, such as `'spam'` or `'not-spam'`), you'd also have to *preprocess* those features into numerical values—known as one-hot encoding, which we cover in the next chapter. Scikit-learn's **`sklearn.preprocessing`** module provides capabilities for converting categorical data to numeric data. The Digits dataset has no categorical features.

For your convenience, the `load_digits` function returns the preprocessed data ready for machine learning. The Digits dataset is numerical, so `load_digits` simply flattens each image's two-dimensional array into a one-dimensional array. For example, the 8-by-8 array `digits.images[13]` shown in snippet `[7]` corresponds to the 1-by-64 array `digits.data[13]` shown below:

lick here to view code image

```
In [8]: digits.data[13]
Out[8]:
array([ 0.,  2.,  9., 15., 14.,  9.,  3.,  0.,  0.,  4.,  13.,  8.,  9.,
```

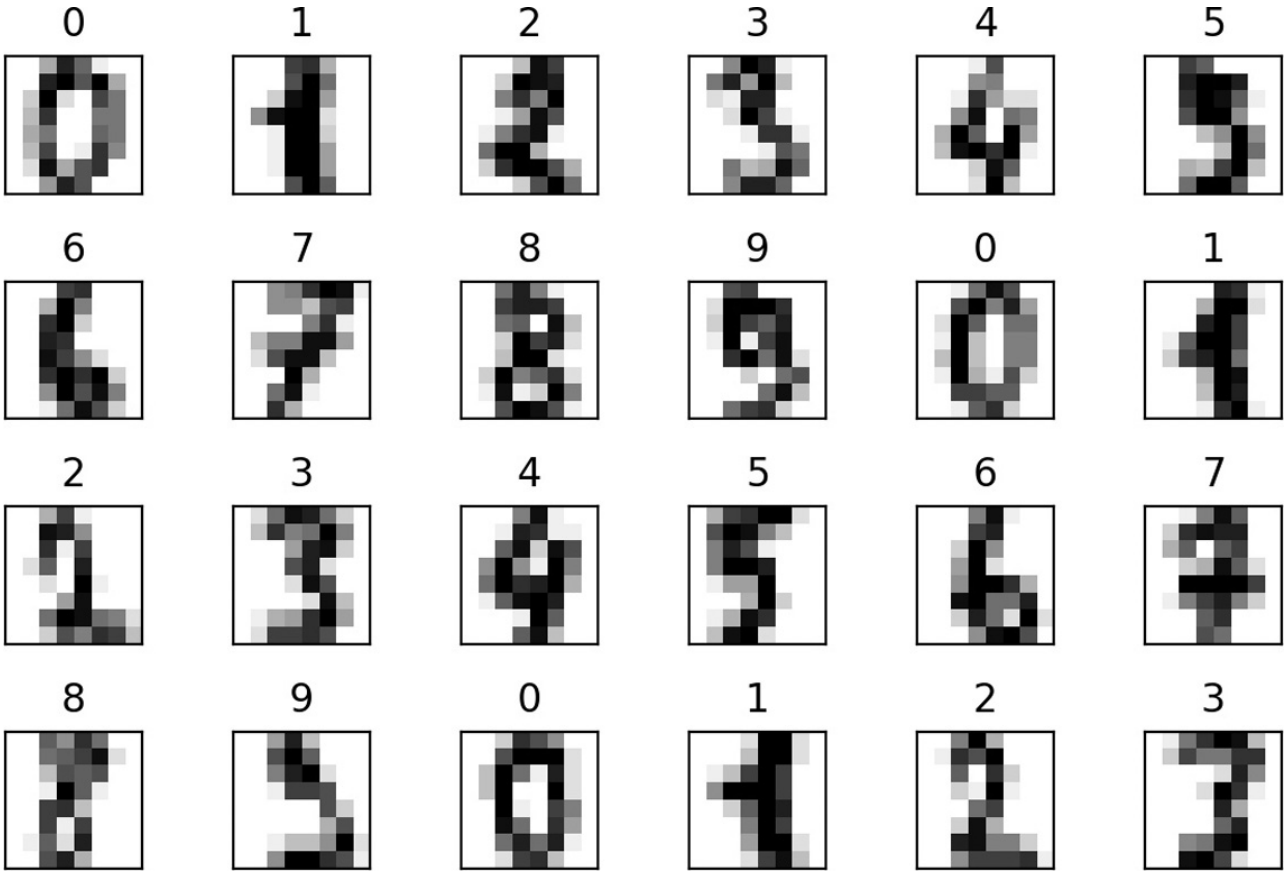
C

```
       16.,  8.,  0.,  0.,  0.,  0.,  6., 14., 15.,  3.,   0.,  0.,  0.,
        0., 11., 14.,  2.,  0.,  0.,  0.,  0.,  0.,  2.,  15., 11.,  0.,
        0.,  0.,  0.,  0.,  0.,  2., 15.,  4.,  0.,  0.,   1.,  5.,  6.,
       13., 16.,  6.,  0.,  0.,  2., 12., 12., 13., 11.,   0.,  0.])
```

In this one-dimensional array, the first eight elements are the two-dimensional array's row 0, the next eight elements are the two-dimensional array's row 1, and so on.

14.2.3 Visualizing the Data

You should always familiarize yourself with your data. This process is called **data exploration**. For the digit images, you can get a sense of what they look like by displaying them with the Matplotlib `implot` function. The following image shows the dataset's first 24 images. To see how difficult a problem handwritten digit recognition is, consider the *variations* among the images of the 3s in the first, third and fourth rows, and look at the images of the 2s in the first, third and fourth rows.



