Unit 5: Text Analysis

Contents

Lab Questions

1, 2, 3, 4, 5, 6, 7, 8, 9, 10

Getting Started

A company is often interested in determining the public's opinion of the company's products and services. Traditionally, this data might have been obtained through the use of customer satisfaction surveys or broader market research efforts. With the growth in use of online shopping there are large collections of user reviews attesting to the quality of a company's offerings. Similarly, the popular social media platforms allow users to easily share opinions in a public forum. Collecting and analyzing this data can provide useful information that could be used to guide a company's decision-making process.

In previous lessons, we worked primarily with *structured data* - data with a well-defined organizational structure that adhered to certain properties which we could easily infer through exploration. For example, the when we worked with county auditor data, each dataset was tabular with rows that represented an individual parcel and columns represented different properties of those parcels. Each column had a specific data type and it was unlikely that a column value for a specific row would violate that type property. Structured data could be stored in a relational database (making use of the structured imposed by such a data store) and easily searched.

Text in the form of user reviews or social media posts are often classified as *unstructured data* - data lacking a pre-defined organizational model where the characteristics of one record could differ significantly from another. For example, in a collection of user reviews, one record might be "This is a great product. I've owned it for five years and it always works perfectly!". Characteristics of this data could be that it is 84 characters long, written in English, and uses only ASCII (https://en.wikipedia.org/wiki/ASCII) characters and we might write code that relies on those assumptions. However, among the many reviews might be this review, "Buen equipo, robusto, rápido, ... Perfecto para el uso que se le va a dar. Tiene de todo. Diseño moderno." This review is clearly not written in English and uses non-ASCII characters so any analysis that assumed these features might fail. We'll discuss collecting and analyzing unstructured data, focusing on text data.

As computing power and algorithms improve, it has become increasingly easy to extract information from unstructured data. With this ability, it is important to be aware that there are potential ethical and legal issues. As users freely post their thought on social media, is it ethically right to mine this data in attempt to increase profit or for other gain? While an exploration of legal issues is outside the scope of the material covered here, it is important that one is aware that such issues could exist.

To work with text data, we'll make use of the following libraries.

- beautifulsoup (https://www.crummy.com/software/BeautifulSoup/) - a web scraping library
- requests-oauthlib (https://github.com/requests/requests-oauthlib) - OAuth (https://en.wikipedia.org/wiki/OAuth) support for the requests) library
- textblob (https://textblob.readthedocs.io/en/dev/) - a library for processing text data built on [NLTK])https://www.nltk.org/ (https://www.nltk.org/)) and pattern (https://www.clips.uantwerpen.be/pages/pattern-en)
- tinydb (https://tinydb.readthedocs.io/en/latest/) - a document data store implemented in Python

We'll use `pip` to install these libraries. We've worked with some of these before but we can ensure they are installed using `pip`.

In [145]:
```
import sys
!{sys.executable} -m pip install beautifulsoup4 requests_oauthlib textblob t
```

```
Requirement already satisfied: beautifulsoup4 in /usr/local/lib/python3.
6/site-packages
Requirement already satisfied: requests_oauthlib in /usr/local/lib/python
3.6/site-packages
Collecting textblob
  Downloading textblob-0.15.1-py2.py3-none-any.whl (631kB)
    100% |████████████████████████████████| 634kB 1.8MB/s ta 0:00:01
Requirement already satisfied: tinydb in /usr/local/lib/python3.6/site-pa
ckages
Requirement already satisfied: oauthlib>=0.6.2 in /usr/local/lib/python3.
6/site-packages (from requests_oauthlib)
Requirement already satisfied: requests>=2.0.0 in /usr/local/lib/python3.
6/site-packages (from requests_oauthlib)
Requirement already satisfied: nltk>=3.1 in /usr/local/lib/python3.6/site
-packages (from textblob)
Requirement already satisfied: chardet<3.1.0,>=3.0.2 in /usr/local/lib/py
thon3.6/site-packages (from requests>=2.0.0->requests_oauthlib)
Requirement already satisfied: idna<2.7,>=2.5 in /usr/local/lib/python3.
6/site-packages (from requests>=2.0.0->requests_oauthlib)
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/pytho
n3.6/site-packages (from requests>=2.0.0->requests_oauthlib)
Requirement already satisfied: urllib3<1.23,>=1.21.1 in /usr/local/lib/py
thon3.6/site-packages (from requests>=2.0.0->requests_oauthlib)
Requirement already satisfied: six in /usr/local/lib/python3.6/site-packa
ges (from nltk>=3.1->textblob)
Installing collected packages: textblob
Successfully installed textblob-0.15.1
```

