CHAPTER 9

Examples

This chapter includes several larger examples of web scrapers. Contrary to most of the examples showcased during the previous chapters, the examples here serve a twofold purpose. First, they showcase some more examples using real-life websites instead of a curated, safe environment. The reason why we haven't used many real-life examples so far is due to the dynamic nature of the web. It might be that the examples covered here do not provide the exact same results anymore or will be broken by the time you read them. That being said, we have tried to use a selection of sites that are rather scraper friendly and not very prone to changes. The second purpose of these examples is to highlight how various concepts seen throughout the book "fall together" and interact, as well as to hint toward some interesting data science-oriented use cases.

The following examples are included in this chapter:

- **Scraping Hacker News:** This example uses requests and Beautiful Soup to scrape the Hacker News front page.
- **Using the Hacker News API:** This example provides an alternative by showing how you can use APIs with requests.
- **Quotes to Scrape:** This example uses requests and Beautiful Soup and introduces the "dataset" library as an easy means to store data.
- **Books to Scrape:** This example uses requests and Beautiful Soup, as well as the dataset library, illustrating how you can run a scraper again without storing duplicate results.
- **Scraping GitHub Stars:** This example uses requests and Beautiful Soup to scrape GitHub repositories and show how you can perform a login using requests, reiterating our warnings regarding legal concerns.

S. vanden Broucke and B. Baesens, *Practical Web Scraping for Data Science*,
https://doi.org/10.1007/978-1-4842-3582-9_9

- **Scraping Mortgage Rates:** This example uses requests to scrape mortgage rates using a particularly tricky site.
- **Scraping and Visualizing IMDB Ratings:** This example uses requests and Beautiful Soup to get a list of IMDB ratings for TV series episodes. We also introduce the "matplotlib" library to create plots in Python.
- **Scraping IATA Airline Information:** This example uses requests and Beautiful Soup to scrape airline information from a site that employs a difficult web form. An alternative approach using Selenium is also provided. Scraped results are converted to a tabular format using the "pandas" library, also introduced in this example.
- **Scraping and Analyzing Web Forum Interactions:** This example uses requests and Beautiful Soup to scrape web forum posts and stores them using the dataset library. From the collected results, we use pandas and matplotlib to create heat map plots showing user activity.
- **Collecting and Clustering a Fashion Data Set:** This example uses requests and Beautiful Soup to download a set of fashion images. The images are then clustered using the "scikit-learn" library.
- **Sentiment Analysis of Scraped Amazon Reviews:** This example uses requests and Beautiful Soup to scrape a list of user reviews from Amazon, stored using the dataset library. We then analyze these using the "nltk" and "vaderSentiment" libraries in Python, and plot the results using matplotlib.
- **Scraping and Analyzing News Articles:** This example uses Selenium to scrape a list of news articles, stored using the dataset library. We then associate these to a list of topics by constructing a topic model using nltk.
- **Scraping and Analyzing a Wikipedia Graph:** In this example, we extended our Wikipedia crawler to scrape pages using requests and Beautiful Soup, stored using the dataset library, which we then use to create a graph using "NetworkX" and visualize it with matplotlib.

- **Scraping and Visualizing a Board Members Graph:** This example uses requests and Beautiful Soup to scrape board members for S&P 500 companies. A graph is created using NetworkX and visualized using "Gephi."
- **Breaking CAPTCHA's Using Deep Learning:** This example shows how a convolutional neural network can be used to break CAPTCHA's.

Source Code The source code for all examples is also provided at the companion website for this book at *http://www.webscrapingfordatascience.com*.

9.1 Scraping Hacker News

We're going to scrape the *https://news.ycombinator.com/news* front page, using requests and Beautiful Soup. Take some time to explore the page if you haven't heard about it already. Hacker News is a popular aggregator of news articles that "hackers" (computer scientists, entrepreneurs, data scientists) find interesting.

We'll store the scraped information in a simple Python list of dictionary objects for this example. The code to scrape this page looks as follows:

```
import requests
import re
from bs4 import BeautifulSoup

articles = []

url = 'https://news.ycombinator.com/news'

r = requests.get(url)
html_soup = BeautifulSoup(r.text, 'html.parser')

for item in html_soup.find_all('tr', class_='athing'):
    item_a = item.find('a', class_='storylink')
    item_link = item_a.get('href') if item_a else None
```

```
    item_text = item_a.get_text(strip=True) if item_a else None
    next_row = item.find_next_sibling('tr')
    item_score = next_row.find('span', class_='score')
    item_score = item_score.get_text(strip=True) if item_score else '0 points'
    # We use regex here to find the correct element
    item_comments = next_row.find('a', string=re.compile('\d+( |\s)
    comment(s?)'))
    item_comments = item_comments.get_text(strip=True).replace('\xa0', ' ') \
                        if item_comments else '0 comments'
    articles.append({
        'link' : item_link,
        'title' : item_text,
        'score' : item_score,
        'comments' : item_comments})

for article in articles:
    print(article)
```

This will output the following:

```
{'link': 'http://moolenaar.net/habits.html', 'title': 'Seven habits of   ↩
    effective text editing (2000)', 'score': '44 points', 'comments':    ↩
    '9 comments'}
{'link': 'https://www.repository.cam.ac.uk/handle/1810/251038', 'title': ↩
    'Properties of expanding universes (1966)', 'score': '52 points',    ↩
    'comments': '8 comments'}
[...]
```

Try expanding this code to scrape a link to the comments page as well. Think about potential use cases that would be possible when you also scrape the comments themselves (for example, in the context of text mining).

9.2 Using the Hacker News API

Note that Hacker News also offers an API providing structured, JSON-formatted results (see *https://github.com/HackerNews/API*). Let's rework our Python code to now serve as an API client without relying on Beautiful Soup for HTML parsing:

```
import requests

articles = []

url = 'https://hacker-news.firebaseio.com/v0'

top_stories = requests.get(url + '/topstories.json').json()

for story_id in top_stories:
    story_url = url + '/item/{}.json'.format(story_id)
    print('Fetching:', story_url)
    r = requests.get(story_url)
    story_dict = r.json()
    articles.append(story_dict)

for article in articles:
    print(article)
```

This will output the following:

```
Fetching: https://hacker-news.firebaseio.com/v0/item/15532457.json
Fetching: https://hacker-news.firebaseio.com/v0/item/15531973.json
Fetching: https://hacker-news.firebaseio.com/v0/item/15532049.json
[...]
{'by': 'laktak', 'descendants': 30, 'id': 15532457, 'kids': [15532761,
    15532768, 15532635, 15532727, 15532776, 15532626, 15532700, 15532634],
    'score': 60, 'time': 1508759764, 'title': 'Seven habits of effective
    text editing (2000)', 'type': 'story', 'url': 'http://moolenaar.net/
    habits.html'}
[...]
```

9.3 Quotes to Scrape

We're going to scrape *http://quotes.toscrape.com*, using requests and Beautiful Soup. This page is provided by Scrapinghub as a more realistic scraping playground. Take some time to explore the page. We'll scrape out all the information, that is:

- The quotes, with their author and tags;
- And the author information, that is, date and place of birth, and description.

We'll store this information in a SQLite database. Instead of using the "records" library and writing manual SQL statements, we're going to use the "dataset" library (see *https://dataset.readthedocs.io/en/latest/*). This library provides a simple abstraction layer removing most direct SQL statements without the necessity for a full ORM model, so that we can use a database just like we would with a CSV or JSON file to quickly store some information. Installing a dataset can be done easily through pip:

```
pip install -U dataset
```

Not a Full ORM Note that dataset does not want to replace a full-blown ORM (Object Relational Mapping) library like SQLAlchemy (even though it uses SQLAlchemy behind the scenes). It's meant simply to quickly store a bunch of data in a database without having to define a schema or write SQL. For more advanced use cases, it's a good idea to consider using a true ORM library or to define a database schema by hand and query it manually.

The code to scrape this site looks as follows:

```
import requests
import dataset
from bs4 import BeautifulSoup
from urllib.parse import urljoin, urlparse

db = dataset.connect('sqlite:///quotes.db')

authors_seen = set()
```

```
base_url = 'http://quotes.toscrape.com/'

def clean_url(url):
    # Clean '/author/Steve-Martin' to 'Steve-Martin'
    # Use urljoin to make an absolute URL
    url = urljoin(base_url, url)
    # Use urlparse to get out the path part
    path = urlparse(url).path
    # Now split the path by '/' and get the second part
    # E.g. '/author/Steve-Martin' -> ['', 'author', 'Steve-Martin']
    return path.split('/')[2]

def scrape_quotes(html_soup):
    for quote in html_soup.select('div.quote'):
        quote_text = quote.find(class_='text').get_text(strip=True)
        quote_author_url = clean_url(quote.find(class_='author') \
                                     .find_next_sibling('a').get('href'))
        quote_tag_urls = [clean_url(a.get('href'))
                          for a in quote.find_all('a', class_='tag')]
        authors_seen.add(quote_author_url)
        # Store this quote and its tags
        quote_id = db['quotes'].insert({ 'text' : quote_text,
                                         'author' : quote_author_url })
        db['quote_tags'].insert_many(
                 [{'quote_id' : quote_id, 'tag_id' : tag} for tag in
                 quote_tag_urls])

def scrape_author(html_soup, author_id):
    author_name = html_soup.find(class_='author-title').get_text(strip=True)
    author_born_date = html_soup.find(class_='author-born-date').get_text
    (strip=True)
    author_born_loc = html_soup.find(class_='author-born-location').
    get_text(strip=True)
    author_desc = html_soup.find(class_='author-description').get_text
    (strip=True)
```

```
    db['authors'].insert({ 'author_id' : author_id,
                           'name' : author_name,
                           'born_date' : author_born_date,
                           'born_location' : author_born_loc,
                           'description' : author_desc})

# Start by scraping all the quote pages
url = base_url
while True:
    print('Now scraping page:', url)
    r = requests.get(url)
    html_soup = BeautifulSoup(r.text, 'html.parser')
    # Scrape the quotes
    scrape_quotes(html_soup)
    # Is there a next page?
    next_a = html_soup.select('li.next > a')
    if not next_a or not next_a[0].get('href'):
        break
    url = urljoin(url, next_a[0].get('href'))

# Now fetch out the author information
for author_id in authors_seen:
    url = urljoin(base_url, '/author/' + author_id)
    print('Now scraping author:', url)
    r = requests.get(url)
    html_soup = BeautifulSoup(r.text, 'html.parser')
    # Scrape the author information
    scrape_author(html_soup, author_id)
```

This will output the following:

```
Now scraping page: http://quotes.toscrape.com/
Now scraping page: http://quotes.toscrape.com/page/2/
Now scraping page: http://quotes.toscrape.com/page/3/
Now scraping page: http://quotes.toscrape.com/page/4/
Now scraping page: http://quotes.toscrape.com/page/5/
Now scraping page: http://quotes.toscrape.com/page/6/
```

```
Now scraping page: http://quotes.toscrape.com/page/7/
Now scraping page: http://quotes.toscrape.com/page/8/
Now scraping page: http://quotes.toscrape.com/page/9/
Now scraping page: http://quotes.toscrape.com/page/10/
Now scraping author: http://quotes.toscrape.com/author/Ayn-Rand
Now scraping author: http://quotes.toscrape.com/author/E-E-Cummings
[...]
```

Note that there are still a number of ways to make this code more robust. We're not checking for None results when scraping the quote or author pages. In addition, we're using "dataset" here to simply insert rows in three tables. In this case, dataset will automatically increment a primary "id" key. If you want to run this script again, you'll hence first have to clean up the database to start fresh, or modify the script to allow for resuming its work or updating the results properly. In later examples, we'll use dataset's `upsert` method to do so.

Once the script has finished, you can take a look at the database ("quotes.db") using a SQLite client such as "DB Browser for SQLite," which can be obtained from *http://sqlitebrowser.org/*. Figure 9-1 shows this tool in action.

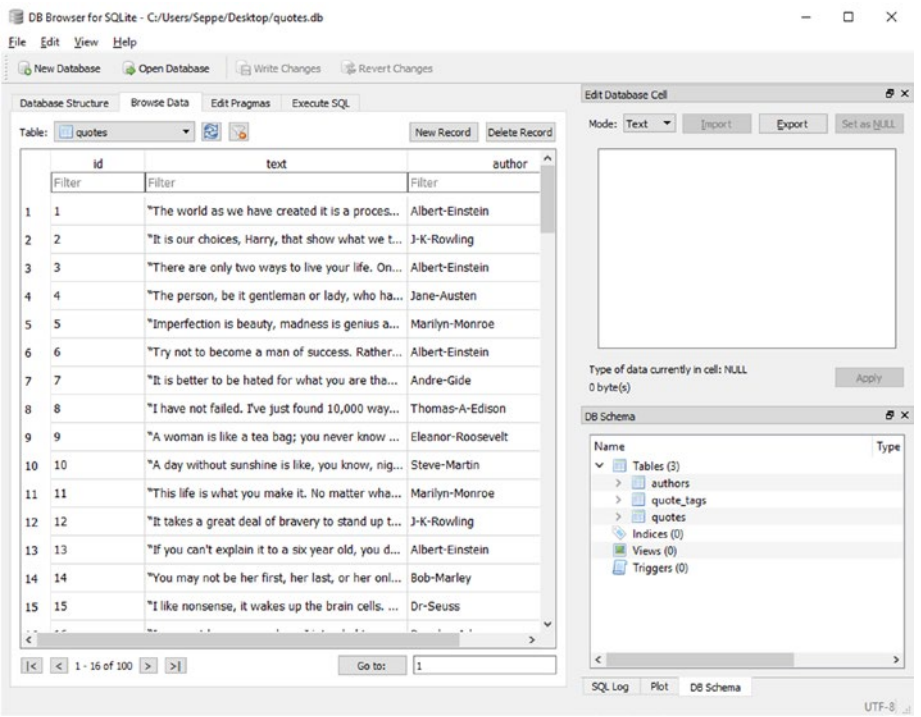

Figure 9-1. *Exploring an SQLite database with "DB Browser for SQLite"*

9.4 Books to Scrape

We're going to scrape *http://books.toscrape.com*, using requests and Beautiful Soup. This page is provided by Scrapinghub as a more realistic scraping playground. Take some time to explore the page. We'll scrape out all the information, that is, for every book, we'll obtain:

- Its title;
- Its image;
- Its price and stock availability;
- Its rating;
- Its product description;
- Other product information.

We're going to store this information in an SQLite database, again using the "dataset" library. However, this time we're going to write our program in such a way that it takes into account updates — so that we can run it multiple times without inserting duplicate records in the database.

The code to scrape this site looks as follows:

```
import requests
import dataset
import re
from datetime import datetime
from bs4 import BeautifulSoup
from urllib.parse import urljoin, urlparse

db = dataset.connect('sqlite:///books.db')

base_url = 'http://books.toscrape.com/'

def scrape_books(html_soup, url):
    for book in html_soup.select('article.product_pod'):
        # For now, we'll only store the books url
        book_url = book.find('h3').find('a').get('href')
        book_url = urljoin(url, book_url)
        path = urlparse(book_url).path
```

```
        book_id = path.split('/')[2]
        # Upsert tries to update first and then insert instead
        db['books'].upsert({'book_id' : book_id,
                            'last_seen' : datetime.now()
                            }, ['book_id'])

def scrape_book(html_soup, book_id):
    main = html_soup.find(class_='product_main')
    book = {}
    book['book_id'] = book_id
    book['title'] = main.find('h1').get_text(strip=True)
    book['price'] = main.find(class_='price_color').get_text(strip=True)
    book['stock'] = main.find(class_='availability').get_text(strip=True)
    book['rating'] = ' '.join(main.find(class_='star-rating') \
                        .get('class')).replace('star-rating', '').strip()
    book['img'] = html_soup.find(class_='thumbnail').find('img').get('src')
    desc = html_soup.find(id='product_description')
    book['description'] = ''
    if desc:
        book['description'] = desc.find_next_sibling('p') \
                                  .get_text(strip=True)
    info_table = html_soup.find(string='Product Information').find_
    next('table')
    for row in info_table.find_all('tr'):
        header = row.find('th').get_text(strip=True)
        # Since we'll use the header as a column, clean it a bit
        # to make sure SQLite will accept it
        header = re.sub('[^a-zA-Z]+', '_', header)
        value = row.find('td').get_text(strip=True)
        book[header] = value
    db['book_info'].upsert(book, ['book_id'])

# Scrape the pages in the catalogue
url = base_url
inp = input('Do you wish to re-scrape the catalogue (y/n)? ')
```

```
while True and inp == 'y':
    print('Now scraping page:', url)
    r = requests.get(url)
    html_soup = BeautifulSoup(r.text, 'html.parser')
    scrape_books(html_soup, url)
    # Is there a next page?
    next_a = html_soup.select('li.next > a')
    if not next_a or not next_a[0].get('href'):
        break
    url = urljoin(url, next_a[0].get('href'))

# Now scrape book by book, oldest first
books = db['books'].find(order_by=['last_seen'])
for book in books:
    book_id = book['book_id']
    book_url = base_url + 'catalogue/{}'.format(book_id)
    print('Now scraping book:', book_url)
    r = requests.get(book_url)
    r.encoding = 'utf-8'
    html_soup = BeautifulSoup(r.text, 'html.parser')
    scrape_book(html_soup, book_id)
    # Update the last seen timestamp
    db['books'].upsert({'book_id' : book_id,
                        'last_seen' : datetime.now()
                        }, ['book_id'])
```

Once the script has finished, remember that you can take a look at the database ("books.db") using, for example, "DB Browser for SQLite." Note the use of the dataset's `upsert` method in this example. This method will try to update a record if it exists already (by matching existing records with a list of given field names), or insert a new record otherwise.

9.5 Scraping GitHub Stars

We're going to scrape *https://github.com*, using requests and Beautiful Soup. Our goal is to get, for a given GitHub username, like, for example, *https://github.com/google*, a list of repositories with their GitHub-assigned programming language as well as the number of stars a repository has.

The basic structure of this scraper is quite simple:

```
import requests
from bs4 import BeautifulSoup
import re

session = requests.Session()

url = 'https://github.com/{}'
username = 'google'

r = session.get(url.format(username), params={'page': 1, 'tab':
'repositories'})
html_soup = BeautifulSoup(r.text, 'html.parser')
repos = html_soup.find(class_='repo-list').find_all('li')
for repo in repos:
    name = repo.find('h3').find('a').get_text(strip=True)
    language = repo.find(attrs={'itemprop': 'programmingLanguage'})
    language = language.get_text(strip=True) if language else 'unknown'
    stars = repo.find('a', attrs={'href': re.compile('\/stargazers')})
    stars = int(stars.get_text(strip=True).replace(',', '')) if stars else 0
    print(name, language, stars)
```

Running this will output:

```
sagetv Java 192
ggrc-core Python 233
gapid Go 445
certificate-transparency-rfcs Python 55
mtail Go 936
[...]
```

However, this will fail if we would try to scrape a normal user's page. Google's GitHub account is an enterprise account, which is displayed slightly differently from normal user accounts. You can try this out by setting the "username" variable to "Macuyiko" (one of the authors of this book). We hence need to adjust our code to handle both cases:

```
import requests
from bs4 import BeautifulSoup
import re

session = requests.Session()

url = 'https://github.com/{}'
username = 'Macuyiko'

r = session.get(url.format(username), params={'page': 1, 'tab':
'repositories'})
html_soup = BeautifulSoup(r.text, 'html.parser')

is_normal_user = False
repos_element = html_soup.find(class_='repo-list')
if not repos_element:
    is_normal_user = True
    repos_element = html_soup.find(id='user-repositories-list')

repos = repos_element.find_all('li')
for repo in repos:
    name = repo.find('h3').find('a').get_text(strip=True)
    language = repo.find(attrs={'itemprop': 'programmingLanguage'})
    language = language.get_text(strip=True) if language else 'unknown'
    stars = repo.find('a', attrs={'href': re.compile('\/stargazers')})
    stars = int(stars.get_text(strip=True).replace(',', '')) if stars else 0
    print(name, language, stars)
```

Running this will output:

```
macuyiko.github.io HTML 0
blog JavaScript 1
minecraft-python JavaScript 14
[...]
```

As an extra exercise, try adapting this code to scrape out all pages in case the repositories page is paginated (as is the case for Google's account).