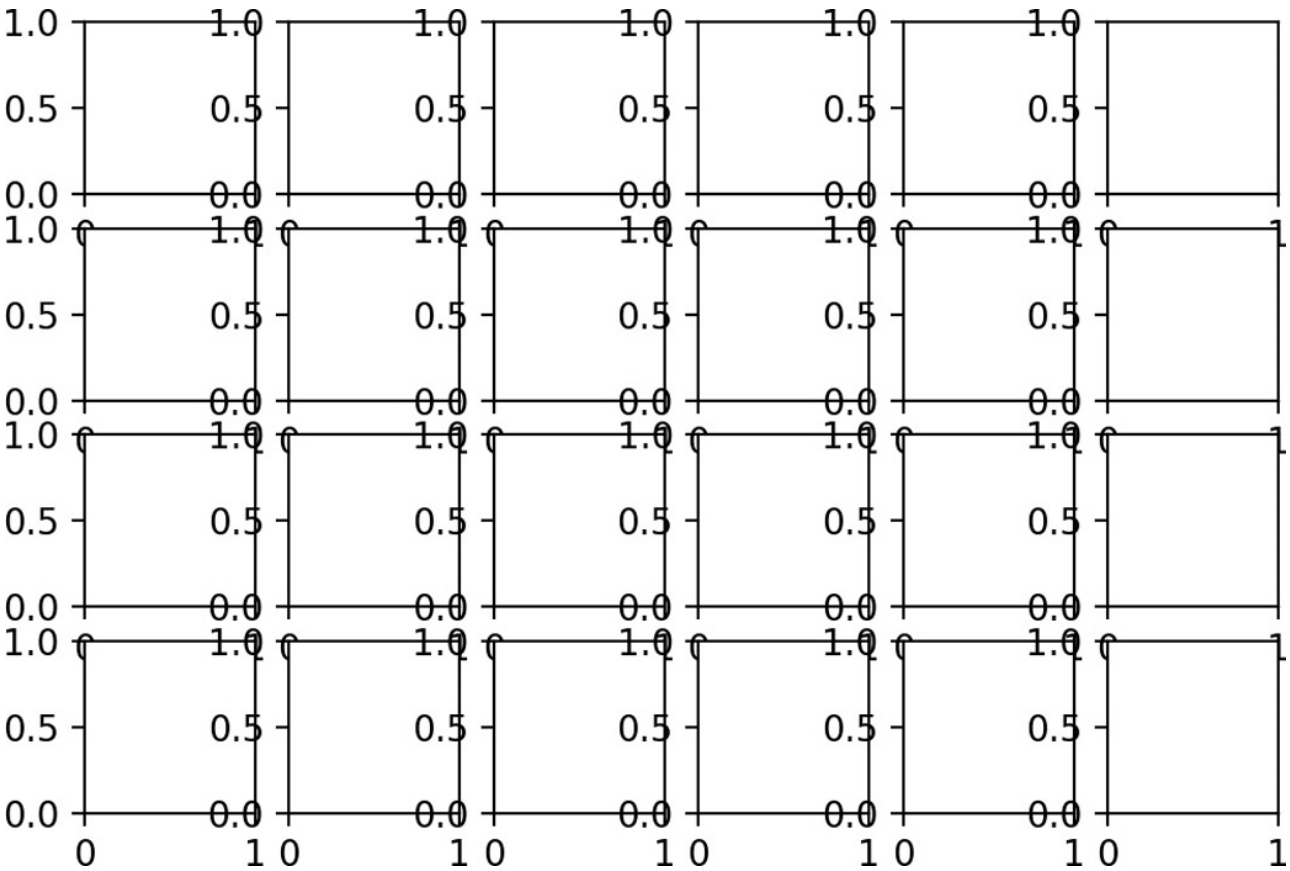
Creating the Diagram

Let's look at the code that displayed these 24 digits. The following call to function `subplots` creates a 6-by-4 inch `Figure` (specified by the `figsize(6, 4)` keyword argument) containing 24 subplots arranged in `4` rows (`nrows=4`) and `6` columns (`ncols=6`). Each subplot has its own `Axes` object, which we'll use to display one digit image:

lick here to view code image

```
In [9]: import matplotlib.pyplot   as plt

In [10]: figure, axes = plt.subplots(nrows=4, ncols=6, figsize=(6, 4))
```

Function `subplots` returns the `Axes` objects in a two-dimensional NumPy array. Initially, the `Figure` appears as shown below with labels (which we'll remove) on every subplot's *x*- and *y*-axes:



Displaying Each Image and Removing the Axes Labels

Next, use a `for` statement with the built-in `zip` function to iterate in parallel through the 24 `Axes` objects, the first 24 images in `digits.images` and the first 24 values in `digits.target`:

lick here to view code image

```
In [11]: for item in zip(axes.ravel(), digits.images, digits.target):
    ...:     axes, image, target = item
    ...:     axes.imshow(image, cmap=plt.cm.gray_r)
    ...:     axes.set_xticks([])  #   remove x-axis tick marks
    ...:     axes.set_yticks([])  #   remove y-axis tick marks
    ...:     axes.set_title(target)
    ...: plt.tight_layout()
```

C

```
        ...:
        ...:
```

Recall that NumPy array method `ravel` creates a *one-dimensional view* of a multidimensional array. Also, recall that `zip` produces tuples containing elements from the same index in each of `zip`'s arguments and that the argument with the fewest elements determines how many tuples `zip` returns.

Each iteration of the loop:

- Unpacks one tuple from the `zip`ped items into three variables representing the `Axes` object, image and target value.
- Calls the `Axes` object's `imshow` method to display one image. The keyword argument `cmap=plt.cm.gray_r` determines the colors displayed in the image. The value `plt.cm.gray_r` is a **color map**—a group of colors that are typically chosen to work well together. This particular color map enables us to display the image's pixels in grayscale, with 0 as white, 16 as black and the values in between as gradually darkening shades of gray. For Matplotlib's color map names see `ttps://matplotlib.org/examples/color/colormaps_reference.html`. Each can be accessed through the `plt.cm` object or via a string, like `'gray_r'`.
- Calls the `Axes` object's `set_xticks` and `set_yticks` methods with empty lists to indicate that the *x*- and *y*-axes should not have tick marks.
- Calls the `Axes` object's `set_title` method to display the `target` value above the image—this shows the actual value that the image represents.

After the loop, we call `tight_layout` to remove the extra whitespace at the `Figure`'s top, right, bottom and left, so the rows and columns of digit images can fill more of the `Figure`.

14.2.4 Splitting the Data for Training and Testing

You typically train a machine-learning model with a subset of a dataset. Typically, the more data you have for training, the better you can train the model. It's important to set aside a portion of your data for testing, so you can evaluate a model's performance using data that the model has not yet seen. Once you're confident that the model is performing well, you can use it to make predictions using new data it hasn't seen.

h

We first break the data into a **training set** and a **testing set** to prepare to train and test the model. The function `train_test_split` from the `sklearn.model_selection` module *shuffles* the data to randomize it, then splits the samples in the `data` array and the target values in the `target` array into training and testing sets. This helps ensure that the training and testing sets have similar characteristics. The shuffling and splitting is performed conveniently for you by a `ShuffleSplit` object from the `sklearn.model_selection` module. Function `train_test_split` returns a tuple of four elements in which the first two are the *samples* split into training and testing sets, and the last two are the corresponding *target values* split into training and testing sets. By convention, uppercase `X` is used to represent the samples, and lowercase `y` is used to represent the target values:

lick here to view code image

```
In [12]: from   sklearn.model_selection import train_test_split

In [13]: X_train, X_test, y_train, y_test =   train_test_split(
   ...:     digits.data, digits.target, random_state=11)
   ...:
```

We assume the data has **balanced classes**—that is, the samples are divided evenly among the classes. This is the case for each of scikit-learn's bundled classification datasets. Unbalanced classes could lead to incorrect results.

In the "Functions" chapter, you saw how to *seed* a random-number generator for *reproducibility*. In machine-learning studies, this helps others confirm your results by working with the *same* randomly selected data. Function `train_test_split` provides the keyword argument `random_state` for *reproducibility*. When you run the code in the future with the *same* seed value, `train_test_split` will select the *same* data for the training set and the *same* data for the testing set. We chose the seed value (`11`) arbitrarily.