Collection

Web Scraping

A lot of human-generated text data in the form of reviews or social media posts are accessible on a web site. One way of accessing this data is through a process knowing as *web scraping* (https://en.wikipedia.org/wiki/Web_scraping). Web pages are creating using HTML (https://en.wikipedia.org/wiki/HTML), a markup language that provides a way of indicating structure for the accompanying content. We can make use of this structure as a way of identifying the location of text on a website that is of interest to us. Modern websites make heavy use of JavaScript and CSS for interactive functionality and to tailor their appearance; while these are nice when we, as humans, are browsing a site, they don't usually impart structural changes to the underlying data so we'll only focus on HTML content for scraping.

Let's look at a relatively simple example. Consider this web page that was generated using Amazon product review data available on Kaggle (https://www.kaggle.com/datafiniti/consumer-reviews-of-amazon-products/data). This page is stored locally with our other data; to load data from a web server, we could use a library like requests (http://docs.python-requests.org/en/master/).

In [10]:
```python
from IPython.display import HTML
HTML(filename="./data/05-reviews.html")
```

Excellent 5.0

This item is exactly what I was looking for. It is a perfect size for home or travel. It is very loud for its size.

Great product 5.0

Makes for a great alarm and reminder device. I own 2 dots and the tap. Use them all.

Great Blutooth Speaker 4.0

Great Bluetooth speaker with the added benefit of Alexa

Amazing 5.0

Love being able to use Alexa unplugged. Amazing device!

We have hundreds of reviews for a specific product where each review includes a title, a rating and the review text. To process this data, we first need to extract it. Let's look at a bit of the underlying HTML.

Lab 1 In the cell below, load the content of `./data/05-reviews.html` and display the first 100 lines.

In [27]:
```
with open('./data/05-reviews.html') as infile:
    reviews = infile.readlines()

for line in reviews[:100]:
    print(line)
```

```
<!DOCTYPE html>

<html>

<head>

    <title>Amazon Tap Reviews</title>

    <style>

        .review_container {

            padding: 1em;

        }



        .review_container:nth-child(odd) {
```

Examining the HTML, we see that within the *body* (https://www.w3schools.com/tags/tag_body.asp) element, we have individual *div* (https://www.w3schools.com/tags/tag_div.asp) elements for each review. For each review, there are three *span* (https://www.w3schools.com/tags/tag_span.asp) elements that correspond to the review title, the review rating, and the review text, in order. We could write code to read the content of the file and extract this review data by examining each line as a string and looking for *div* and *span* elements and extracting the corresponding text but the code would become complicated. We previously used *pandas* to extract data from HTML but that relied on the data being stored in an HTML *table* (https://www.w3schools.com/tags/tag_table.asp), which is not the case with this data.

One option is to make use of Python's standard library XML processing modules (https://docs.python.org/3.6/library/xml.html); XML (https://en.wikipedia.org/wiki/XML) is another markup language often used to store data. While HTML is different and not a subset of XML (https://stackoverflow.com/q/5558502/1298998); we can often use Python's XML tools to process HTML. We'll use the *ElementTree* (https://docs.python.org/3.6/library/xml.etree.elementtree.html#module-xml.etree.ElementTree) module to do this. We begin by importing the module and parsing the file.

In [35]:
```
import xml.etree.ElementTree as ET
tree = ET.parse('./data/05-reviews.html')
```

We can think of the document structure as being tree-like: at the root, we have the *html* (https://www.w3schools.com/tags/tag_html.asp) element, which contains (or branches into) a *head* (https://www.w3schools.com/tags/tag_head.asp) and a *body* element, which each branch further. Frist, let's get the root element. using the *getroot()* method.

```
In [37]: root = tree.getroot()
         display(root)
```

```
<Element 'html' at 0x112c41ea8>
```

We can see that the root is, in fact, an *html* element. To see its child elements, we can use the *getchildren()* method that returns a list of child elements.

```
In [38]: root.getchildren()
```

```
Out[38]: [<Element 'head' at 0x112c41d68>, <Element 'body' at 0x112c41ef8>]
```

If we know the specific tag for a child, we can use the *find()* or *findall()* methods. The *find()* method will return the first matching child and *findall()* will return a list of matching children.

```
In [43]: body = root.find("body")
         display(body)
```

```
<Element 'body' at 0x112c41ef8>
```

Looking at the HTML text content from the file above, we know that the *body* element contains a *div* for each review. We can use *findall()* to access these children.