As a final add-on, you'll note that user pages like *https://github.com/Macuyiko?tab=repositories* also come with a short bio, including (in some cases) an e-mail address. However, this e-mail address is only visible once we log in to GitHub. In what follows, we'll try to get out this information as well.

Warning This practice of hunting for a highly starred GitHub profile and extracting the contact information is frequently applied by recruitment firms. This being said, do note that we're now going to log in to GitHub and that we're crossing the boundary between public and private information. Consider this a practice exercise illustrating how you can do so in Python. Keeping the legal aspects in mind, you're advised to only scrape out your own profile information and to not set up this kind of scrapers on a large scale before knowing what you're getting into. Refer back to the chapter on legal concerns for the details regarding the legality of scraping.

You will need to create a GitHub profile in case you haven't done so already. Let us start by getting out the login form from the login page:

```
import requests
from bs4 import BeautifulSoup

session = requests.Session()

url = 'https://github.com/{}'
username = 'Macuyiko'

# Visit the login page
r = session.get(url.format('login'))
html_soup = BeautifulSoup(r.text, 'html.parser')

form = html_soup.find(id='login')
print(form)
```

Running this will output:

```
<div class="auth-form px-3" id="login"> <!-- '"` -->
<!-- </textarea></xmp> --></div>
```

This is not exactly what we expected. If we take a look at the page source, we see that the page is formatted somewhat strangely:

```
<div class="auth-form px-3" id="login">

    <!-- '"` --><!-- </textarea></xmp> --></option></form>

    <form accept-charset="UTF-8" action="/session" method="post">
    <div style="margin:0;padding:0;display:inline">
    <input name="utf8" type="hidden" value="&#x2713;" />
    <input name="authenticity_token" type="hidden" value="AtuMda[...]zw==" />
    </div>

    <div class="auth-form-header p-0">
        <h1>Sign in to GitHub</h1>
    </div>

    <div id="js-flash-container">
</div>
[...]

</form>
```

The following modification makes sure we get out the forms in the page:

```
import requests
from bs4 import BeautifulSoup

session = requests.Session()

url = 'https://github.com/{}'
username = 'Macuyiko'

# Visit the login page
r = session.get(url.format('login'))
html_soup = BeautifulSoup(r.text, 'html.parser')
```

```
data = {}
for form in html_soup.find_all('form'):
    # Get out the hidden form fields
    for inp in form.select('input[type=hidden]'):
        data[inp.get('name')] = inp.get('value')

# SET YOUR LOGIN DETAILS:
data.update({'login': '', 'password': ''})

print('Going to login with the following POST data:')
print(data)

if input('Do you want to login (y/n): ') == 'y':
    # Perform the login
    r = session.post(url.format('session'), data=data)
    # Get the profile page
    r = session.get(url.format(username))
    html_soup = BeautifulSoup(r.text, 'html.parser')
    user_info = html_soup.find(class_='vcard-details')
    print(user_info.text)
```

Even Browsers Have Bugs If you've been using Chrome, you might wonder why you're not seeing the form data when following along with the login process using Chrome's Developer Tools. The reason is that Chrome contains a bug that will prevent form data from appearing in Developer Tools when the status code of the POST corresponds with a redirect. The POST data is still being sent; however, you just won't see it in the Developer Tools tab. This bug will probably be fixed by the time you're reading this, but it just goes to show that bugs appear in browsers as well.

Running this will output:

```
Going to login with the following POST data:
{'utf8': 'V',
 'authenticity_token': 'zgndmzes [...]',
 'login': 'YOUR_USER_NAME',
 'password': 'YOUR_PASSWORD'}
```

```
Do you want to login (y/n): y

KU Leuven

Belgium

macuyiko@gmail.com

http://blog.macuyiko.com
```

Plain Text Passwords It goes without saying that hard-coding your password in plain text in Python files (and other programs, for that matter) is not advisable for real-life scripts. In a real deployment setting, where your code might get shared with others, make sure to modify your script so that it retrieves stored credentials from a secure data store (e.g., from the operating system environment variables, a file, or a database, preferably encrypted). Take a look at the "secureconfig" library available in pip, for example, on how to do so.

9.6 Scraping Mortgage Rates

We're going to scrape Barclays' mortgage simulator available at *https://www.barclays.co.uk/mortgages/mortgage-calculator/*. There isn't a particular reason why we pick this financial services provider, other than the fact that it applies some interesting techniques that serve as a nice illustration.

Take some time to explore the site a bit (using "What would it cost?"). We're asked to fill in a few parameters, after which we get an overview of possible products that we'd like to scrape out.

If you follow along with your browser's developer tools, you'll note that a POST request is being made to *https://www.barclays.co.uk/dss/service/co.uk/mortgages/costcalculator/productservice*, with an interesting property: the JavaScript on the page performing the POST is using an "application/json" value for the "Content-Type" header and is including the POST data as plain JSON; see Figure 9-2. Depending on requests' `data` argument will not work in this case as it will encode the POST data. Instead, we need to use the `json` argument, which will basically instruct requests to format the POST data as JSON.

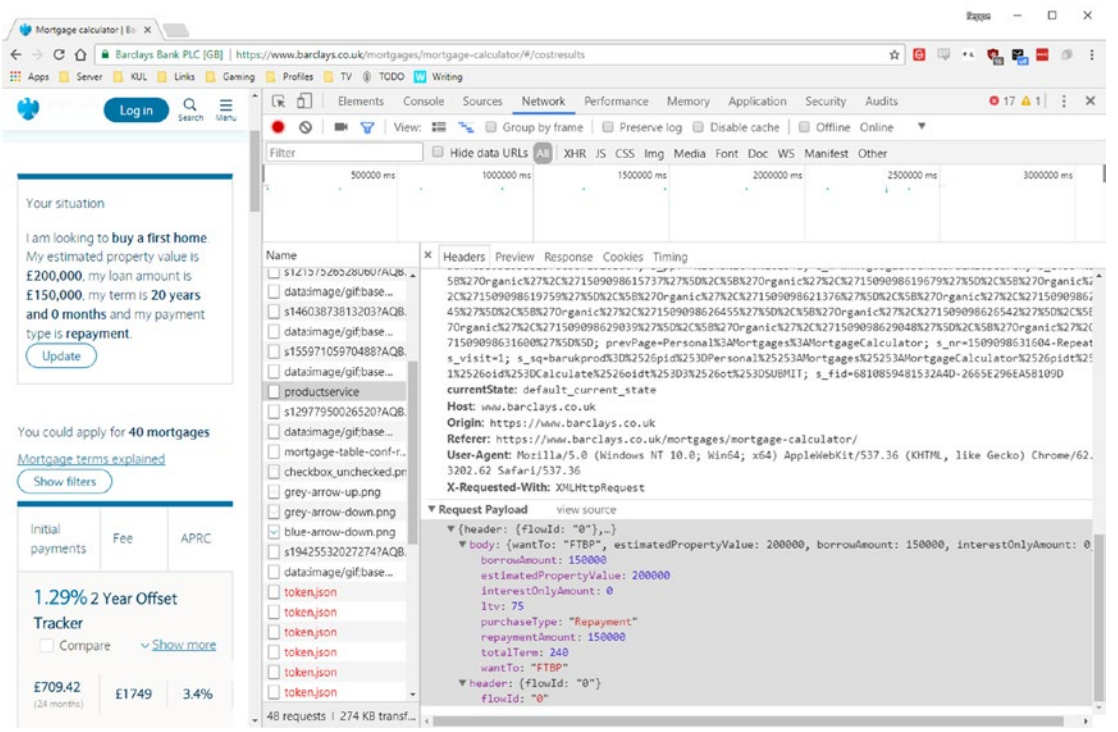

Figure 9-2. *The Barclays mortgage simulator submits a POST request using JavaScript and embeds the request data in a JSON format*

Additionally, you'll note that the result page is formatted as a relatively complex-looking table (with "Show more" links for every entry), though the response returned by the POST request looks like a nicely formatted JSON object; see Figure 9-3, so we might not even need Beautiful Soup here to access this "internal API".

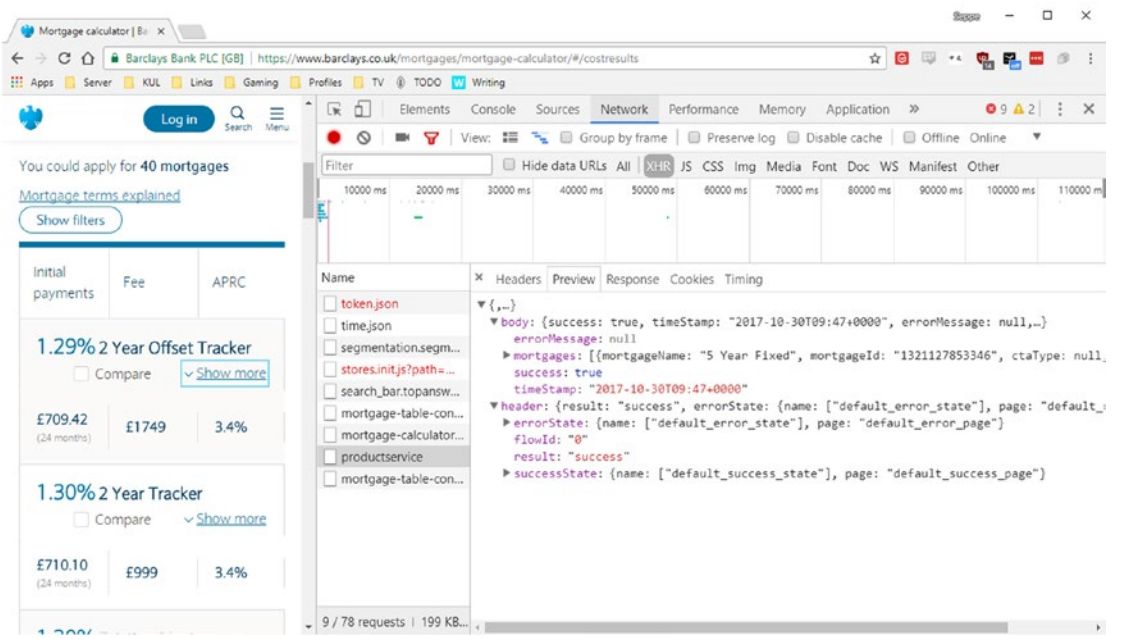

Figure 9-3. *The POST response data also comes back as nicely formatted JSON*

Let's see which response we get by implementing this in Python:

```
import requests

url = 'https://www.barclays.co.uk/dss/service/co.uk/mortgages/' + \
      'costcalculator/productservice'

session = requests.Session()

estimatedPropertyValue = 200000
repaymentAmount = 150000
months = 240
data = {"header": {"flowId":"0"},
        "body":
        {"wantTo":"FTBP",
         "estimatedPropertyValue":estimatedPropertyValue,
         "borrowAmount":repaymentAmount,
         "interestOnlyAmount":0,
```

```
        "repaymentAmount":repaymentAmount,
        "ltv":round(repaymentAmount/estimatedPropertyValue*100),
        "totalTerm":months,
        "purchaseType":"Repayment"}}

r = session.post(url, json=data)

print(r.json())
```

Running this will output:

```
{'header':
{'result': 'error', 'systemError':
  {'errorCode': 'DSS_SEF001', 'type': 'E',
   'severity': 'FRAMEWORK',
   'errorMessage': 'State details not found in database',
   'validationErrors': [],
   'contentType': 'application/json', 'channel': '6'}
}}
```

That doesn't look too good. Remember that, when we don't get back the results we expect, there are various things we can do:

- Check whether we've forgotten to include some cookies. For example, we might need to visit the entry page first, or there might be cookies set by JavaScript. If you inspect the request in your browser, you'll note that there are a lot of cookies present.
- Check whether we've forgotten to include some headers, or whether we need to spoof some.
- If all else fails, resort to Selenium to implement a full browser.

In this particular situation, there are a lot of cookies being included in the request, some of which are set through normal "Set-Cookie" headers, though many are also set through a vast collection of JavaScript files included by the page. These would certainly be hard to figure out, as the JavaScript is obfuscated. There are, however, some interesting headers that are being set and included by JavaScript in the POST request,

which do seem to be connected to the error message. Let's try including these, as well as spoofing the "User-Agent" and "Referer" headers:

```
import requests

url = 'https://www.barclays.co.uk/dss/service/co.uk/mortgages/' + \
      'costcalculator/productservice'

session = requests.Session()

session.headers.update({
    # These are non-typical headers, let's include them
    'currentState': 'default_current_state',
    'action': 'default',
    'Origin': 'https://www.barclays.co.uk',
    # Spoof referer, user agent, and X-Requested-With
    'Referer': 'https://www.barclays.co.uk/mortgages/mortgage-calculator/',
    'User-Agent': 'Mozilla/5.0 (Windows NT 10.0; Win64; x64)
    AppleWebKit/537.36 ' + ' (KHTML, like Gecko) Chrome/62.0.3202.62
    Safari/537.36',
    'X-Requested-With': 'XMLHttpRequest',
    })
estimatedPropertyValue = 200000
repaymentAmount = 150000
months = 240
data = {"header": {"flowId":"0"},
        "body":
        {"wantTo":"FTBP",
         "estimatedPropertyValue":estimatedPropertyValue,
         "borrowAmount":repaymentAmount,
         "interestOnlyAmount":0,
         "repaymentAmount":repaymentAmount,
         "ltv":round(repaymentAmount/estimatedPropertyValue*100),
         "totalTerm":months,
         "purchaseType":"Repayment"}}

r = session.post(url, json=data)
```

```
# Only print the header to avoid text overload
print(r.json()['header'])
```

This seems to work! In this case, it in fact turns out we didn't have to include any cookies at all. We can now clean up this code:

```
import requests

def get_mortgages(estimatedPropertyValue, repaymentAmount, months):
    url = 'https://www.barclays.co.uk/dss/service/' + \
          'co.uk/mortgages/costcalculator/productservice'
    headers = {
        # These are non-typical headers, let's include them
        'currentState': 'default_current_state',
        'action': 'default',
        'Origin': 'https://www.barclays.co.uk',
        # Spoof referer, user agent, and X-Requested-With
        'Referer': 'https://www.barclays.co.uk/mortgages/mortgage-
        calculator/',
        'User-Agent': 'Mozilla/5.0 (Windows NT 10.0; Win64; x64)
        AppleWebKit/537.36 ' + ' (KHTML, like Gecko) Chrome/62.0.3202.62
        Safari/537.36',
        'X-Requested-With': 'XMLHttpRequest',
        }
    data = {"header": {"flowId":"0"},
            "body":
            {"wantTo":"FTBP",
             "estimatedPropertyValue":estimatedPropertyValue,
             "borrowAmount":repaymentAmount,
             "interestOnlyAmount":0,
             "repaymentAmount":repaymentAmount,
             "ltv":round(repaymentAmount/estimatedPropertyValue*100),
             "totalTerm":months,
             "purchaseType":"Repayment"}}
```

```
    r = requests.post(url, json=data, headers=headers)
    results = r.json()
    return results['body']['mortgages']
mortgages = get_mortgages(200000, 150000, 240)

# Print the first mortgage info
print(mortgages[0])
```

Running this will output:

```
{'mortgageName': '5 Year Fixed', 'mortgageId': '1321127853346',
 'ctaType': None, 'uniqueId': '590b357e295b0377d0fb607b',
 'mortgageType': 'FIXED',
 'howMuchCanBeBorrowedNote': '95% (max) of the value of your home',
 'initialRate': 4.99, 'initialRateTitle': '4.99%',
 'initialRateNote': 'until 31st January 2023',
[...]
```

9.7 Scraping and Visualizing IMDB Ratings

The next series of examples moves on toward including some more data science-oriented use cases. We're going to start simple by scraping a list of reviews for episodes of a TV series, using IMDB (the Internet Movie Database). We'll use *Game of Thrones* as an example, the episode list for which can be found at *http://www.imdb.com/title/tt0944947/episodes*. Note that IMDB's overview is spread out across multiple pages (per season or per year), so we iterate over the seasons we want to retrieve using an extra loop:

```
import requests
from bs4 import BeautifulSoup

url = 'http://www.imdb.com/title/tt0944947/episodes'

episodes = []
ratings = []
```

```
# Go over seasons 1 to 7
for season in range(1, 8):
    r = requests.get(url, params={'season': season})
    soup = BeautifulSoup(r.text, 'html.parser')
    listing = soup.find('div', class_='eplist')
    for epnr, div in enumerate(listing.find_all('div', recursive=False)):
        episode = "{}.{}".format(season, epnr + 1)
        rating_el = div.find(class_='ipl-rating-star__rating')
        rating = float(rating_el.get_text(strip=True))
        print('Episode:', episode, '-- rating:', rating)
        episodes.append(episode)
        ratings.append(rating)
```

We can then plot the scraped ratings using "matplotlib," a well-known plotting library for Python that can be easily installed using pip:

```
pip install -U matplotlib
```

Plotting with Python Of course, you could also reproduce the plot below using, for example, Excel, but this example serves as a gentle introduction as we'll continue to use matplotlib for some later examples as well. Note that this is certainly not the only—or even most user-friendly—plotting library for Python, though it remains one of the most prevalent ones. Take a look at Seaborn (*https://seaborn.pydata.org/*), Altair (*https://altair-viz.github.io/*) and ggplot (*http://ggplot.yhathq.com/*) for some other excellent libraries.

Adding in the following lines to our script plots the results in a simple bar chart, as shown in Figure 9-4.

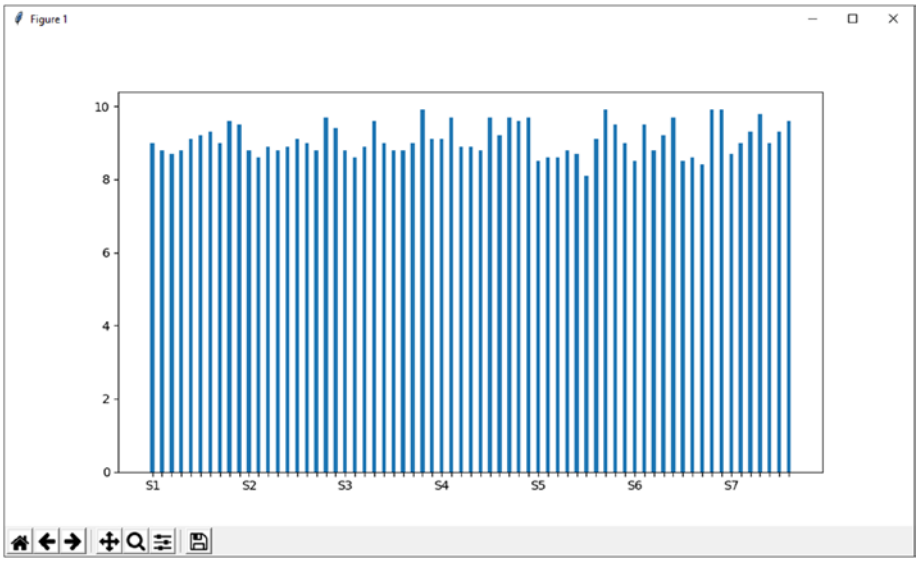

Figure 9-4. *Plotting IMDB ratings per episode using "matplotlib"*

```
import matplotlib.pyplot as plt

episodes = ['S' + e.split('.')[0] if int(e.split('.')[1]) == 1 else '' \
                                  for e in episodes]

plt.figure()
positions = [a*2 for a in range(len(ratings))]
plt.bar(positions, ratings, align='center')
plt.xticks(positions, episodes)
plt.show()
```

9.8 Scraping IATA Airline Information

We're going to scrape airline information using the search form available at *http://www.iata.org/publications/Pages/code-search.aspx*. This is an interesting case to illustrate the "nastiness" of some websites, even though the form we want to use looks incredibly simple (there's only one drop-down and one text field visible on the page). As the URL already shows, the web server driving this page is built on ASP.NET (".aspx"), which has very peculiar opinions about how it handles form data.

It is a good idea to try submitting this form using your browser and taking a look at what happens using its developer tools. As you can see in Figure 9-5, it seems that a lot of form data get included in the POST request — much more than our two fields.