Training and Testing Set Sizes

Looking at `X_train`'s and `X_test`'s `shape`s, you can see that, *by default*, `train_test_split` reserves 75% of the data for training and 25% for testing:

lick here to view code image

```
In [14]: X_train.shape
Out[14]: (1347, 64)
```

C

```
In [15]: X_test.shape
Out[15]: (450, 64)
```

To specify *different* splits, you can set the sizes of the testing and training sets with the `train_test_split` function's keyword arguments `test_size` and `train_size`. Use floating-point values from 0.0 through 1.0 to specify the percentages of the data to use for each. You can use integer values to set the precise numbers of samples. If you specify one of these keyword arguments, the other is inferred. For example, the statement

lick here to view code image

```
X_train, X_test, y_train, y_test = train_test_split(
     digits.data, digits.target, random_state=11, test_size=0.20)
```

specifies that 20% of the data is for testing, so `train_size` is inferred to be `0.80`.

14.2.5 Creating the Model

The **`KNeighborsClassifier`** estimator (module **`sklearn.neighbors`**) implements the k-nearest neighbors algorithm. First, we create the `KNeighborsClassifier` estimator object:

lick here to view code image

```
In [16]: from sklearn.neighbors import KNeighborsClassifier

In [17]: knn = KNeighborsClassifier()
```

To create an estimator, you simply create an object. The internal details of how this object implements the k-nearest neighbors algorithm are hidden in the object. You'll simply call its methods. This is the essence of Python *object-based programming*.

14.2.6 Training the Model

Next, we invoke the `KNeighborsClassifier` object's **`fit` method**, which loads the sample training set (`X_train`) and target training set (`y_train`) into the estimator:

lick here to view code image

C

```
In [18]: knn.fit(X=X_train, y=y_train)
Out[18]:
KNeighborsClassifier(algorithm='auto', leaf_size=30,   metric='minkowski',
           metric_params=None, n_jobs=None, n_neighbors=5,   p=2,
           weights='uniform')
```

For most, scikit-learn estimators, the `fit` method loads the data into the estimator then uses that data to perform complex calculations behind the scenes that learn from the data and train the model. The `KNeighborsClassifier`'s `fit` method just loads the data into the estimator, because k-NN actually has no initial learning process. The estimator is said to be **lazy** because its work is performed only when you use it to make predictions. In this and the next chapter, you'll use lots of models that have significant training phases. In the real-world machine-learning applications, it can sometimes take minutes, hours, days or even months to train your models. We'll see in the next chapter, "Deep Learning," that special-purpose, high-performance hardware called GPUs and TPUs can significantly reduce model training time.

As shown in snippet `[18]`'s output, the `fit` method returns the estimator, so IPython displays its string representation, which includes the estimator's *default* settings. The `n_neighbors` value corresponds to *k* in the k-nearest neighbors algorithm. By default, a `KNeighborsClassifier` looks at the five nearest neighbors to make its predictions. For simplicity, we generally use the default estimator settings. For `KNeighborsClassifier`, these are described at:

ttp://scikit-
earn.org/stable/modules/generated/sklearn.neighbors-.KNeighborsClassifier.html

Many of these settings are beyond the scope of this book. In Part 2 of this case study, we'll discuss how to choose the best value for `n_neighbors`.

14.2.7 Predicting Digit Classes

Now that we've loaded the data into the `KNeighborsClassifier`, we can use it with the test samples to make predictions. Calling the estimator's **`predict` method** with `X_test` as an argument returns an array containing the predicted class of each test image:

lick here to view code image

```
In [19]: predicted = knn.predict(X=X_test)
```

```
In [20]: expected = y_test
```

Let's look at the `predicted` digits vs. `expected` digits for the first 20 test samples:

lick here to view code image

```
In [21]: predicted[:20]
Out[21]: array([0, 4, 9, 9, 3, 1, 4, 1, 5, 0, 4, 9, 4, 1, 5, 3, 3, 8, 5, 6

 n [22]: expected[:20]
Out[22]: array([0, 4, 9, 9, 3, 1, 4, 1, 5, 0, 4, 9, 4, 1, 5, 3, 3, 8, 3, 6
```

s you can see, in the first 20 elements, only the `predicted` and `expected` arrays' values at index 18 do not match. We expected a 3, but the model predicted a 5.

Let's use a list comprehension to locate *all* the incorrect predictions for the *entire* test set—that is, the cases in which the `predicted` and `expected` values do *not* match:

lick here to view code image

```
In [23]: wrong = [(p, e)   for (p, e) in zip(predicted, expected) if p !=

 n [24]: wrong
Out[24]:
[(5, 3),
 (8, 9),
 (4, 9),
 (7, 3),
 (7, 4),
 (2, 8),
 (9, 8),
 (3, 8),
 (3, 8),
 (1, 8)]
```

The list comprehension uses `zip` to create tuples containing the corresponding elements in `predicted` and `expected`. We include a tuple in the result only if its `p` (the predicted value) and `e` (the expected value) differ—that is, the predicted value was incorrect. In this example, the estimator incorrectly predicted only 10 of the 450 test samples. So the prediction accuracy of this estimator is an impressive 97.78%, even though we used only the estimator's default parameters.

14.3 CASE STUDY: CLASSIFICATION WITH K-NEAREST NEIGHBORS AND THE DIGITS DATASET, PART 2

In this section, we continue the digit classification case study. We'll:

- evaluate the k-NN classification estimator's accuracy,
- execute multiple estimators and can compare their results so you can choose the best one(s), and
- show how to tune k-NN's hyperparameter *k* to get the best performance out of a `KNeighborsClassifier`.

14.3.1 Metrics for Model Accuracy

Once you've trained and tested a model, you'll want to measure its accuracy. Here, we'll look at two ways of doing this—a classification estimator's `score` method and a *confusion matrix*.

Estimator Method `score`

Each estimator has a **`score`** method that returns an indication of how well the estimator performs for the test data you pass as arguments. For classification estimators, this method returns the *prediction accuracy* for the test data:

lick here to view code image

```
In [25]: print(f'{knn.score(X_test,    y_test):.2%}')
97.78%
```

The `kNeighborsClassifier`'s with its default *k* (that is, `n_neighbors=5`**)** achieved 97.78% prediction accuracy. Shortly, we'll perform hyperparameter tuning to try to determine the optimal value for *k*, hoping that we get even better accuracy.

Confusion Matrix

Another way to check a classification estimator's accuracy is via a **confusion matrix**, which shows the correct and incorrect predicted values (also known as the *hits* and *misses*) for a given class. Simply call the function **`confusion_matrix`** from the **`sklearn.metrics` module**, passing the `expected` classes and the `predicted` classes as arguments, as in:

C

lick here to view code image

```
In [26]: from sklearn.metrics import confusion_matrix

In [27]: confusion = confusion_matrix(y_true=expected,   y_pred=predicted)
```

The `y_true` keyword argument specifies the test samples' actual classes. People looked at the dataset's images and labeled them with specific classes (the digit values). The `y_pred` keyword argument specifies the predicted digits for those test images.