Lab 2 In the cell below, use the *findall()* method with `body` to find all *div* elements within the body. Store the result in the variable named `div` . Use the *len()* function with `div` to display the number of found elements.

```
In [44]: divs = body.findall("div")
         len(divs)
```

```
Out[44]: 540
```

There are 540 *div* elements in the *body* element. This implies there are 540 reviews on the page that we can extract. Looking again at the HTML file, we can see that within each *div* there are three *span* elements corresponding to the review title, rating, and text. Let's look at the first *div* in `divs` and see its children.

```
In [45]: div = divs[0]
         div.findall("span")
```

```
Out[45]: [<Element 'span' at 0x112c41db8>,
          <Element 'span' at 0x112c41d18>,
          <Element 'span' at 0x112c41cc8>]
```

To access any text associated with an element, we can use the *text* property. We can iterate through the `div` 's children that are *spans* and print the text for each of them. We'll use *enumerate()* (https://docs.python.org/3/library/functions.html#enumerate) to keep track of the index of the span whose text we're displaying.

In [47]:
```
for i, span in enumerate(div.findall('span')):
    print(i, span.text)
```

```
0 Tap Alexa on the go!
1 5.0
2 It was just a few weeks ago that I was bemoaning the fact that I did no
t scoop up an Echo back when they were being
            released last year. Not only could it be had at nearly half o
ff the current price, but it came with a voice remote
            and we could have had a year longer using this amazing devic
e...Well, here I am, a year late to the game and
            I can now proudly say that I own both an Echo and this new li
ttle Tap! Can I just say that I am OBSESSED with
            all things Echo and Alexa. These are devices that you may not
realize how much you'll use it until you've given
            them a try.I have been spending the last few years getting ou
r home automated. I currently use a combination
            of Smartthings, IFTTT and Logitech Harmony to control the hom
e. Alexa fits right into the middle of all that
            and gives us the ability to control our home via voice. For m
e, this is the killer feature. I can integrate all
            three parts of our home control right into Echo/Tap and make
our house even smarter with very minimal setup.Read
            more
```

In HTML and XML, tags can have *attributes* (https://www.w3schools.com/xml/xml_attributes.asp). Here, each *div* and *span* element has a `class` attribute. The XML parser stores a tag's attributes as a dictionary that can be accessed using the *attrib* property.

Lab 3 In the cell below, use the *attrib* property with `div` to display the corresponding *div*'s attributes.

In [50]:
```
div.attrib
```

Out[50]:
```
{'class': 'review_container'}
```

Though it looks like the review *span* elements appear in the same order, title-rating-text, we can use the `class` attribute to be sure as to which *span* we are working with while iterating.

```
In [53]: for span in div.findall('span'):
             span_class = span.attrib.get('class')
             if span_class == 'title':
                 print('TITLE:  ', span.text)
             elif span_class == 'rating':
                 print('RAITING:  ', span.text)
             elif span_class == 'review':
                 print('TEXT:  ', span.text)
             else:
                 print('UNKNOWN CLASS')
```

```
TITLE:  Tap Alexa on the go!
RAITING:  5.0
TEXT:  It was just a few weeks ago that I was bemoaning the fact that I d
id not scoop up an Echo back when they were being
            released last year. Not only could it be had at nearly half o
ff the current price, but it came with a voice remote
            and we could have had a year longer using this amazing devic
e...Well, here I am, a year late to the game and
            I can now proudly say that I own both an Echo and this new li
ttle Tap! Can I just say that I am OBSESSED with
            all things Echo and Alexa. These are devices that you may not
realize how much you'll use it until you've given
            them a try.I have been spending the last few years getting ou
r home automated. I currently use a combination
            of Smartthings, IFTTT and Logitech Harmony to control the hom
e. Alexa fits right into the middle of all that
            and gives us the ability to control our home via voice. For m
e, this is the killer feature. I can integrate all
            three parts of our home control right into Echo/Tap and make
our house even smarter with very minimal setup.Read
            more
```

We can now iterate through the HTML structure to extract the title, rating, and text for each review. We'll look at storage options for unstructured data later but for now, we can create a pandas DataFrame. As we iterate through the HTML, we'll extract the data, create a dictionary for each review's data, and append the dictionary to the DataFrame. We can convert the rating to a floating point value as we iterate as well. For clarity, we'll start by parsing the HTML file again.