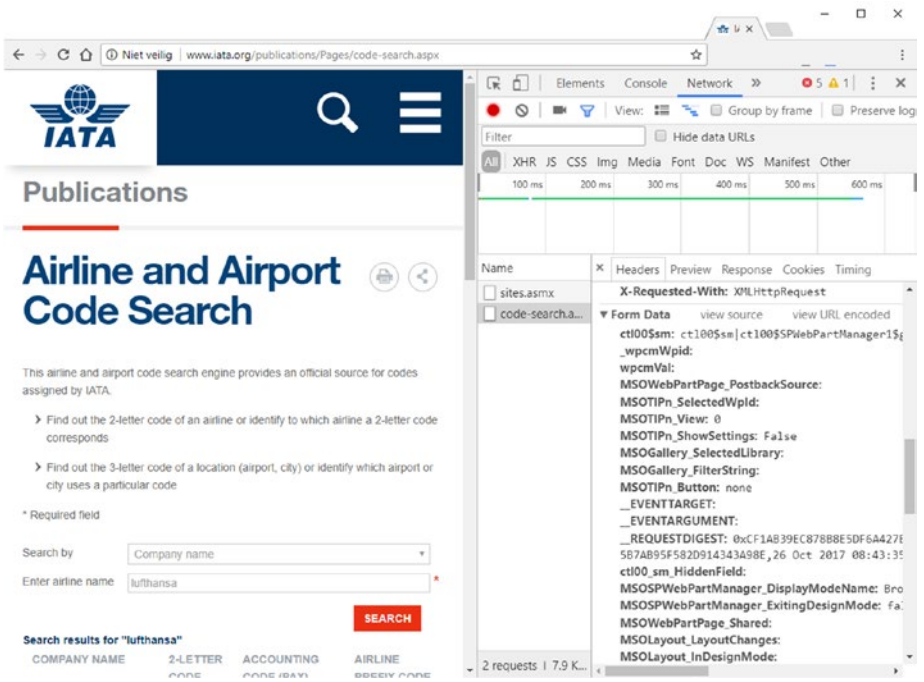

Figure 9-5. *Submitting the IATA form includes lots of form data.*

Certainly, it does not look feasible to manually include all these fields in our Python script. The "__VIEWSTATE" field, for instance, holds session information that changes for every request. Even some names of fields seem to include parts of which we can't really be sure that they wouldn't change in the future, causing our script to break. In addition, it seems that we also need to keep track of cookies as well. Finally, take a look at the response content that comes back from the POST request. This looks like a partial response (which will be parsed and shown by JavaScript) instead of a full HTML page:

```
1|#||4|1330|updatePanel|ctl00_SPWebPartManager1_g_e3b09024_878e
[...]
MSOSPWebPartManager_StartWebPartEditingName|false|5|hiddenField|
MSOSPWebPartManager_EndWebPartEditing|false|
```

To handle these issues, we're going to try to make our code as robust as possible. First, we'll start by performing a GET request to the search page, using requests' sessions mechanism. Next, we'll use Beautiful Soup to get out all the form elements with their names and values:

```
import requests
from bs4 import BeautifulSoup

url = 'http://www.iata.org/publications/Pages/code-search.aspx'

session = requests.Session()
# Spoof the user agent as a precaution
session.headers.update({
    'User-Agent' : 'Mozilla/5.0 (Windows NT 10.0; Win64; x64)
    AppleWebKit/537.36 ' + ' (KHTML, like Gecko) Chrome/62.0.3202.62
    Safari/537.36'
    })

# Get the search page
r = session.get(url)
html_soup = BeautifulSoup(r.text, 'html.parser')
form = html_soup.find(id='aspnetForm')

# Get the form fields
data = {}
for inp in form.find_all(['input', 'select']):
    name = inp.get('name')
    value = inp.get('value')
    if not name:
        continue
    data[name] = value if value else ''

print(data, end='\n\n\n')
```

This will output the following:

```
{'_wpcmWpid': '',
 'wpcmVal': '',
 'MSOWebPartPage_PostbackSource': '',
 'MSOTlPn_SelectedWpId': '',
```

```
 'MSOTlPn_View': '0',
 'MSOTlPn_ShowSettings': 'False',
 'MSOGallery_SelectedLibrary': '',
 'MSOGallery_FilterString': '',
 'MSOTlPn_Button': 'none',
 '__EVENTTARGET': '',
 '__EVENTARGUMENT': '',
[...]
```

Next, we'll use the collected form data to perform a POST request. We do have to make sure to set the correct values for the drop-down and text box, however. We add the following lines to our script:

```
# Set our desired search query
for name in data.keys():
    # Search by
    if 'ddlImLookingFor' in name:
        data[name] = 'ByAirlineName'
    # Airline name
    if 'txtSearchCriteria' in name:
        data[name] = 'Lufthansa'

# Perform a POST
r = session.post(url, data=data)
print(r.text)
```

Strangely enough, contrary to what's happening in the browser, the POST request does return a full HTML page here, instead of a partial result. This is not too bad, as we can now use Beautiful Soup to fetch the table of results.

Instead of parsing this table manually, we'll use a popular data science library for tabular data wrangling called "pandas," which comes with a helpful "HTML table to data frame" method built in. The library is easy to install using pip:

```
pip install -U pandas
```

To parse out HTML, pandas relies on "lxml" by default and falls back to Beautiful Soup with "html5lib" in case "lxml" cannot be found. To make sure "lxml" is available, install it with:

```
pip install -U lxml
```

The full script can now be organized to look as follows:

```
import requests
from bs4 import BeautifulSoup
import pandas

url = 'http://www.iata.org/publications/Pages/code-search.aspx'

def get_results(airline_name):
    session = requests.Session()
    # Spoof the user agent as a precaution
    session.headers.update({
        'User-Agent' : 'Mozilla/5.0 (Windows NT 10.0; Win64; x64)
        AppleWebKit/537.36 ' + ' (KHTML, like Gecko) Chrome/62.0.3202.62
        Safari/537.36'
        })
    r = session.get(url)
    html_soup = BeautifulSoup(r.text, 'html.parser')
    form = html_soup.find(id='aspnetForm')
    data = {}
    for inp in form.find_all(['input', 'select']):
        name = inp.get('name')
        value = inp.get('value')
        if not name:
            continue
        if 'ddlImLookingFor' in name:
            value = 'ByAirlineName'
        if 'txtSearchCriteria' in name:
            value = airline_name
        data[name] = value if value else ''

    r = session.post(url, data=data)
    html_soup = BeautifulSoup(r.text, 'html.parser')
    table = html_soup.find('table', class_='datatable')
    df = pandas.read_html(str(table))
    return df

df = get_results('Lufthansa')
print(df)
```

Running this will output:

```
[                               0   1     2      3
0           Deutsche Lufthansa AG  LH 220.0  220.0
1              Lufthansa Cargo AG  LH   NaN   20.0
2         Lufthansa CityLine GmbH  CL 683.0  683.0
3 Lufthansa Systems GmbH & Co. KG  S1   NaN    NaN]
```

The equivalent Selenium code looks as follows:

```
import pandas
from selenium import webdriver
from selenium.webdriver.support.ui import Select

url = 'http://www.iata.org/publications/Pages/code-search.aspx'

driver = webdriver.Chrome()
driver.implicitly_wait(10)

def get_results(airline_name):
    driver.get(url)
    # Make sure to select the right part of the form
    # This will make finding the elements easier
    # as #aspnetForm wraps the whole page, including
    # the search box
    form_div = driver.find_element_by_css_selector('#aspnetForm
    .iataStandardForm')
    select = Select(form_div.find_element_by_css_selector('select'))
    select.select_by_value('ByAirlineName')
    text = form_div.find_element_by_css_selector('input[type=text]')
    text.send_keys(airline_name)
    submit = form_div.find_element_by_css_selector('input[type=submit]')
    submit.click()
    table = driver.find_element_by_css_selector('table.datatable')
    table_html = table.get_attribute('outerHTML')
    df = pandas.read_html(str(table_html))
    return df
```

```
df = get_results('Lufthansa')
print(df)

driver.quit()
```

There's still one mystery we have to solve: remember that the POST request as made by requests returns a full HTML page, instead of a partial result as we observed in the browser. How does the server figure out how to differentiate between both types of results? The answer lies in the way the search form is submitted. In requests, we perform a simple POST request with a minimal amount of headers. On the live page, however, the form submission is handled by JavaScript, which will perform the actual POST request and will parse out the partial results to show them. To indicate to the server that it is JavaScript making the request, two headers are included in the request, which we can spoof in requests as well. If we modify our code as follows, you will indeed also obtain the same partial result:

```
# Include headers to indicate that we want a partial result
session.headers.update({
    'X-MicrosoftAjax'  : 'Delta=true',
    'X-Requested-With' : 'XMLHttpRequest',
    'User-Agent' : 'Mozilla/5.0 (Windows NT 10.0; Win64; x64)
    AppleWebKit/537.36 ' + ' (KHTML, like Gecko) Chrome/62.0.3202.62
    Safari/537.36'
    })
```

9.9 Scraping and Analyzing Web Forum Interactions

In this example, we're going to scrape web forum posts available at *http://bpbasecamp.freeforums.net/board/27/gear-closet* (a forum for backpackers and hikers) to get an idea about who the most active users are and who is frequently interacting with whom. We're going to keep a tally of interactions that will be constructed as follows:

- The first post in a "thread" is not "replying" to anyone, so we won't consider this as an interaction,
- The next posts in a thread can optionally include one or more quote blocks, which indicate that the poster is directly replying to another user, which we'll regard as such,

- If a post does not include any quote blocks, we'll just assume the post to be a reply to the original poster. This might not necessarily be the case, and users will oftentimes use little pieces of text such as "^^" to indicate they're replying to the direct previous poster, but we're going to keep it simple in this example (feel free to modify the scripts accordingly to your definition of "interaction," however).

Let's get started. First, we're going to extract a list of threads given a forum URL:

```
import requests
import re
from bs4 import BeautifulSoup

def get_forum_threads(url, max_pages=None):
    page = 1
    threads = []
    while not max_pages or page <= max_pages:
        print('Scraping forum page:', page)
        r = requests.get(url, params={'page': page})
        soup = BeautifulSoup(r.text, 'html.parser')
        content = soup.find(class_='content')
        links = content.find_all('a', attrs={'href': re.compile
        ('^\/thread\/')})
        threads_on_page = [a.get('href') for a in links \
                if a.get('href') and not 'page=' in a.get('href')]
        threads += threads_on_page
        page += 1
        next_page = soup.find('li', class_='next')
        if 'state-disabled' in next_page.get('class'):
            break
        return threads

url = 'http://bpbasecamp.freeforums.net/board/27/gear-closet'

threads = get_forum_threads(url, max_pages=5)
print(threads)
```

Note that we have to be a bit clever here regarding pagination. This forum will continue to return the last page, even when supplying higher than maximum page numbers as the URL parameter, so that we can check whether an item with the class "next" also has the class "state-disabled" to determine whether we've reached the end of the thread list. Since we only want thread links corresponding with the first page, we remove all links that have "page=" in their URL as well. In the example, we also decide to limit ourselves to five pages only. Running this will output:

```
Scraping forum page: 1
Scraping forum page: 2
Scraping forum page: 3
Scraping forum page: 4
Scraping forum page: 5
['/thread/2131/before-asking-which-pack-boot', [...] ]
```

For every thread, we now want to get out a list of posts. We can try this out with one thread first:

```
import requests
import re
from urllib.parse import urljoin
from bs4 import BeautifulSoup

def get_thread_posts(url, max_pages=None):
    page = 1
    posts = []
    while not max_pages or page <= max_pages:
        print('Scraping thread url/page:', url, page)
        r = requests.get(url, params={'page': page})
        soup = BeautifulSoup(r.text, 'html.parser')
        content = soup.find(class_='content')
        for post in content.find_all('tr', class_='item'):
            user = post.find('a', class_='user-link')
            if not user:
                # User might be deleted, skip...
                continue
```

```
            user = user.get_text(strip=True)
            quotes = []
            for quote in post.find_all(class_='quote_header'):
                quoted_user = quote.find('a', class_='user-link')
                if quoted_user:
                    quotes.append(quoted_user.get_text(strip=True))
            posts.append((user, quotes))
        page += 1
        next_page = soup.find('li', class_='next')
        if 'state-disabled' in next_page.get('class'):
            break
    return posts
url = 'http://bpbasecamp.freeforums.net/board/27/gear-closet'
thread = '/thread/2131/before-asking-which-pack-boot'

thread_url = urljoin(url, thread)
posts = get_thread_posts(thread_url)
print(posts)
```

Running this will output a list with every element being a tuple containing the poster's name and a list of users that are quoted in the post:

```
Scraping thread url/page:                                              ↩
    http://bpbasecamp.freeforums.net/thread/2131/before-asking-which-pack-boot 1
Scraping thread url/page:                                              ↩
    http://bpbasecamp.freeforums.net/thread/2131/before-asking-which-pack-boot 2
[('almostthere', []), ('trinity', []), ('paula53', []),                ↩
    ('toejam', ['almostthere']), (‘stickman', []), (‘tamtrails', []),  ↩
    ('almostthere', ['tamtrails']), ('kayman', []), (‘almostthere',    ↩
    [‘kayman']), (‘lanceman', []), (‘trinity', [‘trinity']),           ↩
     (‘Christian', [‘almostthere']), (‘pollock', []), (‘mitsmit', []), ↩
    ('intothewild', []), (‘Christian', []), (‘softskull', []), (‘argus',  ↩
    []),(‘lyssa7', []), (‘kevin', []), (‘greenwoodsuncharted', [])]
```

By putting both of these functions together, we get the script below. We'll use Python's "pickle" module to store our scraped results so that we don't have to rescrape the forum over and over again:

```
import requests
import re
from urllib.parse import urljoin
from bs4 import BeautifulSoup
import pickle

def get_forum_threads(url, max_pages=None):
    page = 1
    threads = []
    while not max_pages or page <= max_pages:
        print('Scraping forum page:', page)
        r = requests.get(url, params={'page=': page})
        soup = BeautifulSoup(r.text, 'html.parser')
        content = soup.find(class_='content')
        links = content.find_all('a', attrs={'href': re.compile
        ('^\/thread\/')})
        threads_on_page = [a.get('href') for a in links \
                if a.get('href') and not 'page' in a.get('href')]
        threads += threads_on_page
        page += 1
        next_page = soup.find('li', class_='next')
        if 'state-disabled' in next_page.get('class'):
            break
    return threads
def get_thread_posts(url, max_pages=None):
    page = 1
    posts = []
```

```
    while not max_pages or page <= max_pages:
        print('Scraping thread url/page:', url, page)
        r = requests.get(url, params={'page': page})
        soup = BeautifulSoup(r.text, 'html.parser')
        content = soup.find(class_='content')
        for post in content.find_all('tr', class_='item'):
            user = post.find('a', class_='user-link')
            if not user:
                # User might be deleted, skip...
                continue
            user = user.get_text(strip=True)
            quotes = []
            for quote in post.find_all(class_='quote_header'):
                quoted_user = quote.find('a', class_='user-link')
                if quoted_user:
                    quotes.append(quoted_user.get_text(strip=True))
            posts.append((user, quotes))
        page += 1
        next_page = soup.find('li', class_='next')
        if 'state-disabled' in next_page.get('class'):
            break
    return posts

url = 'http://bpbasecamp.freeforums.net/board/27/gear-closet'

threads = get_forum_threads(url, max_pages=5)
all_posts = []

for thread in threads:
    thread_url = urljoin(url, thread)
    posts = get_thread_posts(thread_url)
    all_posts.append(posts)

with open('forum_posts.pkl', "wb") as output_file:
    pickle.dump(all_posts, output_file)
```

Next, we can load the results and visualize them in a heat map. We're going to use "pandas," "numpy," and "matplotlib" to do so, all of which can be installed through pip (if you've already installed pandas and matplotlib by following the previous examples, there's nothing else you have to install):

```
pip install -U pandas
pip install -U numpy
pip install -U matplotlib
```

Let's start by visualizing the first thread only (shown in the output fragment of the scraper above):

```
import pickle
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

# Load our stored results
with open('forum_posts.pkl', "rb") as input_file:
    posts = pickle.load(input_file)

def add_interaction(users, fu, tu):
    if fu not in users:
        users[fu] = {}
    if tu not in users[fu]:
        users[fu][tu] = 0
    users[fu][tu] += 1

# Create interactions dictionary
users = {}
for thread in posts:
    first_one = None
    for post in thread:
        user = post[0]
        quoted = post[1]
        if not first_one:
            first_one = user
        elif not quoted:
```

```
            add_interaction(users, user, first_one)
        else:
            for qu in quoted:
                add_interaction(users, user, qu)
    # Stop after the first thread
    break

df = pd.DataFrame.from_dict(users, orient='index').fillna(0)

heatmap = plt.pcolor(df, cmap='Blues')
y_vals = np.arange(0.5, len(df.index), 1)
x_vals = np.arange(0.5, len(df.columns), 1)
plt.yticks(y_vals, df.index)
plt.xticks(x_vals, df.columns, rotation='vertical')
for y in range(len(df.index)):
    for x in range(len(df.columns)):
        if df.iloc[y, x] == 0:
            continue
        plt.text(x + 0.5, y + 0.5, '%.0f' % df.iloc[y, x],
                 horizontalalignment='center',
                 verticalalignment='center')
plt.show()
```

This will provide you with a result as shown in Figure 9-6. As you can see, various users are replying to the original poster, and the original poster is also quoting some other users.

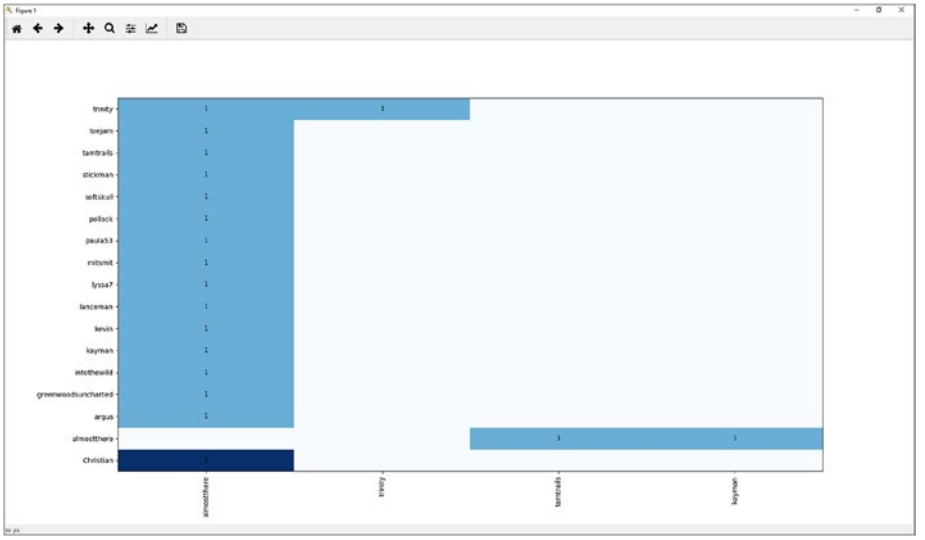

Figure 9-6. *Visualizing user interactions for one forum thread*

There are various ways to play around with this visualization. Figure 9-7, for instance, shows the user interactions over all forum threads, but only taking into account direct quotes.

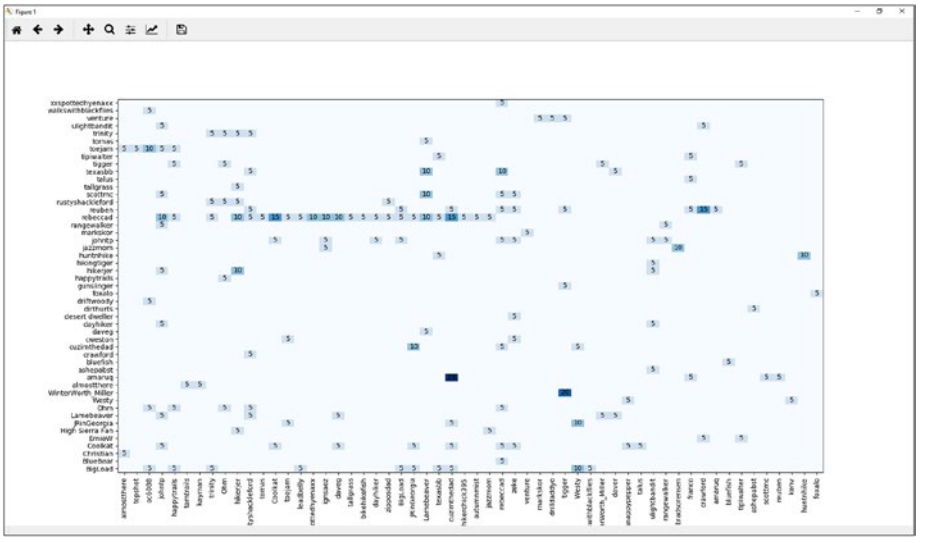

Figure 9-7. *Visualizing user interactions (direct quotes only) for all scraped forum threads*

9.10 Collecting and Clustering a Fashion Data Set

In this example, we're going to use Zalando (a popular Swedish web shop) to fetch a collection of images of fashion products and cluster them using t-SNE.