Below is the confusion matrix produced by the preceding call. The correct predictions are shown on the diagonal from top-left to bottom-right. This is called the **principal diagonal**. The nonzero values that are not on the principal diagonal indicate incorrect predictions:

lick here to view code image

```
In [28]: confusion
Out[28]:
array([[45,  0,  0,  0,  0,  0,  0,  0,  0,  0],
       [ 0, 45,  0,  0,  0,  0,  0,  0,  0,  0],
       [ 0,  0, 54,  0,  0,  0,  0,  0,  0,  0],
       [ 0,  0,  0, 42,  0,  1,  0,  1,  0,  0],
       [ 0,  0,  0,  0, 49,  0,  0,  1,  0,  0],
       [ 0,  0,  0,  0,  0, 38,  0,  0,  0,  0],
       [ 0,  0,  0,  0,  0,  0, 42,  0,  0,  0],
       [ 0,  0,  0,  0,  0,  0,  0, 45,  0,  0],
       [ 0,  1,  1,  2,  0,  0,  0,  0, 39,  1],
       [ 0,  0,  0,  0,  1,  0,  0,  0,  1, 41]])
```

Each row represents one distinct class—that is, one of the digits 0–9. The columns within a row specify how many of the test samples were classified into each distinct class. For example, row 0:

lick here to view code image

```
[45,  0,  0,  0,  0,  0,  0,  0,  0,  0]
```

represents the digit `0` class. The columns represent the ten possible target classes `0` through `9`. Because we're working with digits, the classes (`0`–`9`) and the row and column index numbers (`0`–`9`) happen to match. According to row 0, 45 test samples

C

were classified as the digit `0`, and *none* of the test samples were misclassified as any of the digits `1` through `9`. So 100% of the `0`s were correctly predicted.

On the other hand, consider row 8 which represents the results for the digit `8`:

lick here to view code image

```
[ 0,  1,  1,  2,  0,  0,  0,  0, 39,  1]
```

- The `1` at column index 1 indicates that one `8` was *incorrectly* classified as a `1`.
- The `1` at column index 2 indicates that one `8` was *incorrectly* classified as a `2`.
- The `2` at column index 3 indicates that two `8`s were *incorrectly* classified as `3`s.
- The `39` at column index 8 indicates that 39 `8`s were *correctly* classified as `8`s.
- The `1` at column index 9 indicates that one `8` was *incorrectly* classified as a `9`.

So the algorithm correctly predicted 88.63% (39 of 44) of the `8`s. Earlier we saw that the overall prediction accuracy of this estimator was 97.78%. The lower prediction accuracy for `8`s indicates that they're apparently harder to recognize than the other digits.

Classification Report

The `sklearn.metrics` module also provides function **`classification_report`**, which produces a table of **classification metrics** [5] based on the expected and predicted values:

[5] ttp://scikit-learn.org/stable/modules/model_evaluation.html#precision-recall-nd-f-measures.

lick here to view code image

```
In [29]: from sklearn.metrics import classification_report

In [30]: names = [str(digit) for   digit in digits.target_names]

In [31]: print(classification_report(expected, predicted,
    ...:       target_names=names))
```

```
    ...:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        45
           1       0.98      1.00      0.99        45
           2       0.98      1.00      0.99        54
           3       0.95      0.95      0.95        44
           4       0.98      0.98      0.98        50
           5       0.97      1.00      0.99        38
           6       1.00      1.00      1.00        42
           7       0.96      1.00      0.98        45
           8       0.97      0.89      0.93        44
           9       0.98      0.95      0.96        43

   micro avg       0.98      0.98      0.98       450
   macro avg       0.98      0.98      0.98       450
weighted avg       0.98      0.98      0.98       450
```

In the report:

- **precision** is the total number of correct predictions for a given digit divided by the total number of predictions for that digit. You can confirm the precision by looking at each column in the confusion matrix. For example, if you look at column index `7`, you'll see `1`s in rows 3 and 4, indicating that one 3 and one 4 were incorrectly classified as 7s and a `45` in row 7 indicating the 45 images were correctly classified as 7s. So the *precision* for the digit `7` is 45/47 or 0.96.

- **recall** is the total number of correct predictions for a given digit divided by the total number of samples that should have been predicted as that digit. You can confirm the recall by looking at each row in the confusion matrix. For example, if you look at row index `8`, you'll see three `1`s and a `2` indicating that some `8`s were incorrectly classified as other digits and a `39` indicating that 39 images were correctly classified. So the *recall* for the digit `8` is 39/44 or 0.89.

- **f1-score**—This is the average of the *precision* and the *recall*.

- **support**—The number of samples with a given expected value. For example, 50 samples were labeled as 4s, and 38 samples were labeled as 5s.

For details on the averages displayed at the bottom of the report, see:

ttp://scikit-
earn.org/stable/modules/generated/sklearn.metrics-.classification_report.html

h

Visualizing the Confusion Matrix

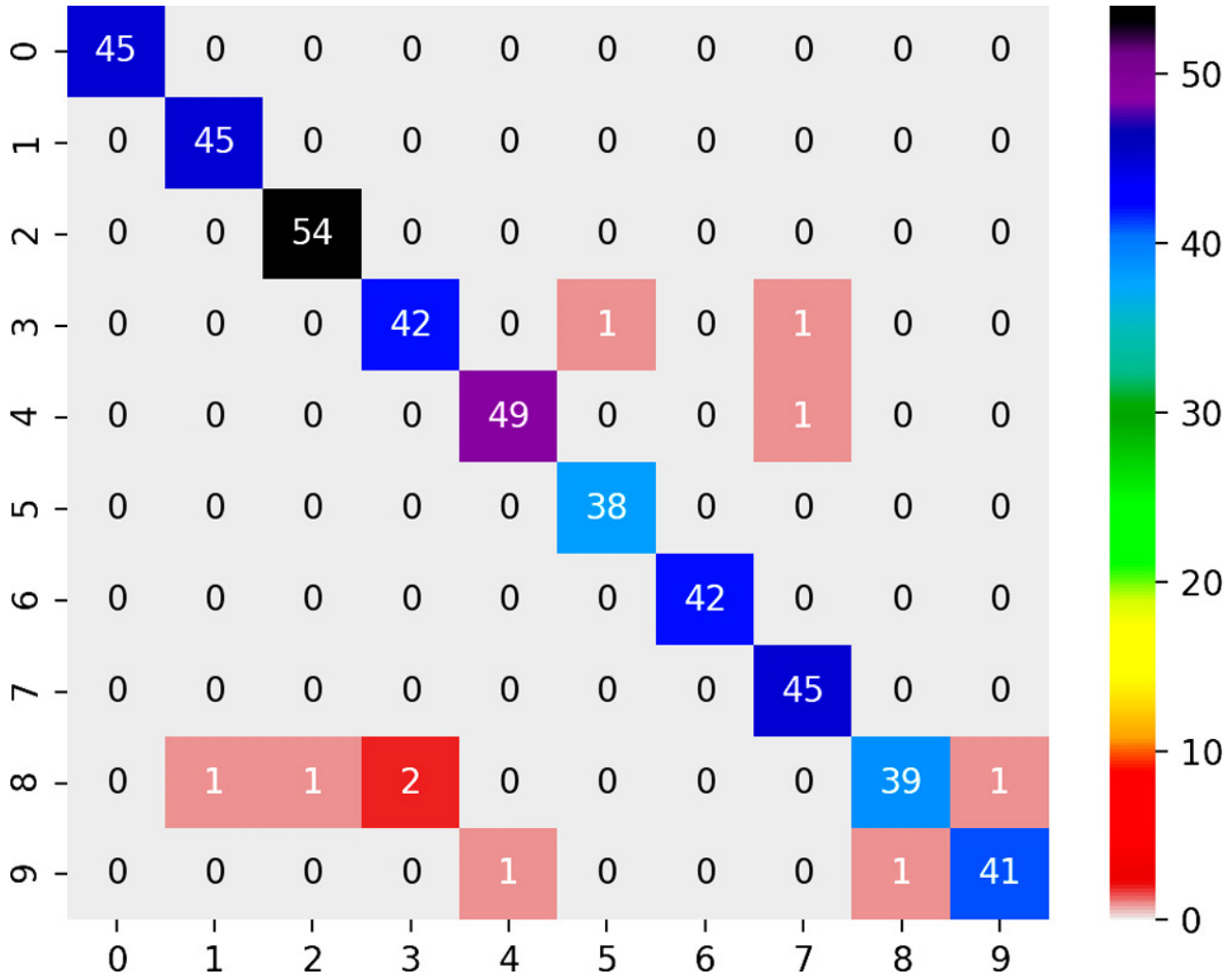
A **heat map** displays values as colors, often with values of higher magnitude displayed as more intense colors. Seaborn's graphing functions work with two-dimensional data. When using a pandas `DataFrame` as the data source, Seaborn automatically labels its visualizations using the column names and row indices. Let's convert the confusion matrix into a `DataFrame`, then graph it:

lick here to view code image

```
In [32]: import pandas as pd

In [33]: confusion_df = pd.DataFrame(confusion,   index=range(10),
    ...:     columns=range(10))
    ...:

In [34]: import seaborn as sns

In [35]: axes = sns.heatmap(confusion_df, annot=True,
    ...:                    cmap='nipy_spectral_r')
    ...:
```

The Seaborn function **`heatmap`** creates a heat map from the specified `DataFrame`. The keyword argument `annot=True` (short for "annotation") displays a color bar to the right of the diagram, showing how the values correspond to the heat map's colors. The `cmap='nipy_spectral_r'` keyword argument specifies which color map to use. We used the `nipy_spectral_r` color map with the colors shown in the heat map's color bar. When you display a confusion matrix as a heat map, the principal diagonal and the incorrect predictions stand out nicely.

C



14.3.2 K-Fold Cross-Validation

K-fold cross-validation enables you to use all of your data for *both* training *and* testing, to get a better sense of how well your model will make predictions for new data by repeatedly training and testing the model with different portions of the dataset. K-fold cross-validation splits the dataset into k equal-size **folds** (this k is unrelated to k in the k-nearest neighbors algorithm). You then repeatedly train your model with $k - 1$ folds and test the model with the remaining fold. For example, consider using $k = 10$ with folds numbered 1 through 10. With 10 folds, we'd do 10 successive training and testing cycles:

- First, we'd train with folds 1–9, then test with fold 10.
- Next, we'd train with folds 1–8 and 10, then test with fold 9.
- Next, we'd train with folds 1–7 and 9–10, then test with fold 8.

This training and testing cycle continues until each fold has been used to test the model.

`KFold` Class

Scikit-learn provides the **`KFold`** class and the **`cross_val_score` function** (both in the module `sklearn.model_selection`) to help you perform the training and testing cycles described above. Let's perform k-fold cross-validation with the Digits dataset and the `KNeighborsClassifier` created earlier. First, create a `KFold` object:

lick here to view code image

```
In [36]: from   sklearn.model_selection import KFold

In [37]: kfold = KFold(n_splits=10,   random_state=11, shuffle=True)
```

The keyword arguments are:

- `n_splits=10`, which specifies the number of folds.
- `random_state=11`, which seeds the random number generator for *reproducibility*.
- `shuffle=True`, which causes the `KFold` object to randomize the data by shuffling it before splitting it into folds. This is particularly important if the samples might be ordered or grouped. For example, the Iris dataset we'll use later in this chapter has 150 samples of three *Iris* species—the first 50 are *Iris setosa*, the next 50 are *Iris versicolor* and the last 50 are *Iris virginica*. If we do not shuffle the samples, then the training data might contain none of a particular *Iris* species and the test data might be all of one species.

Using the `KFold` Object with Function `cross_val_score`

Next, use function `cross_val_score` to train and test your model:

lick here to view code image

```
In [38]: from sklearn.model_selection   import cross_val_score

In [39]: scores = cross_val_score(estimator=knn,   X=digits.data,
    ...:     y=digits.target, cv=kfold)
    ...:
```

The keyword arguments are:

C

- `estimator=knn`, which specifies the estimator you'd like to validate.
- `X=digits.data`, which specifies the samples to use for training and testing.
- `y=digits.target`, which specifies the target predictions for the samples.
- `cv=kfold`, which specifies the cross-validation generator that defines how to split the samples and targets for training and testing.

Function `cross_val_score` returns an array of accuracy scores—one for each fold. As you can see below, the model was quite accurate. Its *lowest* accuracy score was `0.97777778` (97.78%) and in one case it was 100% accurate in predicting an entire fold:

lick here to view code image

```
In [40]: scores
Out[40]:
array([0.97777778, 0.99444444, 0.98888889, 0.97777778,   0.98888889,
       0.99444444, 0.97777778, 0.98882682, 1.        ,   0.98324022])
```

Once you have the accuracy scores, you can get an overall sense of the model's accuracy by calculating the mean accuracy score and the standard deviation among the 10 accuracy scores (or whatever number of folds you choose):

lick here to view code image

```
In [41]: print(f'Mean accuracy: {scores.mean():.2%}')
Mean accuracy: 98.72%

In [42]: print(f'Accuracy standard   deviation: {scores.std():.2%}')
Accuracy standard deviation: 0.75%
```

On average, the model was 98.72% accurate—even better than the 97.78% we achieved when we trained the model with 75% of the data and tested the model with 25% earlier.

14.3.3 Running Multiple Models to Find the Best One

It's difficult to know in advance which machine learning model(s) will perform best for a given dataset, especially when they hide the details of how they operate from their users. Even though the `KNeighborsClassifier` predicts digit images with a high

C

degree of accuracy, it's possible that other scikit-learn estimators are even more accurate. Scikit-learn provides many models with which you can quickly train and test your data. This encourages you to run *multiple models* to determine which is the best for a particular machine learning study.