In [63]:
```python
import pandas as pd

reviews = pd.DataFrame(columns=['title', 'rating', 'text'])

tree = ET.parse('./data/05-reviews.html')
root = tree.getroot()
body = root.find('body')
for review_div in body.findall('div'):

    # skip any div that doesn't have a class attribute with "review_continer
    if review_div.attrib.get('class') != 'review_container':
        continue

    review_data = {}

    for span in review_div.findall('span'):
        span_class = span.attrib.get('class')

        if span_class == 'title':
            review_data['title'] = span.text

        elif span_class == 'rating':
            review_data['rating'] = float(span.text)

        elif span_class == 'review':
            review_data['text'] = span.text

    reviews = reviews.append([review_data], ignore_index=True)

reviews.head()
```

Out[63]:

	title	rating	text
0	Tap Alexa on the go!	5.0	It was just a few weeks ago that I was bemoani...
1	Great for what it does	5.0	Look at this product as a portable speaker fir...
2	Awesome, smart little portable speaker	5.0	This Amazon tap is not only a great Bluetooth ...
3	Amazing technology!	5.0	Bought this on Deal of the Day which surprised...
4	A++++++++	5.0	Amazing Sound! All around great product! Nothi...

At this point, we can calculate the descriptive statistics for the `rating` column

In [64]:
```python
reviews.rating.describe()
```

Out[64]:
```
count    540.000000
mean       4.531481
std        0.787736
min        1.000000
25%        4.000000
50%        5.000000
75%        5.000000
max        5.000000
Name: rating, dtype: float64
```

An alternative to using the standard library is to use the [Beautiful Soup (https://www.crummy.com/software/BeautifulSoup/bs4/doc/)](https://www.crummy.com/software/BeautifulSoup/bs4/doc/) library. Having installed it earlier, we can import it.

```
In [65]: from bs4 import BeautifulSoup
```

To start, we use the *BeautifulSoup()* function specifying the filename for the HTML content and, optionally, the HTML parser we'd like to use.

```
In [79]: with open('./data/05-reviews.html') as infile:
             soup = BeautifulSoup(infile, 'html.parser')
```

While Beautiful Soup allows use to navigate the document tree as we did previously, it provide enhanced functionality that makes it much easier to find the data we want. For our data, we know that each of the reviews is contained in a *div* with class `review_container` . We can quickly get the collection of elements matching these criteria using Beautiful Soup's *find_all()* method.

```
In [80]: divs = soup.find_all('div', {'class': 'review_container'})
         len(divs)

Out[80]: 540
```

We see that we have found all the review *divs*. We can also use the *find()* method to return the first match. Just as with the XML parser, we can access an element's text using the *text* property.

```
In [83]: div = divs[0]
         rating_span = div.find('span', {'class': 'rating'})
         rating_span.text

Out[83]: '5.0'
```

We can rewrite our code that iterate through the file to extract review data using Beautiful Soup.

Lab 4 Using the *find()* method and the *text* property, complete the code below to extract each review's title, rating, and text from corresponding *span* within each review's div using the appropriate class attribute. Replace `[CODE]` with the correct code.

```
In [85]: reviews = pd.DataFrame(columns=['title', 'rating', 'text'])

         with open('./data/05-reviews.html') as infile:
             soup = BeautifulSoup(infile, 'html.parser')

         for review_div in soup.find_all('div', {'class': 'review_container'}):
             review_data = {
                 'title': [CODE],
                 'rating': [CODE],
                 'text': [CODE]
             }
             reviews = reviews.append([review_data], ignore_index=True)

         reviews.head()
```