Check the API Note that Zalando also exposes an easy to use API (see *https://github.com/zalando/shop-api-documentation/wiki/Api-introduction* for the documentation). At the time of writing, the API does not require authentication, though this is scheduled to change in the near future, requiring users to register to get an API access token. Since we'll only fetch images here, we'll not bother to register, though in a proper "app," using the API option would certainly be recommended.

Our first script downloads images and stores them in a directory; see Figure 9-8:

```
import requests
import os, os.path
from bs4 import BeautifulSoup
from urllib.parse import urljoin, urlparse

store = 'images'
if not os.path.exists(store):
    os.makedirs(store)

url = 'https://www.zalando.co.uk/womens-clothing-dresses/'
pages_to_crawl = 15

def download(url):
    r = requests.get(url, stream=True)
    filename = urlparse(url).path.split('/')[-1]
    print('Downloading to:', filename)
    with open(os.path.join(store, filename), 'wb') as the_image:
        for byte_chunk in r.iter_content(chunk_size=4096*4):
            the_image.write(byte_chunk)
```

```
for p in range(1, pages_to_crawl+1):
    print('Scraping page:', p)
    r = requests.get(url, params={'p' : p})
    html_soup = BeautifulSoup(r.text, 'html.parser')
    for img in html_soup.select('#z-nvg-cognac-root z-grid-item img'):
        img_src = img.get('src')
        if not img_src:
            continue
        img_url = urljoin(url, img_src)
        download(img_url)
```

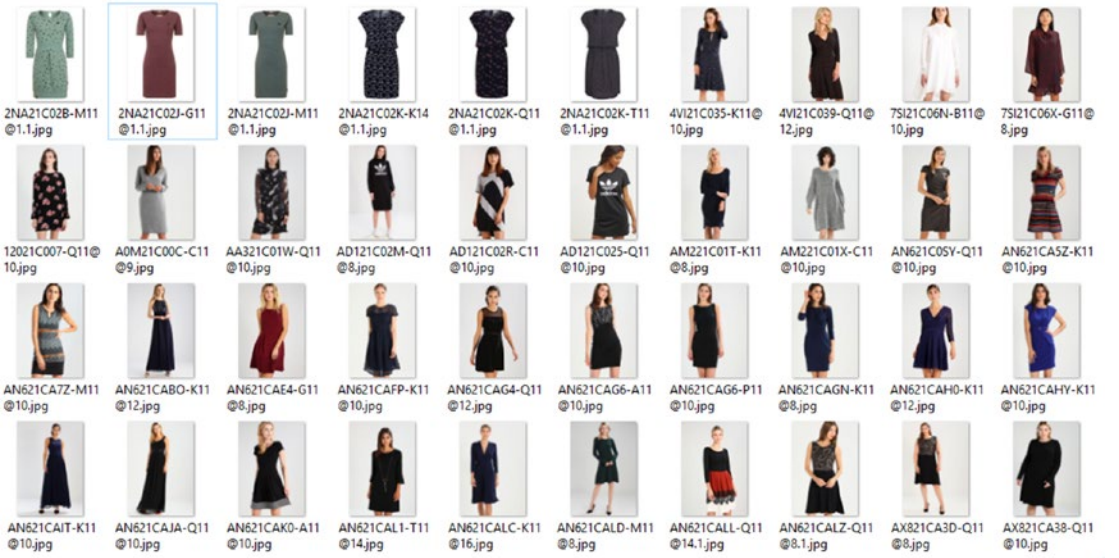

Figure 9-8. *A collection of scraped dress images*

Next, we'll use the t-SNE clustering algorithm to cluster the photos. t-SNE is a relatively recent dimensionality reduction technique that is particularly well-suited for the visualization of high-dimensional data sets, like images. You can read about the technique at *https://lvdmaaten.github.io/tsne/*. We're going to use "scikit-learn" together with "matplotlib," "scipy," and "numpy," all of which are libraries that are familiar to data scientists and can be installed through pip:

```
pip install -U matplotlib
pip install -U scikit-learn
pip install -U numpy
pip install -U scipy
```

Our clustering script looks as follows:

```
import os.path
import numpy as np
import matplotlib.pyplot as plt
from matplotlib import offsetbox
from sklearn import manifold
from scipy.misc import imread
from glob import iglob

store = 'images'

image_data = []
for filename in iglob(os.path.join(store, '*.jpg')):
     image_data.append(imread(filename))

image_np_orig = np.array(image_data)
image_np = image_np_orig.reshape(image_np_orig.shape[0], -1)

def plot_embedding(X, image_np_orig):
    # Rescale
    x_min, x_max = np.min(X, 0), np.max(X, 0)
    X = (X - x_min) / (x_max - x_min)
    # Plot images according to t-SNE position
    plt.figure()
    ax = plt.subplot(111)
    for i in range(image_np.shape[0]):
        imagebox = offsetbox.AnnotationBbox(
            offsetbox=offsetbox.OffsetImage(image_np_orig[i], zoom=.1),
            xy=X[i],
            frameon=False)
        ax.add_artist(imagebox)
```

```
print("Computing t-SNE embedding")

tsne = manifold.TSNE(n_components=2, init='pca')
X_tsne = tsne.fit_transform(image_np)

plot_embedding(X_tsne, image_np_orig)
plt.show()
```

This code works as follows. First, we load all the images (using `imread`) and convert them to a numpy array. The `reshape` function makes sure that we get a *n* x *3m* matrix, with *n* the number of images and *m* the number of pixels per image, instead of an *n* x *r* x *g* x *b* tensor, with *r*, *g*, and *b* the pixel values for the red, green, and blue channels respectively. After constructing the t-SNE embedding, we plot the images with their calculated *x* and *y* coordinates using matplotlib, resulting in an image like shown in Figure 9-9 (using about a thousand scraped photos). As can be seen, the clustering here is primarily driven by the saturation and intensity of the colors in the image.

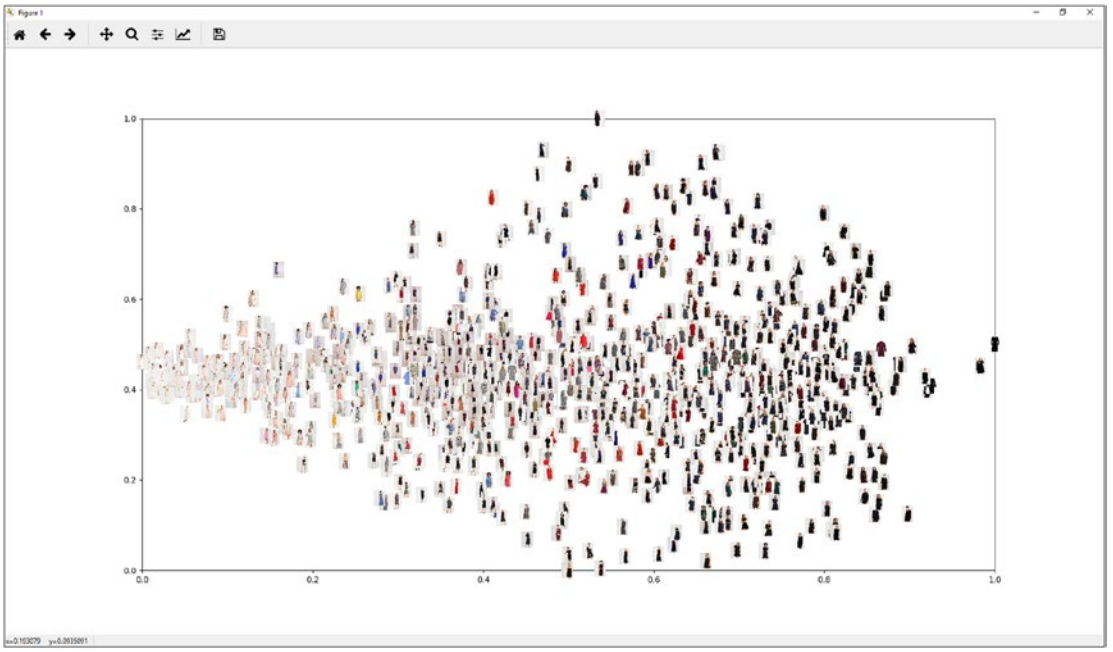

Figure 9-9. *The result of the t-SNE clustering (applied on about a thousand photos)*

Image Sizes We're lucky that all of the images we've scraped have the same width and height. If this would not be the case, we'd first have to apply a resizing to make sure every image will lead to a vector in the data set with equal length.

9.11 Sentiment Analysis of Scraped Amazon Reviews

We're going to scrape a list of Amazon reviews with their ratings for a particular product. We'll use a book with plenty of reviews, say *Learning Python by Mark Lutz*, which can be found at *https://www.amazon.com/Learning-Python-5th-Mark-Lutz/dp/1449355730/*. If you click through "See all customer reviews," you'll end up at *https://www.amazon.com/Learning-Python-5th-Mark-Lutz/product-reviews/1449355730/*. Note that this product has an id of "1449355730," and even using the URL *https://www.amazon.com/product-reviews/1449355730/*, without the product name, will work.

Simple URLs Playing around with URLs as we do here is always a good idea before writing your web scraper. Based on the above, we know that a given product identifier is enough to fetch the reviews page, without a need to figure out the exact URL, including the product name. Why then, does Amazon allow for both and does it default to including the product name? The reason is most likely search engine optimization (SEO). Search engines like Google prefer URLs with human-readable components included.

If you explore the reviews page, you'll note that the reviews are paginated. By browsing to other pages and following along in your browser's developer tools, we see that POST requests are being made (by JavaScript) to URLs looking like *https://www.amazon.com/ss/customer-reviews/ajax/reviews/get/ref=cm_cr_arp_d_paging_btm_2*, with the product id included in the form data, as well as some other form fields that look relatively easy to spoof. Let's see what we get in requests:

```
import requests
from bs4 import BeautifulSoup

review_url = 'https://www.amazon.com/ss/customer-reviews/ajax/reviews/get/'
product_id = '1449355730'
```

```
session = requests.Session()
session.headers.update({
    'User-Agent' : 'Mozilla/5.0 (Windows NT 10.0; Win64; x64)
    AppleWebKit/537.36 ' + ' (KHTML, like Gecko) Chrome/62.0.3202.62
    Safari/537.36'
    })

session.get('https://www.amazon.com/product-reviews/{}/'.format(product_id))

def get_reviews(product_id, page):
    data = {
        'sortBy':'',
        'reviewerType':'all_reviews',
        'formatType':'',
        'mediaType':'',
        'filterByStar':'all_stars',
        'pageNumber':page,
        'filterByKeyword':'',
        'shouldAppend':'undefined',
        'deviceType':'desktop',
        'reftag':'cm_cr_getr_d_paging_btm_{}'.format(page),
        'pageSize':10,
        'asin':product_id,
        'scope':'reviewsAjax1'
        }
    r = session.post(review_url + 'ref=' + data['reftag'], data=data)
    return r.text

print(get_reviews(product_id, 1))
```

Note that we spoof the "User-Agent" header here. If we don't, Amazon will reply with a message requesting us to verify whether we're a human (you can copy the value for this header from your browser's developer tools). In addition, note the "scope" form field that we set to "reviewsAjax1." If you explore the reviews page in the browser, you'll see that the value of this field is in fact increased for each request, that is, "reviewsAjax1," "reviewsAjax2," and so on. We could decide to replicate this behavior as well — which we'd have to do in case Amazon would pick up on our tactics, though it does not seem to be necessary for the results to come back correctly.

Finally, note that the POST request does not return a full HTML page, but some kind of hand-encoded result that will be parsed (normally) by JavaScript:

```
["script",
 "if(window.ue) { ues('id','reviewsAjax1','FE738GN7GRDZK6QO9S9G');
 ues('tO','reviewsAjax1',new Date());
 ues('ctb','reviewsAjax1','1');
 uet('bb','reviewsAjax1'); }"
]
&&&
["update","#cm_cr-review_list",""]
&&&
["loaded"]
&&&
["append","#cm_cr-review_list","<div id=\"R3JQXR4EMWJ7AD\" data-
      hook=\"review\"class=\"a-section review\"><div id=\                ↵
     "customer_review-R3JQXR4EMWJ7AD\"class=\"a-section celwidget\">     ↵
     <div class=\"a-row\"><a class=\"a-link-normal\"title=\"5.0 out      ↵
     of 5 stars\"
[...]
```

Luckily, after exploring the reply a bit (feel free to copy-paste the full reply in a text editor and read through it), the structure seems easy enough to figure out:

- The reply is composed of several "instructions," formatted as a JSON list;
- The instructions themselves are separated by three ampersands, "&&&";
- The instructions containing the reviews start with an "append" string;
- The actual contents of the review are formatted as an HTML element and found on the third position of the list.

Let's adjust our code to parse the reviews in a structured format. We'll loop through all the instructions; convert them using the "json" module; check for "append" entries; and then use Beautiful Soup to parse the HTML fragment and get the review id, rating,

title, and text. We'll also need a small regular expression to get out the rating, which is set as a class with a value like "a-start-1" to "a-star-5". We could use these as is, but simply getting "1" to "5" might be easier to work with later on, so we already perform a bit of cleaning here:

```
import requests
import json
import re
from bs4 import BeautifulSoup

review_url = 'https://www.amazon.com/ss/customer-reviews/ajax/reviews/get/'
product_id = '1449355730'

session = requests.Session()
session.headers.update({
    'User-Agent' : 'Mozilla/5.0 (Windows NT 10.0; Win64; x64)
    AppleWebKit/537.36 ' + ' (KHTML, like Gecko) Chrome/62.0.3202.62
    Safari/537.36'
    })

session.get('https://www.amazon.com/product-reviews/{}/'.format(product_id))

def parse_reviews(reply):
    reviews = []
    for fragment in reply.split('&&&'):
        if not fragment.strip():
            continue
        json_fragment = json.loads(fragment)
        if json_fragment[0] != 'append':
            continue
        html_soup = BeautifulSoup(json_fragment[2], 'html.parser')
        div = html_soup.find('div', class_='review')
        if not div:
            continue
        review_id = div.get('id')
        title = html_soup.find(class_='review-title').get_text(strip=True)
        review = html_soup.find(class_='review-text').get_text(strip=True)
```

```
        # Find and clean the rating:
        review_cls = ' '.join(html_soup.find(class_='review-rating').
        get('class'))
        rating = re.search('a-star-(\d+)', review_cls).group(1)
        reviews.append({'review_id': review_id,
                        'rating': rating,
                        'title': title,
                        'review': review})
    return reviews

def get_reviews(product_id, page):
    data = {
        'sortBy':'',
        'reviewerType':'all_reviews',
        'formatType':'',
        'mediaType':'',
        'filterByStar':'all_stars',
        'pageNumber':page,
        'filterByKeyword':'',
        'shouldAppend':'undefined',
        'deviceType':'desktop',
        'reftag':'cm_cr_getr_d_paging_btm_{}'.format(page),
        'pageSize':10,
        'asin':product_id,
        'scope':'reviewsAjax1'
        }
    r = session.post(review_url + 'ref=' + data['reftag'], data=data)
    reviews = parse_reviews(r.text)
    return reviews

print(get_reviews(product_id, 1))
```

This works! The only thing left to do is to loop through all the pages, and store the reviews in a database using the "dataset" library. Luckily, figuring out when to stop looping is easy: once we do not get any reviews for a particular page, we can stop:

```
import requests
import json
import re
from bs4 import BeautifulSoup
import dataset

db = dataset.connect('sqlite:///reviews.db')

review_url = 'https://www.amazon.com/ss/customer-reviews/ajax/reviews/get/'
product_id = '1449355730'

session = requests.Session()
session.headers.update({
    'User-Agent' : 'Mozilla/5.0 (Windows NT 10.0; Win64; x64)
    AppleWebKit/537.36 ' + ' (KHTML, like Gecko) Chrome/62.0.3202.62
    Safari/537.36'
    })

session.get('https://www.amazon.com/product-reviews/{}/'.format(product_id))

def parse_reviews(reply):
    reviews = []
    for fragment in reply.split('&&&'):
        if not fragment.strip():
            continue
        json_fragment = json.loads(fragment)
        if json_fragment[0] != 'append':
            continue
        html_soup = BeautifulSoup(json_fragment[2], 'html.parser')
        div = html_soup.find('div', class_='review')
        if not div:
            continue
        review_id = div.get('id')
        review_cls = ' '.join(html_soup.find(class_='review-rating').
        get('class'))
        rating = re.search('a-star-(\d+)', review_cls).group(1)
        title = html_soup.find(class_='review-title').get_text(strip=True)
        review = html_soup.find(class_='review-text').get_text(strip=True)
```

```
            reviews.append({'review_id': review_id,
                            'rating': rating,
                            'title': title,
                            'review': review})
        return reviews

    def get_reviews(product_id, page):
        data = {
            'sortBy':'',
            'reviewerType':'all_reviews',
            'formatType':'',
            'mediaType':'',
            'filterByStar':'all_stars',
            'pageNumber':page,
            'filterByKeyword':'',
            'shouldAppend':'undefined',
            'deviceType':'desktop',
            'reftag':'cm_cr_getr_d_paging_btm_{}'.format(page),
            'pageSize':10,
            'asin':product_id,
            'scope':'reviewsAjax1'
            }
        r = session.post(review_url + 'ref=' + data['reftag'], data=data)
        reviews = parse_reviews(r.text)
        return reviews

    page = 1
    while True:
        print('Scraping page', page)
        reviews = get_reviews(product_id, page)
        if not reviews:
            break
        for review in reviews:
            print(' -', review['rating'], review['title'])
            db['reviews'].upsert(review, ['review_id'])
        page += 1
```

This will output the following:

```
Scraping page 1
  - 5 let me try to explain why this 1600 page book may actually end ↩
      up saving you a lot of time and making you a better Python progra
  - 5 Great start, and written for the novice
  - 5 Best teacher of software development
  - 5 Very thorough
  - 5 If you like big thick books that deal with a lot of ...
  - 5 Great book, even for the experienced python programmer
  - 5 Good Tutorial; you'll learn a lot.
  - 2 Takes too many pages to explain even the most simpliest ...
  - 3 If I had a quarter for each time he says something like "here's ↩
      an intro to X
  - 4 it almost seems better suited for a college class
[...]
```

Now that we have a database containing the reviews, let's do something fun with these. We'll run a sentiment analysis algorithm over the reviews (providing a sentiment score per review), which we can then plot over the different ratings given to inspect the correlation between a rating and the sentiment in the text. To do so, we'll use the "vaderSentiment" library, which can simply be installed using pip. We'll also need to install the "nltk" (Natural Language Toolkit) library:

```
pip install -U vaderSentiment
pip install -U nltk
```

Using the vaderSentiment library is pretty simple for a single sentence:

```
from vaderSentiment.vaderSentiment import SentimentIntensityAnalyzer

analyzer = SentimentIntensityAnalyzer()

sentence = "I'm really happy with my purchase"
vs = analyzer.polarity_scores(sentence)

print(vs)
# Shows: {'neg': 0.0, 'neu': 0.556, 'pos': 0.444, 'compound': 0.6115}
```

To get the sentiment for a longer piece of text, a simple approach is to calculate the sentiment score per sentence and average this over all the sentences in the text, like so:

```
from vaderSentiment.vaderSentiment import SentimentIntensityAnalyzer
from nltk import tokenize

analyzer = SentimentIntensityAnalyzer()

paragraph = """
    I'm really happy with my purchase.
    I've been using the product for two weeks now.
    It does exactly as described in the product description.
    The only problem is that it takes a long time to charge.
    However, since I recharge during nights, this is something I can
    live with.
    """

sentence_list = tokenize.sent_tokenize(paragraph)
cumulative_sentiment = 0.0
for sentence in sentence_list:
    vs = analyzer.polarity_scores(sentence)
    cumulative_sentiment += vs["compound"]
    print(sentence, ' : ', vs["compound"])

average_sentiment = cumulative_sentiment / len(sentence_list)
print('Average score:', average_score)
```

If you run this code, ntlk will most likely complain about the fact that a resource is missing:

```
Resource punkt not found.
  Please use the NLTK Downloader to obtain the resource:

  >>> import nltk
  >>> nltk.download('punkt')
[...]
```

To fix this, execute the recommended commands on a Python shell:

```
>>> import nltk
>>> nltk.download('punkt')
```

After the resource has been downloaded and installed, the code above should work fine and will output:

```
I'm really happy with my purchase. : 0.6115
I've been using the product for two weeks now. : 0.0
It does exactly as described in the product description. : 0.0
The only problem is that it takes a long time to charge. : -0.4019
However, since I recharge during nights, this is something I can live
with. : 0.0
Average score: 0.04192000000000001
```

Let's apply this to our list of Amazon reviews. We'll calculate the sentiment for each rating, organize them by rating, and then use the "matplotlib" library to draw violin plots of the sentiment scores per rating:

```
from vaderSentiment.vaderSentiment import SentimentIntensityAnalyzer
from nltk import tokenize
import dataset
import matplotlib.pyplot as plt

db = dataset.connect('sqlite:///reviews.db')
reviews = db['reviews'].all()

analyzer = SentimentIntensityAnalyzer()

sentiment_by_stars = [[] for r in range(1,6)]

for review in reviews:
    full_review = review['title'] + '. ' + review['review']
    sentence_list = tokenize.sent_tokenize(full_review)
    cumulative_sentiment = 0.0
    for sentence in sentence_list:
        vs = analyzer.polarity_scores(sentence)
        cumulative_sentiment += vs["compound"]
    average_score = cumulative_sentiment / len(sentence_list)
    sentiment_by_stars[int(review['rating'])-1].append(average_score)
```

```
plt.violinplot(sentiment_by_stars,
               range(1,6),
               vert=False, widths=0.9,
               showmeans=False, showextrema=True, showmedians=True,
               bw_method='silverman')
plt.axvline(x=0, linewidth=1, color='black')
plt.show()
```

This should output a figure similar to the one shown in Figure 9-10. In this case, we can indeed observe a strong correlation between the rating and the sentiments of the texts, though it's interesting to note that even for lower ratings (two and three stars), the majority of reviews are still somewhat positive. Of course, there is a lot more that can be done with this data set. Think, for instance, about a predictive model to detect fake reviews.