Let's use the techniques from the preceding section to compare several classification estimators—`KNeighborsClassifier`, `SVC` and `GaussianNB` (there are more). Though we have not studied the `SVC` and `GaussianNB` estimators, scikit-learn nevertheless makes it easy for you to test-drive them by using their default settings. [6] First, let's import the other two estimators:

[6] To avoid a warning in the current scikit-learn version at the time of this writing (version 0.20), we supplied one keyword argument when creating the `SVC` estimator. This arguments value will become the default in scikit-learn version 0.22.

lick here to view code image

```
In [43]: from sklearn.svm import SVC

In [44]: from   sklearn.naive_bayes import GaussianNB
```

Next, let's create the estimators. The following dictionary contains key–value pairs for the existing `KNeighborsClassifier` we created earlier, plus new `SVC` and `GaussianNB` estimators:

lick here to view code image

```
In [45]: estimators = {
    ...:     'KNeighborsClassifier':   knn,
    ...:     'SVC': SVC(gamma='scale'),
    ...:     'GaussianNB':   GaussianNB()}
    ...:
```

Now, we can execute the models:

lick here to view code image

```
In [46]: for estimator_name,   estimator_object in estimators.items():
    ...:     kfold = KFold(n_splits=10,   random_state=11, shuffle=True)
    ...:     scores =   cross_val_score(estimator=estimator_object,
    ...:         X=digits.data, y=digits.target, cv=kfold)
    ...:     print(f'{estimator_name:>20}: ' +
```

C

```
...:             f'mean accuracy={scores.mean():.2%}; ' +
...:             f'standard   deviation={scores.std():.2%}')
...:
KNeighborsClassifier: mean accuracy=98.72%; standard   deviation=0.75%
                 SVC: mean accuracy=99.00%; standard   deviation=0.85%
          GaussianNB: mean accuracy=84.48%; standard   deviation=3.47%
```

This loop iterates through items in the `estimators` dictionary and for each key-value pair performs the following tasks:

- Unpacks the key into `estimator_name` and value into `estimator_object`.
- Creates a `KFold` object that shuffles the data and produces 10 folds. The keyword argument `random_state` is particularly important here because it ensures that each estimator works with identical folds, so we're comparing "apples to apples."
- Evaluates the current `estimator_object` using `cross_val_score`.
- Prints the estimator's name, followed by the mean and standard deviation of the accuracy scores' computed for each of the 10 folds.

Based on the results, it appears that we can get slightly better accuracy from the `SVC` estimator—at least when using the estimator's default settings. It's possible that by tuning some of the estimators' settings, we could get even better results. The `KNeighborsClassifier` and `SVC` estimators' accuracies are nearly identical so we might want to perform hyperparameter tuning on each to determine the best.

Scikit-Learn Estimator Diagram

The scikit-learn documentation provides a helpful diagram for choosing the right estimator, based on the kind and size of your data and the machine learning task you wish to perform:

ttps://scikit-learn.org/stable/tutorial/machine_learning_map/index.html

14.3.4 Hyperparameter Tuning

Earlier in this section, we mentioned that *k* in the k-nearest neighbors algorithm is a hyperparameter of the algorithm. Hyperparameters are set *before* using the algorithm to train your model. In real-world machine learning studies, you'll want to use hyperparameter tuning to choose hyperparameter values that produce the best possible

h

predictions.

To determine the best value for *k* in the kNN algorithm, try different values of *k* then compare the estimator's performance with each. We can do this using techniques similar to comparing estimators. The following loop creates `KNeighborsClassifier`s with odd *k* values from 1 through 19 (again, we use odd *k* values in kNN to avoid ties) and performs k-fold cross-validation on each. As you can see from the accuracy scores and standard deviations, the *k* value `1` in kNN produces the most accurate predictions for the Digits dataset. You can also see that accuracy tends to decrease for higher *k* values:

lick here to view code image

```
In [47]: for k in range(1, 20, 2):
    ...:     kfold = KFold(n_splits=10,   random_state=11, shuffle=True)
    ...:     knn = KNeighborsClassifier(n_neighbors=k)
    ...:     scores = cross_val_score(estimator=knn,
    ...:         X=digits.data, y=digits.target, cv=kfold)
    ...:     print(f'k={k:<2}; mean accuracy={scores.mean():.2%}; ' +
    ...:           f'standard   deviation={scores.std():.2%}')
    ...:
k=1 ; mean accuracy=98.83%; standard deviation=0.58%
k=3 ; mean accuracy=98.78%; standard deviation=0.78%
k=5 ; mean accuracy=98.72%; standard deviation=0.75%
k=7 ; mean accuracy=98.44%; standard deviation=0.96%
k=9 ; mean accuracy=98.39%; standard deviation=0.80%
k=11; mean accuracy=98.39%; standard deviation=0.80%
k=13; mean accuracy=97.89%; standard deviation=0.89%
k=15; mean accuracy=97.89%; standard deviation=1.02%
k=17; mean accuracy=97.50%; standard deviation=1.00%
k=19; mean accuracy=97.66%; standard deviation=0.96%
```

Machine learning is not without its costs, especially as we head toward big data and deep learning. You must "know your data" and "know your tools." For example, compute time grows rapidly with *k*, because k-NN needs to perform more calculations to find the nearest neighbors. There is also function `cross_validate`, which does cross-validation *and* times the results.

14.4 CASE STUDY: TIME SERIES AND SIMPLE LINEAR REGRESSION

In the previous section, we demonstrated classification in which each sample was associated with a *distinct* class. Here, we continue our discussion of simple linear

C

regression—the simplest of the regression algorithms—that began in hapter 10's Intro to Data Science section. Recall that given a collection of numeric values representing an independent variable and a dependent variable, simple linear regression describes the relationship between these variables with a straight line, known as the regression line.

Previously, we performed simple linear regression on a time series of average New York City January high-temperature data for 1895 through 2018. In that example, we used Seaborn's `regplot` function to create a scatter plot of the data with a corresponding regression line. We also used the `scipy.stats` module's `linregress` function to calculate the regression line's slope and intercept. We then used those values to predict future temperatures and estimate past temperatures.

In this section, we'll

- use a *scikit-learn estimator* to reimplement the simple linear regression we showed in hapter 10,
- use Seaborn's `scatterplot` function to plot the data and Matplotlib's `plot` function to display the regression line, then
- use the coefficient and intercept values calculated by the scikit-learn estimator to make predictions.

Later, we'll look at *multiple linear regression* (also simply called *linear regression*).

For your convenience, we provide the temperature data in the `ch14` examples folder in a CSV file named `ave_hi_nyc_jan_1895-2018.csv`. Once again, launch IPython with the `--matplotlib` option:

```
ipython --matplotlib
```

Loading the Average High Temperatures into a `DataFrame`

As we did in hapter 10, let's load the data from `ave_hi_nyc_jan_1895-2018.csv`, rename the `'Value'` column to `'Temperature'`, remove `01` from the end of each date value and display a few data samples:

lick here to view code image

```
In [1]: import pandas as pd
```

C

```
In [2]: nyc = pd.read_csv('ave_hi_nyc_jan_1895-2018.csv')

In [3]: nyc.columns = ['Date',   'Temperature', 'Anomaly']

In [4]: nyc.Date = nyc.Date.floordiv(100)

In [5]: nyc.head(3)
Out[5]:
   Date  Temperature  Anomaly
0  1895         34.2     -3.2
1  1896         34.7     -2.7
2  1897         35.5     -1.9
```

Splitting the Data for Training and Testing

In this example, we'll use the **`LinearRegression`** estimator from **`sklearn.linear_model`**. By default, this estimator uses *all* the numeric features in a dataset, performing a **multiple linear regression** (which we'll discuss in the next section). Here, we perform *simple linear regression* using *one* feature as the independent variable. So, we'll need to select one feature (the `Date`) from the dataset.