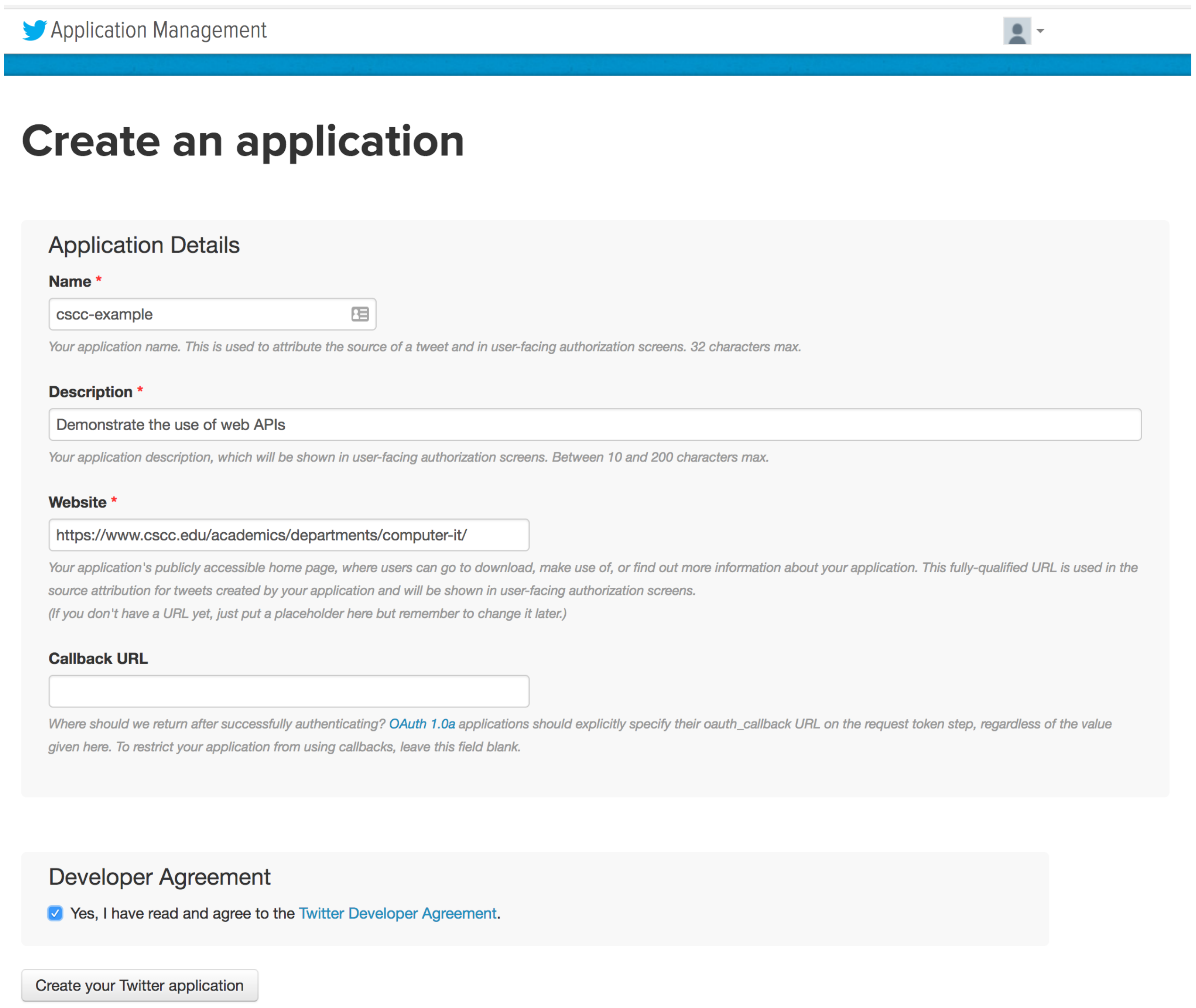
Out[85]:

	title	rating	text
0	Tap Alexa on the go!	5.0	It was just a few weeks ago that I was bemoani...
1	Great for what it does	5.0	Look at this product as a portable speaker fir...
2	Awesome, smart little portable speaker	5.0	This Amazon tap is not only a great Bluetooth ...
3	Amazing technology!	5.0	Bought this on Deal of the Day which surprised...
4	A++++++++	5.0	Amazing Sound! All around great product! Nothi...