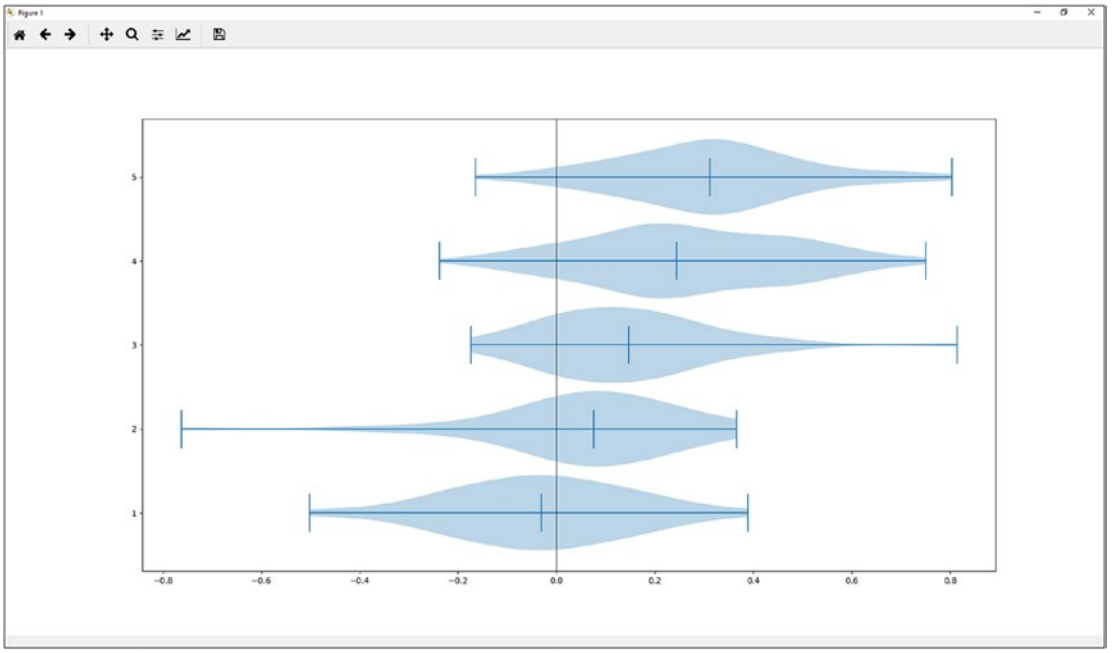

Figure 9-10. *Sentiment plots per rating level*

9.12 Scraping and Analyzing News Articles

We're going to use Selenium to scrape the "Top Stories" from Google News, see *https://news.google.com/news/?ned=us&hl=en*. Our goal is to visit every article and get out the title and main content of the article.

Not as Easy as It Looks Getting out the "main content" from a page is trickier as it might seem at first sight. You might try to iterate all the lowest-level HTML elements and keeping the one with the most text embedded in it, though this approach will break if the text in an article is split up over multiple sibling elements, like a series of "<p>" tags inside a larger " ", for instance. Considering all elements does not resolve this issue, as you'll end up by simply selecting the top element (e.g., "<html>" or "<body>") on the page, as this will always contain the largest amount (i.e., all) text. The same holds in case you'd rely on the `rect` attribute Selenium provides to apply a visual approach (i.e., find the element taking up the most space on the page). A large number of libraries and tools have been written to solve this issue. Take a look at, for example, *https://github.com/masukomi/ar90-readability*, *https://github.com/misja/python-boilerpipe*, *https://github.com/codelucas/newspaper* and *https://github.com/fhamborg/news-please* for some interesting libraries for the specific purpose of news extraction, or specialized APIs such as *https://newsapi.org/* and *https://webhose.io/news-api*. In this example, we'll use Mozilla's implementation of Readability; see *https://github.com/mozilla/readability*. This is a JavaScript-based library, but we'll figure out a way to use it with Python and Selenium nonetheless. Finally, although it has sadly fallen a bit out of use in recent years, it is interesting to know that there exists already a nice format that sites can apply to offer their content updates in a structured way: RSS (Rich Site Summary): a web feed that allows users to access updates to online content in a standardized, XML-based format. Keep an eye out for "<link>" tags with their "type" attribute set to "application/rss+xml". The "href" attribute will then announce the URL where the RSS feed can be found.

Let's start by getting out a list of "Top Stories" links from Google News using Selenium. A first iteration of our script looks as follows:

```
from selenium import webdriver

base_url = 'https://news.google.com/news/?ned=us&hl=en'

driver = webdriver.Chrome()
driver.implicitly_wait(10)
driver.get(base_url)

for link in driver.find_elements_by_css_selector('main a[role="heading"]'):
    news_url = link.get_attribute('href')
    print(news_url)

driver.quit()
```

This will output the following (of course, your links might vary):

```
http://news.xinhuanet.com/english/2017-10/24/c_136702615.htm
http://www.cnn.com/2017/10/24/asia/china-xi-jinping-thought/index.html
[...]
```

Navigate to *http://edition.cnn.com/2017/10/24/asia/china-xi-jinping-thought/index.html* in your browser and open your browser's console in its developer tools. Our goal is now to extract the content from this page, using Mozilla's Readability implementation in JavaScript, a tool which is normally used to display articles in a more readable format. That is, we would like to "inject" the JavaScript code available at *https://raw.githubusercontent.com/mozilla/readability/master/Readability.js* in the page. Since we are able to instruct the browser to execute JavaScript using Selenium, we hence need to come up with an appropriate piece of JavaScript code to perform this injection. Using your browser's console, try executing the following block of code:

```
(function(d, script) {
  script = d.createElement('script');
  script.type = 'text/javascript';
  script.async = true;
  script.onload = function(){
    console.log('The script was successfully injected!');
  };
```

```
  script.src = 'https://raw.githubusercontent.com/' +
        'mozilla/readability/master/Readability.js';
  d.getElementsByTagName('head')[0].appendChild(script);
}(document));
```

This script works as follows: a new "<script>" element is constructed with its "src" parameter set to *https://raw.githubusercontent.com/mozilla/readability/master/Readability.js*, and appended into the "<head>" of the document. Once the script has loaded, we show a message on the console. This will provide you with a result as shown in Figure 9-11.

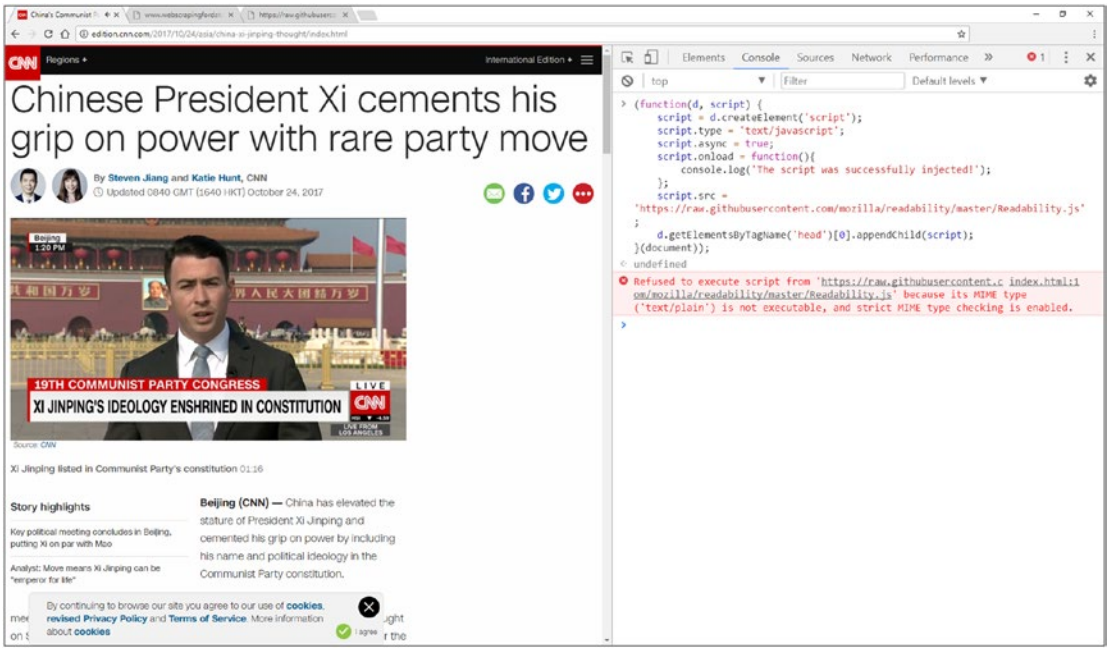

Figure 9-11. *Trying to inject a "<script>" tag using JavaScript*

This does not work as we had expected, as Chrome refuses to execute this script:

```
Refused to execute script from
'https://raw.githubusercontent.com/mozilla/readability/master/Readability.js'
because its MIME type ('text/plain') is not executable, and strict MIME ↩
type checking  is enabled.
```

The problem here is that GitHub indicates in its headers that the content type of this document is "text/plain," and Chrome prevents us from using it as a script. To work around this issue, we'll host a copy of the script ourselves at *http://www.webscrapingfordatascience.com/readability/Readability.js* and try again:

```
(function(d, script) {
  script = d.createElement('script');
  script.type = 'text/javascript';
  script.async = true;
  script.onload = function(){
    console.log('The script was successfully injected!');
  };
  script.src = 'http://www.webscrapingfordatascience.com/readability/
  Readability.js';
  d.getElementsByTagName('head')[0].appendChild(script);
}(document));
```

Which should give the correct result:

```
The script was successfully injected!
```

Now that the "<script>" tag has been injected and executed, but we need to figure out how to use it. Mozilla's documentation at *https://github.com/mozilla/readability* provides us with some instructions, based on which we can try executing the following (still in the console window):

```
var documentClone = document.cloneNode(true);
var loc = document.location;
var uri = {
  spec: loc.href,
  host: loc.host,
  prePath: loc.protocol + "//" + loc.host,
  scheme: loc.protocol.substr(0, loc.protocol.indexOf(":")),
  pathBase: loc.protocol + "//" + loc.host +
            loc.pathname.substr(0, loc.pathname.lastIndexOf("/") + 1)
};

var article = new Readability(uri, documentClone).parse();
console.log(article);
```

This should provide you with a result as shown in Figure 9-12, which looks promising indeed: the "article" object contains a "title" and "content" attribute we'll be able to use.

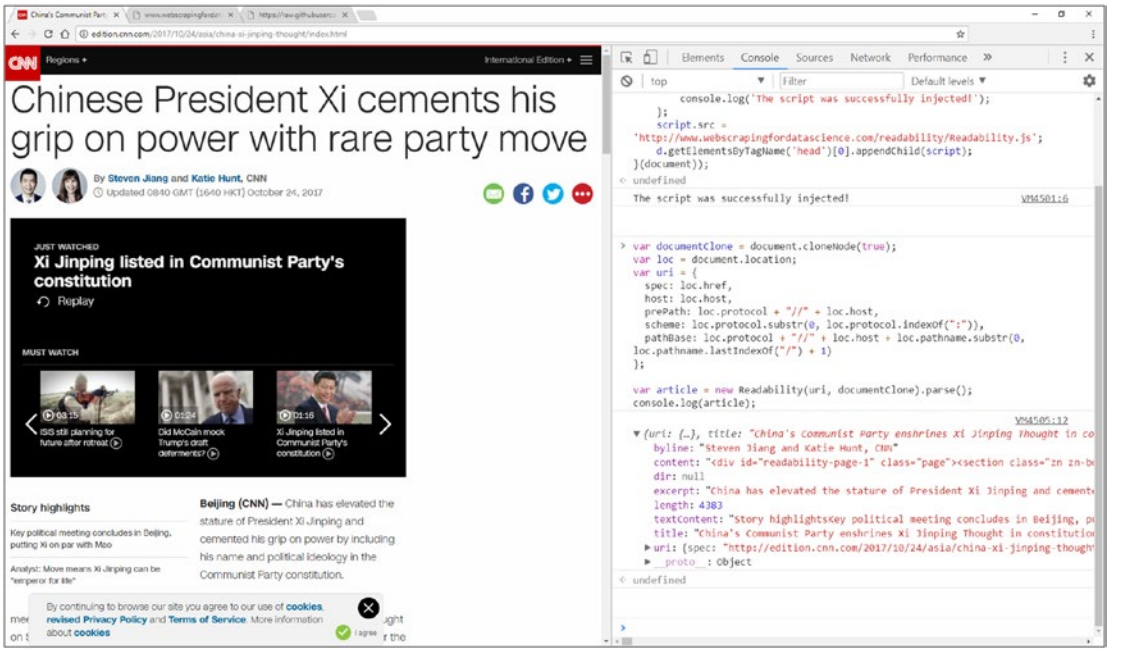

Figure 9-12. *Extracting the article's information*

The question is now how we can return this information to Selenium. Remember that we can execute JavaScript commands from Selenium through the `execute_script` method. One possible approach to get out the information we want is to use JavaScript to replace the whole page's contents with the information we want, and then use Selenium to get out that information:

```
from selenium import webdriver

base_url = 'http://edition.cnn.com/2017/10/24/asia/china-xi-jinping-
thought/index.html'

driver = webdriver.Chrome()
driver.implicitly_wait(10)

driver.get(base_url)
```

```
js_cmd = '''
(function(d, script) {
  script = d.createElement('script');
  script.type = 'text/javascript';
  script.async = true;
  script.onload = function(){
    var documentClone = document.cloneNode(true);
    var loc = document.location;
    var uri = {
      spec: loc.href,
      host: loc.host,
      prePath: loc.protocol + "//" + loc.host,
      scheme: loc.protocol.substr(0, loc.protocol.indexOf(":")),
      pathBase: loc.protocol + "//" + loc.host +
                loc.pathname.substr(0, loc.pathname.lastIndexOf("/") + 1)
    };
    var article = new Readability(uri, documentClone).parse();
    document.body.innerHTML = '<h1 id="title">' + article.title + '</h1>' +
      '<div id="content">' + article.content + '</div>';
  };
  script.src = 'http://www.webscrapingfordatascience.com/readability/
  Readability.js';
d.getElementsByTagName('head')[0].appendChild(script);
}(document));
'''

driver.execute_script(js_cmd)

title = driver.find_element_by_id('title').text.strip()
content = driver.find_element_by_id('content').text.strip()

print('Title was:', title)

driver.quit()
```

The “document.body.innerHTML” line in the JavaScript command will replace the contents of the “<body>” tag with a header and a “ ” tag, from which we can then simply retrieve our desired information.

However, the execute_script method also allows us to pass back JavaScript objects to Python, so the following approach also works:

```
from selenium import webdriver
from selenium.webdriver.common.by import By
from selenium.webdriver.support.ui import WebDriverWait
from selenium.webdriver.support import expected_conditions as EC

base_url = 'http://edition.cnn.com/2017/10/24/asia/china-xi-jinping-
thought/index.html'

driver = webdriver.Chrome()
driver.implicitly_wait(10)

driver.get(base_url)

js_cmd = '''
(function(d, script) {
  script = d.createElement('script');
  script.type = 'text/javascript';
  script.async = true;
  script.onload = function() {
      script.id = 'readability-script';
  }
  script.src = 'http://www.webscrapingfordatascience.com/readability/
  Readability.js';
  d.getElementsByTagName('head')[0].appendChild(script);
}(document));
'''

js_cmd2 = '''
var documentClone = document.cloneNode(true);
var loc = document.location;
var uri = {
  spec: loc.href,
  host: loc.host,
  prePath: loc.protocol + "//" + loc.host,
  scheme: loc.protocol.substr(0, loc.protocol.indexOf(":")),
```

```
  pathBase: loc.protocol + "//" + loc.host +
            loc.pathname.substr(0, loc.pathname.lastIndexOf("/") + 1)
};
var article = new Readability(uri, documentClone).parse();
return JSON.stringify(article);
'''

driver.execute_script(js_cmd)

wait = WebDriverWait(driver, 10)
wait.until(EC.presence_of_element_located((By.ID, "readability-script")))

returned_result = driver.execute_script(js_cmd2)

print(returned_result)

driver.quit()
```

There are several intricacies here that warrant some extra information. First, note that we're using the `execute_script` method twice: once to inject the "<script>" tag, and then again to get out our "article" object. However, since executing the script might take some time, and Selenium's implicit wait does not take this into account when using `execute_script`, we use an explicit wait to check for the presence of an element with an "id" of "readability-script," which is set by the "script.onload" function. Once such an id is found, we know that the script has finished loading and we can execute the second JavaScript command. Here, we do need to use "JSON.stringify" to make sure we return a JSON-formatted string instead of a raw JavaScript object to Python, as Python will not be able to make sense of this return value and convert it to a list of None values (simple types, such as integers and strings, are fine, however).