When you select one column from a two-dimensional `DataFrame`, the result is a *one-dimensional* `Series`. However, scikit-learn estimators require their training and testing data to be *two-dimensional arrays* (or two-dimensional *array-like* data, such as lists of lists or pandas `DataFrame`s). To use one-dimensional data with an estimator, you must transform it from one dimension containing *n* elements, into two dimensions containing *n rows* and one *column* as you'll see below.

As we did in the previous case study, let's split the data into training and testing sets. Once again, we used the keyword argument `random_state` for reproducibility:

lick here to view code image

```
In [6]: from sklearn.model_selection   import train_test_split

In [7]: X_train, X_test, y_train, y_test =   train_test_split(
   ...:     nyc.Date.values.reshape(-1,   1), nyc.Temperature.values,
   ...:     random_state=11)
   ...:
```

The expression `nyc.Date` returns the `Date` column's `Series`, and the `Series`' `values` attribute returns the NumPy array containing that `Series`' values. To transform this one-dimensional array into two dimensions, we call the array's **`reshape`**

C

method. Normally, two arguments are the precise number of rows and columns. However, the first argument `-1` tells `reshape` to *infer* the number of rows, based on the number of columns (`1`) and the number of elements (124) in the array. The transformed array will have only one column, so `reshape` infers the number of rows to be 124, because the only way to fit 124 elements into an array with one column is by distributing them over 124 rows.

We can confirm the 75%–25% train-test split by checking the `shape`s of `X_train` and `X_test`:

lick here to view code image

```
In [8]: X_train.shape
Out[8]: (93, 1)

In [9]: X_test.shape
Out[9]: (31, 1)
```

Training the Model

Scikit-learn does not have a separate class for simple linear regression because it's just a special case of multiple linear regression, so let's train a `LinearRegression` estimator:

lick here to view code image

```
In [10]: from sklearn.linear_model   import LinearRegression

In [11]: linear_regression = LinearRegression()

In [12]: linear_regression.fit(X=X_train, y=y_train)
Out[12]:
LinearRegression(copy_X=True, fit_intercept=True,   n_jobs=None,
         normalize=False)
```

After training the estimator, `fit` returns the estimator, and IPython displays its string representation. For descriptions of the default settings, see:

ttp://scikit-
earn.org/stable/modules/generated/sklearn.linear_model-.LinearRegression.html

To find the best fitting regression line for the data, the `LinearRegression` estimator iteratively adjusts the slope and intercept values to minimize the sum of the squares of

the data points' distances from the line. In hapter 10's Intro to Data Science section, we gave some insight into how the slope and intercept values are discovered.

Now, we can get the slope and intercept used in the $y = mx + b$ calculation to make predictions. The slope is stored in the estimator's **`coeff_`** attribute (m in the equation) and the intercept is stored in the estimator's **`intercept_`** attribute (b in the equation):

lick here to view code image

```
In [13]: linear_regression.coef_
Out[13]: array([0.01939167])

In [14]: linear_regression.intercept_
Out[14]: -0.30779820252656265
```

We'll use these later to plot the regression line and make predictions for specific dates.

Testing the Model

Let's test the model using the data in `X_test` and check some of the predictions throughout the dataset by displaying the `predicted` and `expected` values for every fifth element—we discuss how to assess the regression model's accuracy in ection 4.5.8:

lick here to view code image

```
In [15]: predicted = linear_regression.predict(X_test)

In [16]: expected = y_test

In [17]: for p, e in zip(predicted[::5],   expected[::5]):
    ...:     print(f'predicted: {p:.2f}, expected: {e:.2f}')
    ...:
predicted: 37.86, expected: 31.70
predicted: 38.69, expected: 34.80
predicted: 37.00, expected: 39.40
predicted: 37.25, expected: 45.70
predicted: 38.05, expected: 32.30
predicted: 37.64, expected: 33.80
predicted: 36.94, expected: 39.70
```

Predicting Future Temperatures and Estimating Past Temperatures

Let's use the coefficient and intercept values to predict the January 2019 average high temperature and to estimate what the average high temperature was in January of

1890. The `lambda` in the following snippet implements the equation for a line

```
y = mx + b
```

using the `coef_` as m and the `intercept_` as b.

lick here to view code image

```
In [18]: predict = (lambda x:   linear_regression.coef_ * x +
    ...:                        linear_regression.intercept_)
    ...:

In [19]: predict(2019)
Out[19]: array([38.84399018])

In [20]: predict(1890)
Out[20]: array([36.34246432])
```

Visualizing the Dataset with the Regression Line

Next, let's create a scatter plot of the dataset using Seaborn's **`scatterplot`** function and Matplotlib's **`plot`** function. First, use `scatterplot` with the `nyc DataFrame` to display the data points:

lick here to view code image

```
In [21]: import seaborn as sns

In [22]: axes = sns.scatterplot(data=nyc, x='Date', y='Temperature',
    ...:     hue='Temperature',   palette='winter', legend=False)
    ...:
```

The keyword arguments are:

- `data`, which specifies the `DataFrame` (`nyc`) containing the data to display.
- `x` and `y`, which specify the names of `nyc`'s columns that are the source of the data along the x- and y-axes, respectively. In this case, `x` is the `'Date'` and `y` is the `'Temperature'`. The corresponding values from each column form x-y coordinate pairs used to plot the dots.
- `hue`, which specifies which column's data should be used to determine the dot

C

colors. In this case, we use the `'Temperature'` column. Color is not particularly important in this example, but we wanted to add some visual interest to the graph.

- `palette`, which specifies a Matplotlib color map from which to choose the dots' colors.
- `legend=False`, which specifies that `scatterplot` should not show a legend for the graph—the default is `True`, but we do not need a legend for this example.

As we did in hapter 10, let's scale the y-axis range of values so you'll be able to see the linear relationship better once we display the regression line:

lick here to view code image

```
In [23]: axes.set_ylim(10, 70)
Out[23]: (10, 70)
```

Next, let's display the regression line. First, create an array containing the minimum and maximum date values in `nyc.Date`. These are the *x*-coordinates of the regression line's start and end points:

lick here to view code image

```
In [24]: import numpy as np

In [25]: x = np.array([min(nyc.Date.values),   max(nyc.Date.values)])
```

Passing the array `x` to the `predict lambda` from snippet `[16]` produces an array containing the corresponding predicted values, which we'll use as the *y*-coordinates:

lick here to view code image

```
In [26]: y = predict(x)
```

Finally, we can use Matplotlib's `plot` function to plot a line based on the `x` and `y` arrays, which represent the *x*- and *y*-coordinates of the points, respectively:

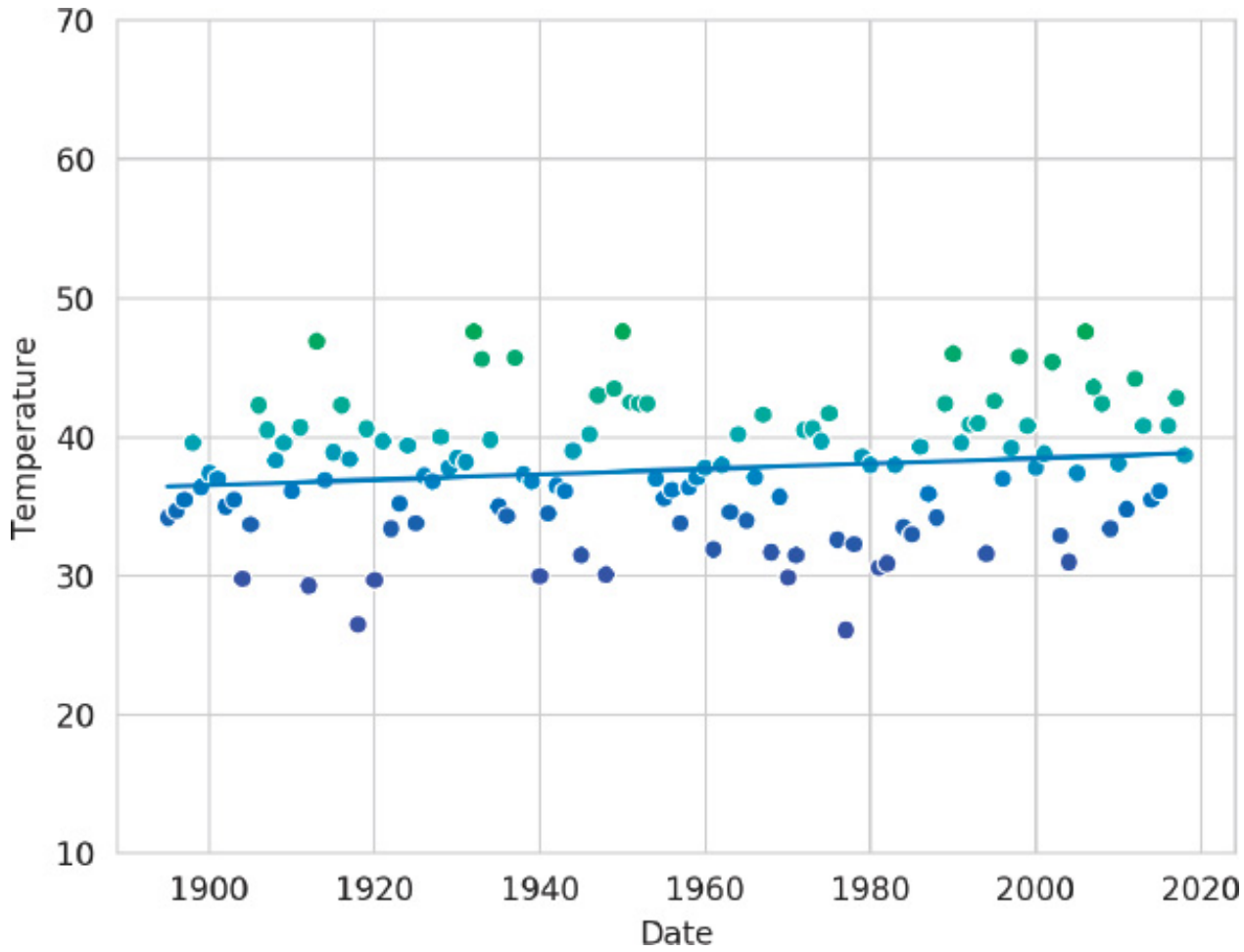
lick here to view code image

```
In [27]: import   matplotlib.pyplot as plt
```

C

```
In [28]: line = plt.plot(x, y)
```

The resulting scatterplot and regression line are shown below. This graph is nearly identical to the one you saw in hapter 10's Intro to Data Science section.



Overfitting/Underfitting

When creating a model, a key goal is to ensure that it is capable of making accurate predictions for data it has not yet seen. Two common problems that prevent accurate predictions are overfitting and underfitting:

- **Underfitting** occurs when a model is too simple to make predictions, based on its training data. For example, you may use a linear model, such as simple linear regression, when in fact, the problem really requires a non-linear model. For example, temperatures vary significantly throughout the four seasons. If you're trying to create a general model that can predict temperatures year-round, a simple linear regression model will underfit the data.

- **Overfitting** occurs when your model is too complex. The most extreme case, would be a model that memorizes its training data. That may be acceptable if your new data looks *exactly* like your training data, but ordinarily that's not the case. When

C

you make predictions with an overfit model, new data that matches the training data will produce perfect predictions, but the model will not know what to do with data it has never seen.

For additional information on underfitting and overfitting, see

- ttps://en.wikipedia.org/wiki/Overfitting
- ttps://machinelearningmastery.com/overfitting-and-nderfitting-with-machine-learning-algorithms/

14.5 CASE STUDY: MULTIPLE LINEAR REGRESSION WITH THE CALIFORNIA HOUSING DATASET

In hapter 10's Intro to Data Science section, we performed simple linear regression on a small weather data time series using pandas, Seaborn's `regplot` function and the SciPy's `stats` module's `linregress` function. In the previous section, we reimplemented that same example using scikit-learn's `LinearRegression` estimator, Seaborn's `scatterplot` function and Matplotlib's `plot` function. Now, we'll perform linear regression with a much larger real-world dataset.

The **California Housing dataset** [7] bundled with scikit-learn has 20,640 samples, each with eight numerical features. We'll perform a *multiple linear regression* that uses all eight numerical features to make more sophisticated housing price predictions than if we were to use only a single feature or a subset of the features. Once again, scikit-learn will do most of the work for you—`LinearRegression` performs multiple linear regression by default.

[7] ttp://lib.stat.cmu.edu/datasets. Pace, R. Kelley and Ronald Barry, Sparse Spatial Autoregressions, Statistics and Probability Letters, 33 (1997) 291-297. Submitted to the StatLib Datasets Archive by Kelley Pace (pace@unix1.sncc.lsu.edu). [9/Nov/99].

We'll visualize some of the data using Matplotlib and Seaborn, so launch IPython with Matplotlib support:

```
ipython --matplotlib
```

14.5.1 Loading the Dataset

According to the California Housing Prices dataset's description in scikit-learn, "This dataset was derived from the 1990 U.S. census, using one row per census block group. A block group is the smallest geographical unit for which the U.S. Census Bureau publishes sample data (a block group typically has a population of 600 to 3,000 people)." The dataset has 20,640 samples—one per block group—with eight features each:

- median income—in tens of thousands, so 8.37 would represent $83,700
- median house age—in the dataset, the maximum value for this feature is 52
- average number of rooms
- average number of bedrooms
- block population
- average house occupancy
- house block latitude
- house block longitude

Each sample also has as its *target* a corresponding median house value in hundreds of thousands, so 3.55 would represent $355,000. In the dataset, the maximum value for this feature is 5, which represents $500,000.

It's reasonable to expect that more bedrooms or more rooms or higher income would mean higher house value. By combining these features to make predictions, we're more likely to get more accurate predictions.

Loading the Data

Let's load the dataset and familiarize ourselves with it. The **`fetch_california_housing`** function from the `sklearn.datasets` module returns a `Bunch` object containing the data and other information about the dataset:

lick here to view code image

```
In [1]: from sklearn.datasets import fetch_california_housing

In [2]: california = fetch_california_housing()
```

C