Working with an API

As mentioned previously, website owners typically discourage the use of scraping for collection of data; in fact, many prohibit the practice in their terms of service. The data is, however, often made available via an application programming interface (https://en.wikipedia.org/wiki/Application_programming_interface) or API. An API provides us with the means of systematically accessing data from a website or other Internet data store. An overview of what an API is as well as a directory of available APIs is avilable at [Programmable Web]. For our work, we'll use the Twitter API (https://developer.twitter.com/en/docs), specifically the [search] functionality.

Twitter requires users to authenticate their API request. To do this, we can follow the instructions (https://developer.twitter.com/en/docs/basics/getting-started#get-started-app) provided by Twitter. we must first register an application with Twitter. We begin by going to the Twitter Apps (https://apps.twitter.com) page and signing in. If you don't have a Twitter account, you will have to create one and provide a mobile phone number in order to create an app. Once logged in, we can create a new app. When creating a new app, we have to provide some details including the application name, description, and a website; we also have to accept the developer agreement (https://dev.twitter.com/overview/terms/agreement-and-policy). See the following image for an example of the details provided for a new application.


Application Management

Create an application

Application Details

Name *

cscc-example

Your application name. This is used to attribute the source of a tweet and in user-facing authorization screens. 32 characters max.

Description *

Demonstrate the use of web APIs

Your application description, which will be shown in user-facing authorization screens. Between 10 and 200 characters max.

Website *

https://www.cscc.edu/academics/departments/computer-it/

Your application's publicly accessible home page, where users can go to download, make use of, or find out more information about your application. This fully-qualified URL is used in the source attribution for tweets created by your application and will be shown in user-facing authorization screens.
(If you don't have a URL yet, just put a placeholder here but remember to change it later.)

Callback URL

Where should we return after successfully authenticating? OAuth 1.0a applications should explicitly specify their oauth_callback URL on the request token step, regardless of the value given here. To restrict your application from using callbacks, leave this field blank.

Developer Agreement

✓ Yes, I have read and agree to the Twitter Developer Agreement.

Twitter App Details

Once the application is created, the application details page will be displayed. Near the top of the page is a collection of tabs including *Keys and Access Tokens* and *Permission*. We'll use the *Keys and Access Tokens* page in a moment but for now, let's change the access premissions for this application. Click *Permission*, choose *Read only* and update the settings.

Typically, to access data on a website we have to log in with a username or email address and a password to verify our identity and that we are allowed to access certain content. In the same way, we have to prove our identity when using many APIs. Twitter makes use of a standard known as OAuth (https://en.wikipedia.org/wiki/OAuth) to support access to its API. We'll need the following pieces of data from the Twitter App page in order to access the API.

- Consumer Key
- Consumer Secret
- Access Token
- Access Token Secret

The consumer key and consumer secret are on the Twitter App's *Keys and Access Tokens* page - navigate to the page. We'll need to generate an access token and can do so from the same page. After generating the token, we can store the necessary data in variables within the notebook.

Replace the empty strings below with strings containing the appropriate data. Be careful when sharing this data.

Lab 5 Register a Twitter App and replace the empty strings below with values from the Twiteer App page.

```
In [278]: # update these values in order to use the Twitter API
CONSUMER_KEY = ""
CONSUMER_SECRET = ""
ACCESS_TOKEN = ""
ACCESS_TOKEN_SECRET = ""
```

With this data, we can now begin to access the API. We'll create a session with our authentication details and with which we can make additional API requests. To do this, we use the *requests_oathlib* module which relies on *requests*.

```
In [90]: from requests_oauthlib import OAuth1Session

twitter = OAuth1Session(CONSUMER_KEY, CONSUMER_SECRET,
                        ACCESS_TOKEN, ACCESS_TOKEN_SECRET)
```

We can use the `twitter` session to make requests. Let's look at an example of the search API (https://developer.twitter.com/en/docs/tweets/search/api-reference/get-search-tweets.html). To search for popular tweets containing the word "columbus" and with full text in the results, we can make a request to the following URL:

```
https://api.twitter.com/1.1/search/tweets.json?q=columbus&result_typ
e=popular&tweet_mode=extended
```

To make the request in code, we'll use the session's *get()* (http://docs.python-requests.org/en/master/api/#requests.get) method.

```
In [98]: url = "https://api.twitter.com/1.1/search/tweets.json?q=columbus&result_type
response = twitter.get(url)
```

The Twitter API returns JSON content. We can deserialize the content using the response's *json()* (http://docs.python-requests.org/en/master/api/#requests.Response.json) method.