Let's clean up our script a little and merge it with our basic framework:

```
from selenium import webdriver
from selenium.webdriver.common.by import By
from selenium.webdriver.support.ui import WebDriverWait
from selenium.webdriver.support import expected_conditions as EC

base_url = 'https://news.google.com/news/?ned=us&hl=en'
```

```
inject_readability_cmd = '''
(function(d, script) {
  script = d.createElement('script');
  script.type = 'text/javascript';
  script.async = true;
  script.onload = function() {
      script.id = 'readability-script';
  }
  script.src = 'http://www.webscrapingfordatascience.com/readability/
  Readability.js';
  d.getElementsByTagName('head')[0].appendChild(script);
}(document));
'''

get_article_cmd = '''
var documentClone = document.cloneNode(true);
var loc = document.location;
var uri = {
  spec: loc.href,
  host: loc.host,
  prePath: loc.protocol + "//" + loc.host,
  scheme: loc.protocol.substr(0, loc.protocol.indexOf(":")),
  pathBase: loc.protocol + "//" + loc.host +
            loc.pathname.substr(0, loc.pathname.lastIndexOf("/") + 1)
};
var article = new Readability(uri, documentClone).parse();
return JSON.stringify(article);
'''
driver = webdriver.Chrome()
driver.implicitly_wait(10)

driver.get(base_url)

news_urls = []
for link in driver.find_elements_by_css_selector('main a[role="heading"]'):
    news_url = link.get_attribute('href')
    news_urls.append(news_url)
```

```
for news_url in news_urls:
    print('Now scraping:', news_url)
    driver.get(news_url)

    print('Injecting scripts')
    driver.execute_script(inject_readability_cmd)
    wait = WebDriverWait(driver, 10)
    wait.until(EC.presence_of_element_located((By.ID, "readability-script")))
    returned_result = driver.execute_script(get_article_cmd)

    # Do something with returned_result

driver.quit()
```

Note that we're using two "for" loops: one to extract the links we wish to scrape, which we'll store in a list; and another one to iterate over the list. Using one loop wouldn't work in this case: as we're navigating to other pages inside of the loop, Selenium would complain about "stale elements" when trying to find the next link with `find_elements_by_css_selector`. This is basically saying: "I'm trying to find the next element for you, but the page has changed in the meantime, so I can't be sure anymore what you want to retrieve."

If you try to execute this script, you'll note that it quickly fails anyway. What is happening here? To figure out what is going wrong, try opening another link in your browser, say *https://www.washingtonpost.com/world/chinas-leader-elevated-to-the-level-of-mao-in-communist-pantheon/2017/10/24/ddd911e0-b832-11e7-9b93-b97043e57a22_story.html?utm_term=.720e06a5017d* (a site using HTTPS), and executing the first JavaScript command manually in your browser's console, that is, by copy-pasting and executing:

```
(function(d, script) {
  script = d.createElement('script');
  script.type = 'text/javascript';
  script.async = true;
  script.onload = function() {
      script.id = 'readability-script';
}
```

```
script.src = 'http://www.webscrapingfordatascience.com/readability/
Readability.js';
  d.getElementsByTagName('head')[0].appendChild(script);
}(document));
```

You'll probably get a result like what follows:

```
GET https://www.webscrapingfordatascience.com/readability/Readability.js ↩
net::ERR_CONNECTION_CLOSED
```

On other pages, you might get:

```
Mixed Content: The page at [...] was loaded over HTTPS, but requested an ↩
insecure script 'http://www.webscrapingfordatascience.com/readability/
Readability.js'.
This request has been blocked; the content must be served over HTTPS.
```

It's clear what's going on here: if we load a website through HTTPS and try to inject a script through HTTP, Chrome will block this request as it deems it insecure (which is true). Other sites might apply other approaches to prevent script injection, using for example, a "Content-Security-Policy" header. that would result in an error like this:

```
Refused to load the script
    'http://www.webscrapingfordatascience.com/readability/Readability.js' ↩
    because it violates the following Content Security Policy directive: ↩
    "script-src 'self' 'unsafe-eval' 'unsafe-inline'".
```

There are extensions available for Chrome that will disable such checks, but we're going to take a different approach here, which will work on the majority of pages except those with the most strict Content Security Policies: instead of trying to inject a "<script>" tag, we're going to simply take the contents of our JavaScript file and execute these directly using Selenium. We can do so by loading the contents from a local file, but since we've already hosted the file online, we're going to use requests to fetch the contents instead:

```
from selenium import webdriver
import requests
```

```
base_url = 'https://news.google.com/news/?ned=us&hl=en'
script_url = 'http://www.webscrapingfordatascience.com/readability/
Readability.js'

get_article_cmd = requests.get(script_url).text
get_article_cmd += '''

var documentClone = document.cloneNode(true);
var loc = document.location;
var uri = {
  spec: loc.href,
  host: loc.host,
  prePath: loc.protocol + "//" + loc.host,
  scheme: loc.protocol.substr(0, loc.protocol.indexOf(":")),
  pathBase: loc.protocol + "//" + loc.host +
            loc.pathname.substr(0, loc.pathname.lastIndexOf("/") + 1)
};
var article = new Readability(uri, documentClone).parse();
return JSON.stringify(article);
'''

driver = webdriver.Chrome()
driver.implicitly_wait(10)

driver.get(base_url)

news_urls = []
for link in driver.find_elements_by_css_selector('main a[role="heading"]'):
    news_url = link.get_attribute('href')
    news_urls.append(news_url)

for news_url in news_urls:
    print('Now scraping:', news_url)
    driver.get(news_url)

    print('Injecting script')
    returned_result = driver.execute_script(get_article_cmd)

    # Do something with returned_result
driver.quit()
```

This approach also has the benefit that we can execute our whole JavaScript command in one go and do not need to rely on an explicit wait anymore to check whether the script has finished loading. The only thing remaining now is to convert the retrieved result to a Python dictionary and store our results in a database, once more using the "dataset" library:

```
from selenium import webdriver
import requests
import dataset
from json import loads

db = dataset.connect('sqlite:///news.db')

base_url = 'https://news.google.com/news/?ned=us&hl=en'
script_url = 'http://www.webscrapingfordatascience.com/readability/
Readability.js'

get_article_cmd = requests.get(script_url).text
get_article_cmd += '''

var documentClone = document.cloneNode(true);
var loc = document.location;
var uri = {
  spec: loc.href,
  host: loc.host,
  prePath: loc.protocol + "//" + loc.host,
  scheme: loc.protocol.substr(0, loc.protocol.indexOf(":")),
  pathBase: loc.protocol + "//" + loc.host +
            loc.pathname.substr(0, loc.pathname.lastIndexOf("/") + 1)
};
var article = new Readability(uri, documentClone).parse();
return JSON.stringify(article);
'''

driver = webdriver.Chrome()
driver.implicitly_wait(10)

driver.get(base_url)
```

```
news_urls = []
for link in driver.find_elements_by_css_selector('main a[role="heading"]'):
    news_url = link.get_attribute('href')
    news_urls.append(news_url)

for news_url in news_urls:
    print('Now scraping:', news_url)
    driver.get(news_url)

    print('Injecting script')
    returned_result = driver.execute_script(get_article_cmd)

    # Convert JSON string to Python dictionary
    article = loads(returned_result)
    if not article:
        # Failed to extract article, just continue
        continue
    # Add in the url
    article['url'] = news_url
    # Remove 'uri' as this is a dictionary on its own
    del article['uri']
    # Add to the database
    db['articles'].upsert(article, ['url'])

    print('Title was:', article['title'])

driver.quit()
```

The output looks as follows:

```
Now scraping: https://www.usnews.com/news/world/articles/2017-10-24/
              china-southeast-asia-aim-to-build-trust-with-sea-drills-
              singapore-says Injecting script
Title was: China, Southeast Asia Aim to Build Trust With Sea Drills,    ↩
           Singapore Says | World News
```

```
Now scraping: http://www.philstar.com/headlines/2017/10/24/
              1751999/pentagon-chief-seeks-continued-maritime-cooperation-
              asean Injecting script
Title was: Pentagon chief seeks continued maritime cooperation with ASEAN |
           Headlines News, The Philippine Star,
[...]
```

Remember to take a look at the database ("news.db") using a SQLite client such as "DB Browser for SQLite"; see Figure 9-13.

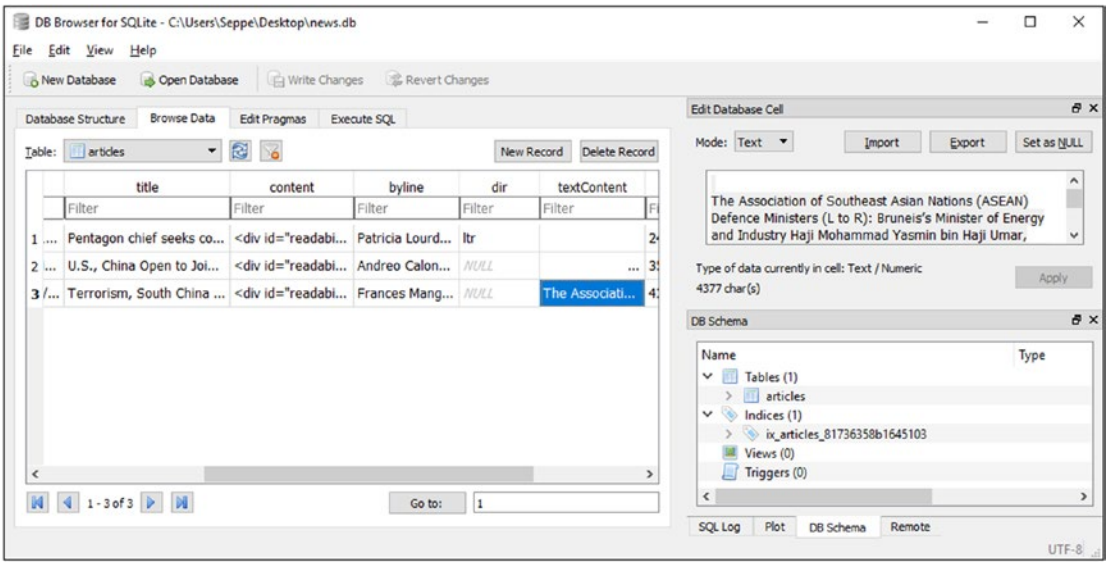

Figure 9-13. *Exploring some scraped articles with DB Browser for SQLite*

We can now analyze our collected articles using Python. We're going to construct a topic model using Latent Dirichlet Allocation (LDA) that will help us to categorize our articles along some topics. To do so, we'll use the "nltk," "stop-words," and "gensim" libraries, which can simply be installed using pip:

```
pip install -U nltk
pip install -U stop-words
pip install -U gensim
```

First, we're going to loop through all our articles in order to tokenize them (convert text into a list of word elements) using a simple regular expression, remove stop words, and apply stemming:

```
import dataset
from nltk.tokenize import RegexpTokenizer
from nltk.stem.porter import PorterStemmer
from stop_words import get_stop_words

db = dataset.connect('sqlite:///news.db')

articles = []

tokenizer = RegexpTokenizer(r'\w+')
stop_words = get_stop_words('en')
p_stemmer = PorterStemmer()

for article in db['articles'].all():
    text = article['title'].lower().strip()
    text += " " + article['textContent'].lower().strip()
    if not text:
        continue
    # Tokenize
    tokens = tokenizer.tokenize(text)
    # Remove stop words and small words
    clean_tokens = [i for i in tokens if not i in stop_words]
    clean_tokens = [i for i in clean_tokens if len(i) > 2]
    # Stem tokens
    stemmed_tokens = [p_stemmer.stem(i) for i in clean_tokens]
    # Add to list
    articles.append((article['title'], stemmed_tokens))

print(articles[0])
```

Our first article now looks as follows (we keep the title for later reporting):

```
('Paul Manafort, former business partner to surrender in Mueller
investigation', ['presid', 'trump', 'former', 'campaign', 'chairman', [...]]
```

To generate an LDA model, we need to calculate how frequently each term occurs within each document. To do that, we can construct a document-term matrix with gensim:

```
from gensim import corpora

dictionary = corpora.Dictionary([a[1] for a in articles])
corpus = [dictionary.doc2bow(a[1]) for a in articles]

print(corpus[0])
```

The `Dictionary` class traverses texts and assigns a unique integer identifier to each unique token while also collecting word counts and relevant statistics. Next, our dictionary is converted to a bag of words corpus that results in a list of vectors equal to the number of documents. Each document vector is a series of "(id, count)" tuples:

```
[(0, 10), (1, 17), (2, 7), (3, 11), [...]]
```

We're now ready to construct an LDA model:

```
from gensim.models.ldamodel import LdaModel

nr_topics = 30
ldamodel = LdaModel(corpus, num_topics=nr_topics,
                    id2word=dictionary, passes=20)

print(ldamodel.print_topics())
```

This will show something like:

```
[(0, '0.027*"s" + 0.018*"trump" + 0.018*"manafort" + 0.011*"investig"    ↵
     + 0.008*"presid" + 0.008*"report" + 0.007*"mueller" + 0.007*"year"  ↵
     + 0.007*"campaign" + 0.006*"said"'),
 (1, '0.014*"s" + 0.014*"said" + 0.013*"percent" + 0.008*"1" +           ↵
     0.007*"0" + 0.006*"year" + 0.006*"month" + 0.005*"increas" +        ↵
     0.005*"3" + 0.005*"spend"'),
 [...]
]
```

This overview shows an entry per topic. Each topic is represented by a list of probable words to appear in that topic, ordered by probability of appearance. Note that adjusting the model's number and amount of "passes" is important to get a good result. Once the results look acceptable (we've increased the number of topics for our scraped set), we can use our model to assign topics to our documents:

```
from random import shuffle

# Show topics by top-3 terms
for t in range(nr_topics):
    print(ldamodel.print_topic(t, topn=3))

# Show some random articles
idx = list(range(len(articles)))
shuffle(idx)
for a in idx[:3]:
    article = articles[a]
    print('=========================')
    print(article[0])
    prediction = ldamodel[corpus[a]][0]
    print(ldamodel.print_topic(prediction[0], topn=3))
    print('Probability:', prediction[1])
```

This will show something like the following:

```
0.014*"new" + 0.013*"power" + 0.013*"storm"
0.030*"rapp" + 0.020*"spacey" + 0.016*"said"
0.024*"catalan" + 0.020*"independ" + 0.019*"govern"
0.025*"manafort" + 0.020*"trump" + 0.015*"investig"
0.007*"quickli" + 0.007*"complex" + 0.007*"deal"
0.018*"earbud" + 0.016*"iconx" + 0.014*"samsung"
0.012*"halloween" + 0.007*"new" + 0.007*"star"
0.021*"octopus" + 0.014*"carver" + 0.013*"vega"
0.000*"rapp" + 0.000*"spacey" + 0.000*"said"
0.025*"said" + 0.017*"appel" + 0.012*"storm"
0.039*"akzo" + 0.018*"axalta" + 0.017*"billion"
```

```
0.024*"rapp" + 0.024*"spacey" + 0.017*"said"
0.000*"boehner" + 0.000*"one" + 0.000*"trump"
0.033*"boehner" + 0.010*"say" + 0.009*"hous"
0.000*"approv" + 0.000*"boehner" + 0.000*"quarter"
0.017*"tax" + 0.013*"republican" + 0.011*"week"
0.012*"trump" + 0.008*"plan" + 0.007*"will"
0.005*"ludwig" + 0.005*"underlin" + 0.005*"sensibl"
0.015*"tax" + 0.011*"trump" + 0.011*"look"
0.043*"minist" + 0.032*"prime" + 0.030*"alleg"
0.058*"harri" + 0.040*"polic" + 0.032*"old"
0.040*"musk" + 0.026*"tunnel" + 0.017*"compani"
0.055*"appl" + 0.038*"video" + 0.027*"peterson"
0.011*"serv" + 0.008*"almost" + 0.007*"insid"
0.041*"percent" + 0.011*"year" + 0.010*"trump"
0.036*"univers" + 0.025*"econom" + 0.012*"special"
0.022*"chees" + 0.021*"patti" + 0.019*"lettuc"
0.000*"boehner" + 0.000*"said" + 0.000*"year"
0.000*"boehner" + 0.000*"new" + 0.000*"say"
0.030*"approv" + 0.025*"quarter" + 0.021*"rate"
==========
Paul Manafort, Who Once Ran Trump Campaign, Indicted on Money Laundering
and Tax Charges
0.025*"manafort" + 0.020*"trump" + 0.015*"investig"
Probability: 0.672658189483
==========
Apple fires employee after daughter's iPhone X video goes viral
0.055*"appl" + 0.038*"video" + 0.027*"peterson"
Probability: 0.990880503145
==========
Theresa May won't say when she knew about sexual harassment allegations
0.043*"minist" + 0.032*"prime" + 0.030*"alleg"
Probability: 0.774530402797
```

Scraping Topics There is still a lot of room to improve on this by, for example, exploring other topic model mapping algorithms, applying better tokenization, adding custom stop words, or expanding the set of articles or adjusting the parameters. Alternatively, you might also consider scraping the tags for each article straight from the Google News page, which also includes these as "topics" on its page.

9.13 Scraping and Analyzing a Wikipedia Graph

In this example, we'll work once again with Wikipedia (we already used Wikipedia in the chapter on web crawling). Our goal here is to scrape titles of Wikipedia pages, while keeping track of links between them, which we'll use to construct a graph and analyze it using Python. We'll again use the "dataset" library as a simple means to store results. The following code contains the full crawling setup:

```
import requests
import dataset
from bs4 import BeautifulSoup
from urllib.parse import urljoin, urldefrag
from joblib import Parallel, delayed

db = dataset.connect('sqlite:///wikipedia.db')
base_url = 'https://en.wikipedia.org/wiki/'

def store_page(url, title):
    print('Visited page:', url)
    print(' title:', title)
    db['pages'].upsert({'url': url, 'title': title}, ['url'])

def store_links(from_url, links):
    db.begin()
    for to_url in links:
        db['links'].upsert({'from_url': from_url, 'to_url': to_url},
                           ['from_url', 'to_url'])
    db.commit()
```

```
def get_random_unvisited_pages(amount=10):
    result = db.query('''SELECT * FROM links
        WHERE to_url NOT IN (SELECT url FROM pages)
        ORDER BY RANDOM() LIMIT {}'''.format(amount))
    return [r['to_url'] for r in result]

def should_visit(base_url, url):
    if url is None:
        return None
    full_url = urljoin(base_url, url)
    full_url = urldefrag(full_url)[0]
    if not full_url.startswith(base_url):
        # This is an external URL
        return None
    ignore = ['Wikipedia:', 'Template:', 'File:', 'Talk:', 'Special:',
              'Template talk:', 'Portal:', 'Help:', 'Category:', 'index.php']
    if any([i in full_url for i in ignore]):
        # This is a page to be ignored
        return None
    return full_url

def get_title_and_links(base_url, url):
    html = requests.get(url).text
    html_soup = BeautifulSoup(html, 'html.parser')
    page_title = html_soup.find(id='firstHeading')
    page_title = page_title.text if page_title else ''
    links = []
    for link in html_soup.find_all("a"):
        link_url = should_visit(base_url, link.get('href'))
        if link_url:
            links.append(link_url)
    return url, page_title, links

if __name__ == '__main__':
    urls_to_visit = [base_url]
```

```
while urls_to_visit:
    scraped_results = Parallel(n_jobs=5, backend="threading")(
        delayed(get_title_and_links)(base_url, url) for url in
        urls_to_visit
    )
    for url, page_title, links in scraped_results:
        store_page(url, page_title)
        store_links(url, links)
    urls_to_visit = get_random_unvisited_pages()
```

There are a lot of things going on here that warrant some extra explanation:

- The database is structured as follows: a table "pages" holds a list of visited URLs with their page titles. The method `store_page` is used to store entries in this table. Another table, "links," simply contains pairs of URLs to represent links between pages. The method `store_link` is used to update these, and both methods use the "dataset" library. For the latter, we perform multiple upsert operations inside a single explicit database transaction to speed things up.
- The method `get_random_unvisited_pages` now returns a list of unvisited URLs, rather than just one, by selecting a random list of linked-to URLs that do not yet appear in the "pages" table (and hence have not been visited yet).
- The `should_visit` method is used to determine whether a link should be considered for crawling. It returns a proper formatted URL if it should be included, or None otherwise.
- The `get_title_and_links` method performs the actual scraping of pages, fetching their title and a list of URLs.
- The script itself loops until there are no more unvisited pages (basically forever, as new pages will continue to be discovered). It fetches out a list of random pages we haven't visited yet, gets their title and links, and stores these in the database.

- Note that we use the "joblib" library here to set up a parallel approach. Simply visiting URLs one by one would be a tad too slow here, so we use joblib to set up a multithreaded approach to visit links at the same time, effectively spawning multiple network requests. It's important not to hammer our own connection or Wikipedia, so we limit the `n_jobs` argument to five. The `back-end` argument is used here to indicate that we want to set up a parallel calculation using multiple threads, instead of multiple processes. Both approaches have their pros and cons in Python. A multi-process approach comes with a bit more overhead to set up, but it can be faster as Python's internal threading system can be a bit tedious due to the "global interpreter lock" (the GIL) (a full discussion about the GIL is out of scope here, but feel free to look up more information online if this is the first time you have heard about it). In our case, the work itself is relatively straightforward: execute a network request and perform some parsing, so a multithreading approach is fine.
- This is also the reason why we don't store the results in the database inside the `get_title_and_links` method itself, but wait until the parallel jobs have finished their execution and have returned their results. SQLite doesn't like to be written to from multiple threads or many processes at once, so we wait until we have collected the results before writing them to the database. An alternative would be to use a client-server database system. Note that we should avoid overloading the database too much with a huge set of results. Not only will the intermediate results have to be stored in memory, but we'll also incur a waiting time when writing the large set of results. Since the `get_random_unvisited_pages` method returns a list of ten URLs maximum, we don't need to worry about this too much in our case.
- Finally, note that the main entry point of the script is now placed under "if __name__ == '__main__':". In other examples, we have not done so for the sake of simplicity, although it is good practice to do so nonetheless. The reason for this is as follows: when a Python script imports another module, all the code contained in that module is executed at once. For instance, if we'd like to reuse the `should_visit` method in another script, we could import our original script using "import myscript"

or "from myscript import should_visit." In both cases, the full code in "myscript.py" will be executed. If this script contains a block of code, like our "while" loop in this example, it will start executing that block of code, which is not what we want when importing our script; we just want to load the function definitions. We hence want to indicate to Python to "only execute this block of code when the script is directly executed," which is what the "if __name__ == '__main__':" check does. If we start our script from the command line, the special "__name__" variable will be set to "__main__". If our script would be imported from another module, "__name__" will be set to that module's name instead. When using joblib as we do here, the contents of our script will be sent to all "workers" (threads or processes), in order for them to perform the correct imports and load the correct function definitions. In our case, for instance, the different workers should know about the `get_title_and_links` method. However, since the workers will also execute the full code contained in the script (just like an import would), we also need to prevent them from running the main block of code as well, which is why we need to provide an "if __name__ == '__main__':" check.

You can let the crawler run for as long as you like, though note that it is extremely unlikely to ever finish, and a smaller graph will also be a bit easier to look at in the next step. Once it has run for a bit, simply interrupt it to stop it. Since we use "upsert," feel free to resume it later on (it will just continue to crawl based on where it left off).

We can now perform some fun graph analysis using the scraped results. In Python, there are two popular libraries available to do so, NetworkX (the "networkx" library in pip) and iGraph ("python-igraph" in pip). We'll use NetworkX here, as well as "matplotlib" to visualize the graph.

Graph Visualization Is Hard As the NetworkX documentation itself notes, proper graph visualization is hard, and the library authors recommend that people visualize their graphs with tools dedicated to that task. For our simple use case, the built-in methods suffice, even although we'll have to wrangle our way though matplotlib to make things a bit more appealing. Take a look at programs such as Cytoscape, Gephi, and Graphviz if you're interested in graph visualization. In the next example, we'll use Gephi to handle the visualization workload.

The following code visualizes the graph. We first construct a new NetworkX graph object and add in the pages as visited nodes. Next, we add the edges, though only between pages that were visited. As an extra step, we also remove nodes that are completely unconnected (even although these should not be present at this stage). We then calculate a centrality measure, called betweenness, as a measure of importance of nodes. This metric is calculated based on the number of shortest paths from all nodes to all other nodes that pass through the node we're calculating the metric for. The more times a node lies on the shortest path between two other nodes, the more important it is according to this metric. We'll color the nodes based on this metric by giving them different shades of blue. We apply a quick and dirty sigmoid function to the betweenness metric to "squash" the values in a range that will result in a more appealing visualization. We also add labels to nodes manually here, in order to have them appear above the actual nodes. This will provide a result as shown in Figure 9-14.

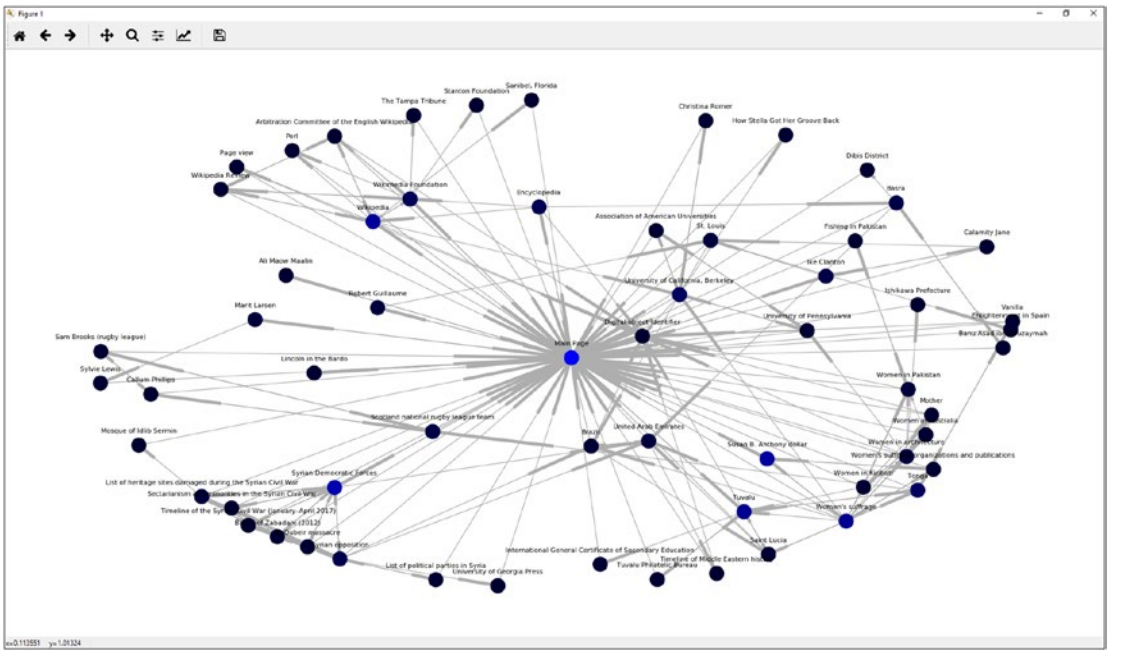

Figure 9-14. *Visualizing our scraped graph*

Ignore the Warnings When running the visualization code, you'll most likely see warnings appear from matplotlib complaining about the fact that NetworkX is using deprecated functions. This is fine and can be safely ignored, though future versions of matplotlib might not play nice with NetworkX anymore. It's unclear whether the authors of NetworkX will continue to focus on visualization in the future. As you'll note, the "arrows" of the edges in the visualization also don't look very pretty. This is a long-standing issue with NetworkX. Again: NetworkX is fine for analysis and graph wrangling, though less so for visualization. Take a look at other libraries if visualization is your core concern.

```
import networkx
import matplotlib.pyplot as plt
import dataset

db = dataset.connect('sqlite:///wikipedia.db')
G = networkx.DiGraph()

print('Building graph...')
for page in db['pages'].all():
    G.add_node(page['url'], title=page['title'])

for link in db['links'].all():
    # Only addedge if the endpoints have both been visited
    if G.has_node(link['from_url']) and G.has_node(link['to_url']):
        G.add_edge(link['from_url'], link['to_url'])

# Unclutter by removing unconnected nodes
G.remove_nodes_from(networkx.isolates(G))

# Calculate node betweenness centrality as a measure of importance
print('Calculating betweenness...')
betweenness = networkx.betweenness_centrality(G, endpoints=False)

print('Drawing graph...')

# Sigmoid function to make the colors (a little) more appealing
squash = lambda x : 1 / (1 + 0.5**(20*(x-0.1)))
```

```
colors = [(0, 0, squash(betweenness[n])) for n in G.nodes()]
labels = dict((n, d['title']) for n, d in G.nodes(data=True))
positions = networkx.spring_layout(G)

networkx.draw(G, positions, node_color=colors, edge_color='#AEAEAE')

# Draw the labels manually to make them appear above the nodes
for k, v in positions.items():

    plt.text(v[0], v[1]+0.025, s=labels[k],
             horizontalalignment='center', size=8)

plt.show()
```

9.14 Scraping and Visualizing a Board Members Graph

In this example, our goal is to construct a social graph of S&P 500 companies and their interconnectedness through their board members. We'll start from the S&P 500 page at Reuters available at *https://www.reuters.com/finance/markets/index/.SPX* to obtain a list of stock symbols:

```
from bs4 import BeautifulSoup
import requests
import re

session = requests.Session()

sp500 = 'https://www.reuters.com/finance/markets/index/.SPX'

page = 1
regex = re.compile(r'\/finance\/stocks\/overview\/.*')
symbols = []

while True:
    print('Scraping page:', page)
    params = params={'sortBy': '', 'sortDir' :'', 'pn': page}
    html = session.get(sp500, params=params).text
    soup = BeautifulSoup(html, "html.parser")
```

```
    pagenav = soup.find(class_='pageNavigation')
    if not pagenav:
        break
    companies = pagenav.find_next('table', class_='dataTable')
    for link in companies.find_all('a', href=regex):
        symbols.append(link.get('href').split('/')[-1])
    page += 1

print(symbols)
```

Once we have obtained a list of symbols, we can scrape the board member pages for each of them (e.g., *https://www.reuters.com/finance/stocks/company-officers/MMM.N*), fetch the table of board members, and store it as a pandas data frame, which we'll save using pandas' `to_pickle` method. Don't forget to install pandas first if you haven't already:

```
pip install -U pandas
```

Add this to the bottom of your script:

```
import pandas as pd

officers = 'https://www.reuters.com/finance/stocks/company-officers/
{symbol}'

dfs = []

for symbol in symbols:
    print('Scraping symbol:', symbol)
    html = session.get(officers.format(symbol=symbol)).text
    soup = BeautifulSoup(html, "html.parser")
    officer_table = soup.find('table', {"class" : "dataTable"})
    df = pd.read_html(str(officer_table), header=0)[0]
    df.insert(0, 'symbol', symbol)
    dfs.append(df)

# Store the results
df = pd.concat(dfs)
df.to_pickle('sp500.pkl')
```

This sort of information can lead to a lot of interesting use cases, especially — again — in the realm of graph and social network analytics. We're going to use NetworkX once more, but simply to parse through our collected information and export a graph in a format that can be read with Gephi, a popular graph visualization tool, which can be downloaded from *https://gephi.org/users/download/*:

```
import pandas as pd
import networkx as nx
from networkx.readwrite.gexf import write_gexf

df = pd.read_pickle('sp500.pkl')

G = nx.Graph()

for row in df.itertuples():
    G.add_node(row.symbol, type='company')
    G.add_node(row.Name,type='officer')
    G.add_edge(row.symbol, row.Name)

write_gexf(G, 'graph.gexf')
```

Open the graph file in Gephi, and apply the "ForceAtlas 2" layout technique for a few iterations. We can also show labels as well, resulting in a figure like the one shown in Figure 9-15.

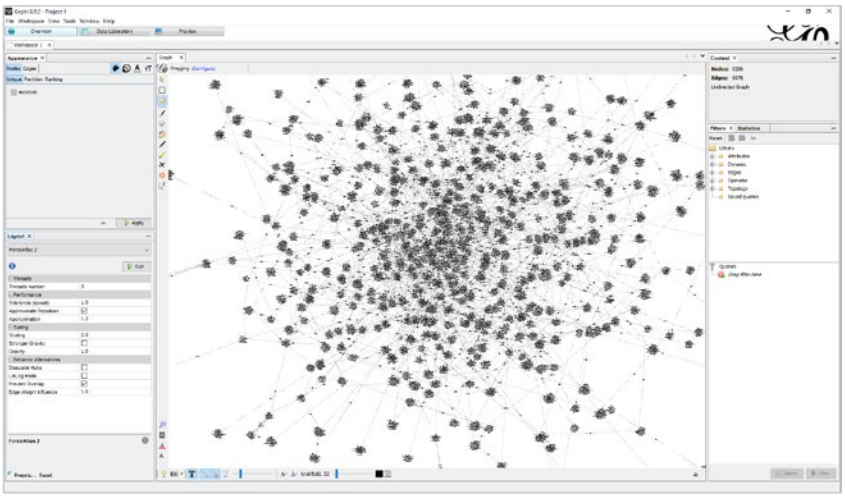

Figure 9-15. *Visualizing our scraped graph using Gephi*

Take some time to explore Gephi's visualization and filtering options if you like. All attributes that you have set in NetworkX ("type," in our case) will be available in Gephi as well. Figure 9-16 shows the filtered graph for Google, Amazon, and Apple with their board members, which are acting as connectors to other firms.

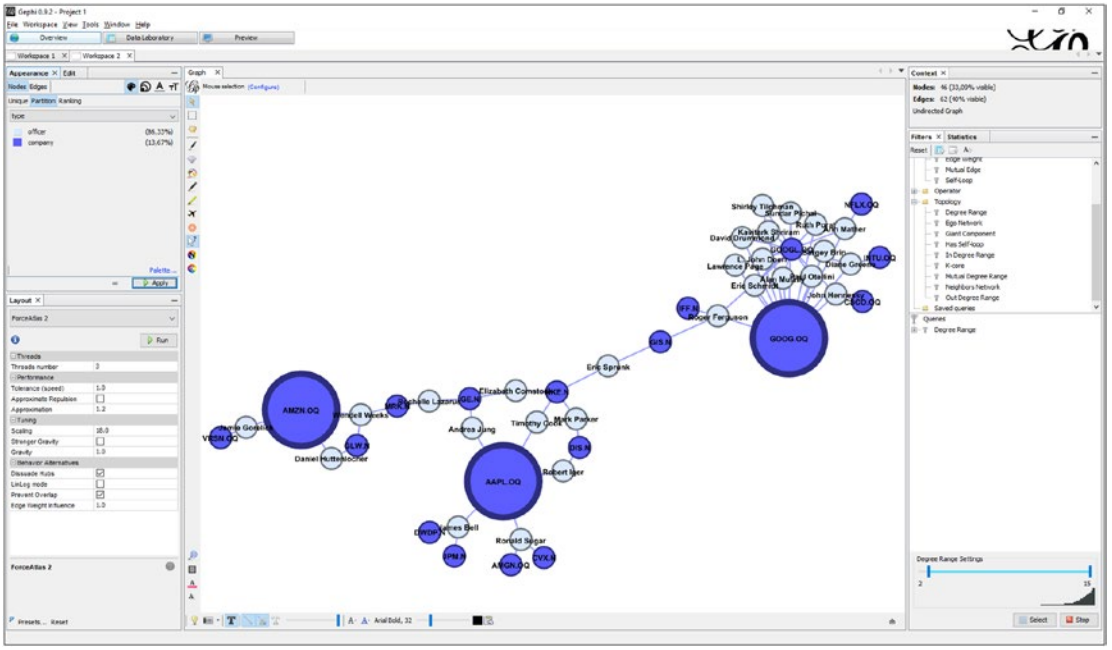

Figure 9-16. *Showing connected board members for Google, Amazon, and Apple*

9.15 Breaking CAPTCHA's Using Deep Learning

This final example is definitely the most challenging one, as well as the one that is mostly related to "data science," rather than web scraping. In fact, we'll not use any web scraping tools here. Instead, we're going to walk through a relatively contained example to illustrate how you could incorporate a predictive model in your web scraping pipeline in order to bypass a CAPTCHA check.

We're going to need to install some tools first. We'll use "OpenCV," an extremely thorough library for computer vision, as well as "numpy" for some basic data wrangling. Finally, we'll use the "captcha" library to generate example images. All of these can be installed as follows:

```
pip install -U opencv-python
pip install -U numpy
pip install -U captcha
```

Next, create a directory somewhere in your system to contain the Python scripts we will create. The first script ("constants.py") will contain some constants we're going to use:

```
CAPTCHA_FOLDER = 'generated_images'
LETTERS_FOLDER = 'letters'

CHARACTERS = list('QWERTPASDFGHKLZXBNM')
NR_CAPTCHAS = 1000
NR_CHARACTERS = 4

MODEL_FILE = 'model.hdf5'
LABELS_FILE = 'labels.dat'

MODEL_SHAPE = (100, 100)
```

Another script ("generate.py") will generate a bunch of CAPTCHA images and save them to the "generated_images" directory:

```
from random import choice
from captcha.image import ImageCaptcha
import os.path
from os import makedirs
from constants import *

makedirs(CAPTCHA_FOLDER)

image = ImageCaptcha()

for i in range(NR_CAPTCHAS):
    captcha = ''.join([choice(CHARACTERS) for c in range(NR_CHARACTERS)])
    filename = os.path.join(CAPTCHA_FOLDER, '{}_{}.png'.format(captcha, i))
    image.write(captcha, filename)
    print('Generated:', captcha)
```

After running this script, you should end up with a collection of CAPTCHA images (with their answers in the file names) as shown in Figure 9-17.

Figure 9-17. *A collection of generated CAPTCHA images*

Isn't This Cheating? Of course, we're lucky here that we are generating the CAPTCHA's ourselves and hence have the opportunity to keep the answers as well. In the real world, however, CAPTCHA's do not expose their answer (it would kind of refute the point of the CAPTCHA), so that we would need to figure out another way to create our training set. One way is to look for the library a particular site is using to generate its CAPTCHA's and use it to collect a set of training images of your own, replicating the originals as closely as possible. Another approach is to manually label the images yourself, which is as dreadful as it sounds, though you might not need to label thousands of images to get the desired result. Since people make mistakes when filling in CAPTCHA's, too, we have more than one chance to get the answer right and hence do not need to target a 100 percent accuracy level. Even if our predictive model is only able to get one out of ten images right, that is still sufficient to break through a CAPTCHA after some retries.

Next, we're going to write another script that will cut up our images into separate pieces, one per character. We could try to construct a model that predicts the complete answer all at once, though in many cases it is much easier to perform the predictions character by character. To cut up our image, we'll need to invoke OpenCV to perform some heavy lifting for us. A complete discussion regarding OpenCV and computer vision would require a book in itself, so we'll stick to some basics here. The main concepts we'll use here are thresholding, opening, and contour detection. To see how this works, let's create a small test script first to show these concepts in action:

```
import cv2
import numpy as np

# Change this to one of your generated images:
image_file = 'generated_images/ABQM_116.png'

image = cv2.imread(image_file)
cv2.imshow('Original image', image)

# Convert to grayscale, followed by thresholding to black and white
gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
_, thresh = cv2.threshold(gray, 0, 255, cv2.THRESH_BINARY_INV | cv2.THRESH_
OTSU)
cv2.imshow('Black and white', thresh)

# Apply opening: "erosion" followed by "dilation"
denoised = thresh.copy()
kernel = np.ones((4, 3), np.uint8)
denoised = cv2.erode(denoised, kernel, iterations=1)
kernel = np.ones((6, 3), np.uint8)
denoised = cv2.dilate(denoised, kernel, iterations=1)
cv2.imshow('Denoised', denoised)

# Now find contours and overlay them over our original image
_, cnts, _ = cv2.findContours(denoised.copy(), cv2.RETR_TREE, cv2.CHAIN_
APPROX_NONE)
cv2.drawContours(image, cnts, contourIdx=-1, color=(255, 0, 0),
thickness=-1)
cv2.imshow('Contours', image)

cv2.waitKey(0)
```

If you run this script, you should obtain a list of preview windows similar as shown in Figure 9-18. In the first two steps, we open our image with OpenCV and convert it to a simple pure black and white representation. Next, we apply an "opening" morphological transformation, which boils down to an erosion followed by dilation. The basic idea of erosion is just like soil erosion: this transformation "erodes away" boundaries of the foreground object (which is assumed to be in white) by sliding a "kernel" over the image (a "window," so to speak) so that only those white pixels are retained if all pixels in the surrounding kernel are white as well. Otherwise, it gets turned to black. Dilation does the opposite: it widens the image by setting pixels to white if at least one pixel in the surrounding kernel was white. Applying these steps is a very common tactic to remove noise from images. The kernel sizes used in the script above are simply the result of some trial and error, and you might want to adjust these with other types of CAPTCHA images. Note that we allow for some noise in the image to remain present. We don't need to obtain a perfect image as we trust that our predictive model will be able to "look over these."

Figure 9-18. *Processing an image with OpenCV. From left to right: original image, image after conversion to black and white, image after applying an opening to remove noise, and the extracted contours overlaid in blue over the original image.*

Next, we use OpenCV's `findContours` method to extract "blobs" of connected white pixels. OpenCV comes with various methods to perform this extraction and different ways to represent the result (e.g., simplifying the contours or not, constructing a hierarchy or not, and so on). Finally, we use the `drawContours` method to draw the discovered blobs. The `contourIdx` argument here indicates that we want to draw all top-level contours, and the `thickness` value of -1 instructs OpenCV to fill up the contours.

We now still need a way to use the contours to create separate images: one per character. The way how we'll do so is by using masking. Note that OpenCV also allows to fetch out the "bounding rectangle" for each contour, which would make "cutting" the image much easier, though this might get us into trouble in case parts of the characters are near to each other. Instead, we'll use the approach illustrated by the following code fragment:

```
import cv2
import numpy as np

image_file = 'generated_images/ABQM_116.png'

# Perform thresholding, erosion and contour finding as shown before
image = cv2.imread(image_file)
gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
_, thresh = cv2.threshold(gray, 0, 255, cv2.THRESH_BINARY_INV | cv2.THRESH_
OTSU)
denoised = thresh.copy()
kernel = np.ones((4, 3), np.uint8)
denoised = cv2.erode(denoised, kernel, iterations=1)
kernel = np.ones((6, 3), np.uint8)
denoised = cv2.dilate(denoised, kernel, iterations=1)
_, cnts, _ = cv2.findContours(denoised.copy(), cv2.RETR_TREE, cv2.CHAIN_
APPROX_NONE)

# Create a fresh 'mask' image
mask = np.ones((image.shape[0], image.shape[1]), dtype="uint8") * 0
# We'll use the first contour as an example
contour = cnts[0]
# Draw this contour over the mask
cv2.drawContours(mask, [contour], -1, (255, 255, 255), -1)

cv2.imshow('Denoised image', denoised)

cv2.imshow('Mask after drawing contour', mask)

result = cv2.bitwise_and(denoised, mask)

cv2.imshow('Result after and operation', result)

retain = result > 0
result = result[np.ix_(retain.any(1), retain.any(0))]

cv2.imshow('Final result', result)

cv2.waitKey(0)
```

If you run this script, you'll obtain a result as shown in Figure 9-19. First, we create new black image with the same size as the starting, denoised image. We take one contour and draw it in white on top of this "mask." Next, the denoised image and mask are combined in a bitwise "and" operation, which will retain white pixels if the corresponding pixels in both input images were white, and sets it to black otherwise. Next, we apply some clever numpy slicing to crop the image.

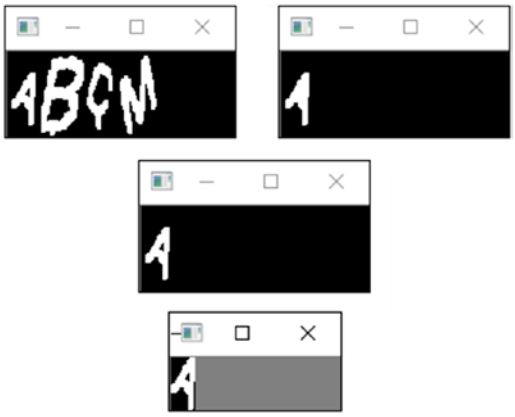

Figure 9-19. *Extracting part of an image using a contour mask in OpenCV. On the top left, the starting image is shown. On the right, a new image is created with the contour drawn in white and filled. These two images are combined in a bitwise "and" operation to obtain the image in the second row. The bottom image shows the final result after applying cropping.*

This is sufficient to get started, though there still is one problem we need to solve: overlap. In case characters overlap, they would be discovered as one large contour. To work around this issue, we'll apply the following operations. First, starting from a list of contours, check whether there is a significant degree of overlap between two distinct contours, in which case we only retain the largest one. Next, we order the contours based on their size, take the first *n* contours, and order these on the horizontal axis, from left to right (with *n* being the number of characters in a CAPTCHA). This still might lead to fewer contours than we need, so that we iterate over each contour, and check whether its width is higher than an expected value. A good heuristic for the expected value is to take the estimated width based on the distance from the leftmost white pixel to the rightmost white pixel divided by the number of characters we expect to see. In case a contour is wider than we expect, we cut it up into *m* equal parts, with *m* being equal to the width of the contour divided by the expected width. This is a heuristic that still might lead to

some characters not being perfectly cut off (some characters are larger than others), but this is something we'll just accept. In case we don't end up with the desired number of characters at the end of all this, we'll simply skip over the given image.

We'll put all of this in a separate list of functions (in a file "functions.py"):

```
import cv2
import numpy as np
from math import ceil, floor
from constants import *

def overlaps(contour1, contour2, threshold=0.8):
    # Check whether two contours' bounding boxes overlap
    area1 = contour1['w'] * contour1['h']
    area2 = contour2['w'] * contour2['h']
    left = max(contour1['x'], contour2['x'])
    right = min(contour1['x'] + contour1['w'], contour2['x'] +
    contour2['w'])
    top = max(contour1['y'], contour2['y'])
    bottom = min(contour1['y'] + contour1['h'], contour2['y'] +
    contour2['h'])
    if left <= right and bottom >= top:
        intArea = (right - left) * (bottom - top)
        intRatio = intArea / min(area1, area2)
        if intRatio >= threshold:
            # Return True if the second contour is larger
            return area2 > area1
    # Don't overlap or doesn't exceed threshold
    return None

def remove_overlaps(cnts):
    contours = []
    for c in cnts:
        x, y, w, h = cv2.boundingRect(c)
        new_contour = {'x': x, 'y': y, 'w': w, 'h': h, 'c': c}
        for other_contour in contours:
            overlap = overlaps(other_contour, new_contour)
```

```
            if overlap is not None:
                if overlap:
                    # Keep this one...
                    contours.remove(other_contour)
                    contours.append(new_contour)
                # ... otherwise do nothing: keep the original one
                break
        else:
            # We didn't break, so no overlap found, add the contour
            contours.append(new_contour)
    return contours

def process_image(image):
    # Perform basic pre-processing
    gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
    _, thresh = cv2.threshold(gray, 0, 255, cv2.THRESH_BINARY_INV |
    cv2.THRESH_OTSU)
    denoised = thresh.copy()
    kernel = np.ones((4, 3), np.uint8)
    denoised = cv2.erode(denoised, kernel, iterations=1)
    kernel = np.ones((6, 3), np.uint8)
    denoised = cv2.dilate(denoised, kernel, iterations=1)
    return denoised

def get_contours(image):
    # Retrieve contours
    _, cnts, _ = cv2.findContours(image.copy(), cv2.RETR_TREE, cv2.CHAIN_
    APPROX_NONE)
    # Remove overlapping contours
    contours = remove_overlaps(cnts)
    # Sort by size, keep only the first NR_CHARACTERS
    contours = sorted(contours, key=lambda x: x['w'] * x['h'],
                      reverse=True)[:NR_CHARACTERS]
    # Sort from left to right
    contours = sorted(contours, key=lambda x: x['x'], reverse=False)
    return contours
```

```
def extract_contour(image, contour, desired_width, threshold=1.7):
    mask = np.ones((image.shape[0], image.shape[1]), dtype="uint8") * 0
    cv2.drawContours(mask, [contour], -1, (255, 255, 255), -1)
    result = cv2.bitwise_and(image, mask)
    mask = result > 0
    result = result[np.ix_(mask.any(1), mask.any(0))]

    if result.shape[1] > desired_width * threshold:
        # This contour is wider than expected, split it
        amount = ceil(result.shape[1] / desired_width)
        each_width = floor(result.shape[1] / amount)
        # Note: indexing based on im[y1:y2, x1:x2]
        results = [result[0:(result.shape[0] - 1),
                          (i * each_width):((i + 1) * each_width - 1)] \
                   for i in range(amount)]
        return results
    return [result]

def get_letters(image, contours):
    desired_size = (contours[-1]['x'] + contours[-1]['w'] - contours[0]['x']) \
                    / NR_CHARACTERS
    masks = [m for l in [extract_contour(image, contour['c'], desired_size) \
             for contour in contours] for m in l]
    return masks
```

With this, we're finally ready to write our cutting script ("cut.py")

```
from os import makedirs
import os.path
from glob import glob
from functions import *
from constants import *

image_files = glob(os.path.join(CAPTCHA_FOLDER, '*.png'))
```

```
for image_file in image_files:
    print('Now doing file:', image_file)
    answer = os.path.basename(image_file).split('_')[0]
    image = cv2.imread(image_file)
    processed = process_image(image)
    contours = get_contours(processed)
    if not len(contours):
        print('[!] Could not extract contours')
        continue
    letters = get_letters(processed, contours)
    if len(letters) != NR_CHARACTERS:
        print('[!] Could not extract desired amount of characters')
        continue
    if any([l.shape[0] < 10 or l.shape[1] < 10 for l in letters]):
        print('[!] Some of the extracted characters are too small')
        continue
    for i, mask in enumerate(letters):
        letter = answer[i]
        outfile = '{}_{}.png'.format(answer, i)
        outpath = os.path.join(LETTERS_FOLDER, letter)
        if not os.path.exists(outpath):
            makedirs(outpath)
        print('[i] Saving', letter, 'as', outfile)
        cv2.imwrite(os.path.join(outpath, outfile), mask)
```

If you run this script, the "letters" directory should now contain a directory for each letter; see, for example, Figure 9-20. We're now ready to construct our deep learning model . We'll use a simple convolutional neural network architecture, using the "Keras" library.

```
pip install -U keras
```

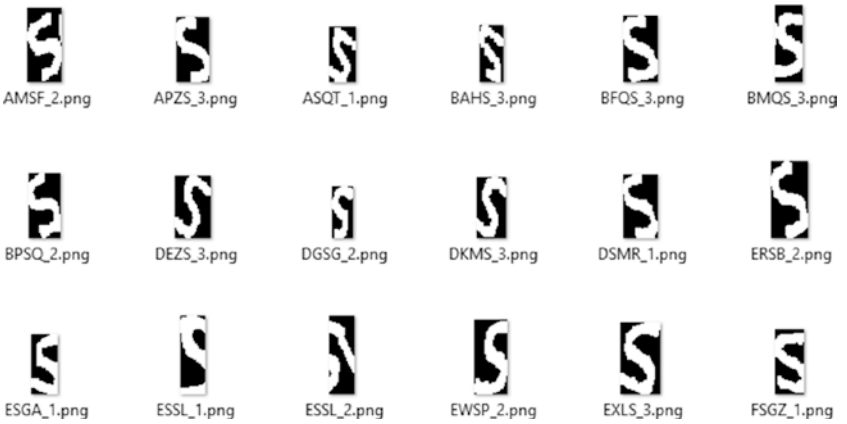

Figure 9-20. *A collection of extracted "S" images*

For Keras to work, we also need to install a back end (the "engine" Keras will use, so to speak). You can use the rather limited "theano" library, Google's "Tensorflow," or Microsoft's "CNTK." We assume you're using Windows, so CNTK is the easiest option to go with. (If not, install the "theano" library using pip instead.) To install CNTK, navigate to *https://docs.microsoft.com/en-us/cognitive-toolkit/setup-windows-python?tabs=cntkpy231* and look for the URL corresponding with your Python version. If you have a compatible GPU in your computer, you can use the "GPU" option. If this doesn't work or you run into trouble, stick to the "CPU" option. Installation is then performed as such (using the GPU Python 3.6 version URL):

```
pip install -U https://cntk.ai/PythonWheel/GPU/cntk-2.3.1-cp36-cp36m-
win_amd64.whl
```

Next, we need to create a Keras configuration file. Run a Python REPL and import Keras as follows:

```
>>> import keras
Using TensorFlow backend.
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "\site-packages\keras\__init__.py", line 3, in <module>
    from . import utils
  File "\site-packages\keras\utils\__init__.py", line 6, in <module>
    from . import conv_utils
```

```
  File "\site-packages\keras\utils\conv_utils.py", line 3, in <module>
    from .. import backend as K
  File "\site-packages\keras\backend\__init__.py", line 83, in <module>
    from .tensorflow_backend import *
  File "\site-packages\keras\backend\tensorflow_backend.py", line 1, in
  <module>
    import tensorflow as tf
ModuleNotFoundError: No module named 'tensorflow'
```

Keras will complain about the fact that it can't find Tensorflow, its default back end. That's fine; simply exit the REPL. Next, navigate to "%USERPROFILE%\.keras" in Windows' file explorer. There should be a "keras.json" file there. Open this file using Notepad or another text editor, and replace the contents so that it reads as follows:

```
{
    "floatx": "float32",
    "epsilon": 1e-07,
    "backend": "cntk",
    "image_data_format": "channels_last"
}
```

Using Another Back End In case you're using Tensorflow, just leave the "backend" value set to "tensorflow." If you're using theano, set the value to "theano." Note that in the latter case, you might also need to look for a ".theanorc.txt" file on your system and change its contents as well to get things to work on your system, especially the "device" entry that you should set to "cpu" in case theano has trouble finding your GPU.

Once you've made this change, try test-importing Keras once again into a fresh REPL session. You should now get the following:

```
>>> import keras
Using CNTK backend
Selected GPU[1] GeForce GTX 980M as the process wide default device.
```

Keras is now set up and is recognizing our GPU. If CNTK would complain, remember to try the CPU version instead, though keep in mind that training the model will take much longer in this case (and so will theano and Tensorflow in case you can only use CPU-based computing).

We can now create another Python script to train our model ("train.py"):

```
import cv2
import pickle
from os import listdir
import os.path
import numpy as np
from glob import glob
from sklearn.preprocessing import LabelBinarizer
from sklearn.model_selection import train_test_split
from keras.models import Sequential
from keras.layers.convolutional import Conv2D, MaxPooling2D
from keras.layers.core import Flatten, Dense
from constants import *

data = []
labels = []
nr_labels = len(listdir(LETTERS_FOLDER))

# Convert each image to a data matrix
for label in listdir(LETTERS_FOLDER):
    for image_file in glob(os.path.join(LETTERS_FOLDER, label, '*.png')):
        image = cv2.imread(image_file)
        image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
        # Resize the image so all images have the same input shape
        image = cv2.resize(image, MODEL_SHAPE)
        # Expand dimensions to make Keras happy
        image = np.expand_dims(image, axis=2)
        data.append(image)
        labels.append(label)

# Normalize the data so every value lies between zero and one
data = np.array(data, dtype="float") / 255.0
labels = np.array(labels)
```

```
# Create a training-test split
(X_train, X_test, Y_train, Y_test) = train_test_split(data, labels,
                                    test_size=0.25, random_state=0)

# Binarize the labels
lb = LabelBinarizer().fit(Y_train)
Y_train = lb.transform(Y_train)
Y_test = lb.transform(Y_test)

# Save the binarization for later
with open(LABELS_FILE, "wb") as f:
    pickle.dump(lb, f)
# Construct the model architecture
model = Sequential()
model.add(Conv2D(20, (5, 5), padding="same",
          input_shape=(MODEL_SHAPE[0], MODEL_SHAPE[1], 1),
activation="relu"))
model.add(MaxPooling2D(pool_size=(2, 2), strides=(2, 2)))
model.add(Conv2D(50, (5, 5), padding="same", activation="relu"))
model.add(MaxPooling2D(pool_size=(2, 2), strides=(2, 2)))
model.add(Flatten())
model.add(Dense(500, activation="relu"))
model.add(Dense(nr_labels, activation="softmax"))
model.compile(loss="categorical_crossentropy", optimizer="adam",
metrics=["accuracy"])

# Train and save the model
model.fit(X_train, Y_train, validation_data=(X_test, Y_test),
          batch_size=32, epochs=10, verbose=1)
model.save(MODEL_FILE)
```

We're doing a number of things here. First, we loop through all images we have created, resize them, and store their pixel matrix as well as their answer. Next, we normalize the data so that each value lies between zero and one, which makes things a bit easier on the neural network. Next, since Keras can't work with "Q", "W",… labels directly, we need to binarize these: every label is converted to an output vertex with each

index corresponding to one possible character, with its value set to one or zero, so that "Q" would become "[1, 0, 0, 0,...]," "W" would become "[0, 1, 0, 0,...]," and so on. We save this conversion as we'll also need it to perform the conversion back to characters again during application of the model. Next, we construct the neural architecture (which is relatively simple, in fact), and start training the model. If you run this script, you'll get an output as follows:

```
Using CNTK backend
Selected GPU[0] GeForce GTX 980M as the process wide default device.

Train on 1665 samples, validate on 555 samples
Epoch 1/10

C:\Users\Seppe\Anaconda3\lib\site-packages\cntk\core.py:361: UserWarning: ↩
your data is of type "float64", but your input variable (uid "Input4")   ↩
    expects "<class'numpy.float32'>". Please convert your data           ↩
    beforehand to speed up training.
  (sample.dtype, var.uid, str(var.dtype)))

  32/1665 [..............................] - ETA: 36s - loss: 3.0294 -
                                           acc: 0.0312
  64/1665 [>.............................] - ETA: 22s - loss: 5.1515 -
                                           acc: 0.0312

[...]

1600/1665 [===========================>..] - ETA: 0s - loss: 7.6135e-04 -
                                           acc: 1.0000
1632/1665 [============================>.] - ETA: 0s - loss: 8.3265e-04 -
                                           acc: 1.0000
1664/1665 [============================>.] - ETA: 0s - loss: 8.2343e-04 -
                                           acc: 1.0000
1665/1665 [==============================] - 3s 2ms/step - loss: 8.2306e-
                                           04 - acc:
     1.0000 - val_loss: 0.3644 - val_acc: 0.9207
```

We're getting a 92 percent accuracy on the validation set, not bad at all! The only thing that remains now is to show how we'd use this network to predict a CAPTCHA ("apply.py"):

```
from keras.models import load_model
import pickle
import os.path
from glob import glob
from random import choice
from functions import *
from constants import *

with open(LABELS_FILE, "rb") as f:
    lb = pickle.load(f)

model = load_model(MODEL_FILE)

# We simply pick a random training image here to illustrate how predictions
work. In a real setup, you'd obviously plug this into your web scraping
# pipeline and pass a "live" captcha image
image_files = list(glob(os.path.join(CAPTCHA_FOLDER, '*.png')))
image_file = choice(image_files)

print('Testing:', image_file)

image = cv2.imread(image_file)
image = process_image(image)
contours = get_contours(image)
letters = get_letters(image, contours)

for letter in letters:
    letter = cv2.resize(letter, MODEL_SHAPE)
    letter = np.expand_dims(letter, axis=2)
    letter = np.expand_dims(letter, axis=0)
    prediction = model.predict(letter)
    predicted = lb.inverse_transform(prediction)[0]
    print(predicted)
```

If you run this script, you should see something like the following:

```
Using CNTK backend
Selected GPU[0] GeForce GTX 980M as the process wide default device.

Testing: generated_images\NHXS_322.png
N
H
X
S
```

As you can see, the network correctly predicts the sequence of characters in the CAPTCHA. This concludes our brief tour of CAPTCHA cracking. As we've discussed before, keep in mind that several alternative approaches exist, such as training an OCR toolkit or using a service with "human crackers" at low cost. Also keep in mind that you might have to fine-tune both OpenCV and the Keras model in case you plan to apply this idea on other CAPTCHA's, and that the CAPTCHA generator we've used here is still relatively "easy." Most important, however, remains the fact that CAPTCHA's signpost a warning, basically explicitly stating that web scrapers are not welcome. Keep this intricacy in mind as well before you set off cracking CAPTCHA's left and right.

Even a Traditional Model Might Work As we've seen, it's not that trivial to set up a deep learning pipeline. In case you're wondering whether a traditional predictive modeling technique such as random forests or support vector machines might also work (both of these are available in scikit-learn, for instance, and are much quicker to set up and train), the answer is that yes, in some cases, these might work, albeit at a heavy accuracy cost. Such traditional techniques have a hard time understanding the two-dimensional structure of images, which is exactly what a convolutional neural network aims to solve. This being said, we've set up pipelines using a random forest and about 100 manually labeled CAPTCHA images that obtained a low accuracy of about 10 percent, though enough to get the answer right after a handful of tries.