In [99]:
```
results = response.json()
results
```

```
       'indices': [232, 248],
       'name': 'Pens Inside Scoop',
       'screen_name': 'PensInsideScoop'}]},
    'extended_entities': {'media': [{'display_url': 'pic.twitter.com/CYksZ
58WYy',
       'expanded_url': 'https://twitter.com/penguins/status/98245182026097
8689/photo/1',
       'id': 982451674181664768,
       'id_str': '982451674181664768',
       'indices': [274, 297],
       'media_url': 'http://pbs.twimg.com/tweet_video_thumb/DaJel6iWAAAPpM
o.jpg',
       'media_url_https': 'https://pbs.twimg.com/tweet_video_thumb/DaJel6i
WAAAPpMo.jpg',
       'sizes': {'large': {'h': 238, 'resize': 'fit', 'w': 424},
        'medium': {'h': 238, 'resize': 'fit', 'w': 424},
        'small': {'h': 238, 'resize': 'fit', 'w': 424},
        'thumb': {'h': 150, 'resize': 'crop', 'w': 150}},
       'type': 'animated_gif',
       'url': 'https://t.co/CYksZ58WYy',
```

Looking at the results, we can see that the actual tweets themselves are coupled with additional metadata. For now, we'll work with only the tweets themselves. Looking at the results, we can see that the outer data structure is a dictionary with a key named `statuses` and the associated value is a list where each element corresponds to a tweet. Each tweet element is a dictionary where the full tweet is the value paired with the key `full_text` . We can get the first tweet using the following.

Lab 6 In the cell below, display the text of the first text using the appropriate keys and indexes for the nested data structures.

In [101]:
```
results['statuses'][0]['full_text']
```

Out[101]:
```
'Columbus is sitting out Panarin, Bobrovsky, Jones and Werenski tonight.
New Jersey is scratching Hall, Palmieri and Zajac. I did not expect this.
It sure looks like these teams are actively trying to duck the Penguins.'
```

We can also extract all the tweets.

```
In [102]: tweets = []
          for tweet in results['statuses']:
              tweets.append(tweet['full_text'])

          tweets
```

```
Out[102]: ['Columbus is sitting out Panarin, Bobrovsky, Jones and Werenski tonight.
          New Jersey is scratching Hall, Palmieri and Zajac. I did not expect this.
          It sure looks like these teams are actively trying to duck the Penguin
          s.',
           'My “thank god” comment was a sarcastic joke about being a Michigan man
          in Ohio. And I love Columbus &amp; always will. Never spoke bad about the
          club esp the fans, coaches and my brothers I played with. I just could ne
          ver support an owner moving the team. One love Cbus #SaveTheCrew https://
          t.co/Z7LU3wFt1S', (https://t.co/Z7LU3wFt1S',)
           'Our hearts are heavy as we see the tragic news of the bus crash involvi
          ng the Humboldt Broncos. The Columbus Blue Jackets join the hockey world
           in sending our thoughts &amp; prayers to the players, coaches, staff, fa
          milies &amp; community touched by this devastating accident.',
           'Columbus, Ohio we ended this tour with destruction and now we are off t
          o Canada for our headline tour. The tour that never ends and we couldn’t
           be more thankful. Also, we are taking over @altpress Instagram story in
           the morning. Be ready for that! x #palayeroyale https://t.co/PUZg3Wh6i
          D', (https://t.co/PUZg3Wh6iD',)
           "The Penguins couldn't have ended the regular season on a better note. A
          fter a resilient, come-from-behind victory on Thursday vs. Columbus, the
           #Pens built on that with an even better effort at home against Ottawa.\n
          \nSnap Shots from @PensInsideScoop: https://t.co/x3Fc15dv1z (https://t.c
          o/x3Fc15dv1z) https://t.co/CYksZ58WYy", (https://t.co/CYksZ58WYy",)
           'Roster move: \n\n- Chisenhall to 10-day DL with right calf strain \n- N
          aquin recalled from Columbus https://t.co/PG1xI5LJ6K', (https://t.co/PG1x
          I5LJ6K',)
           'Naquin left Cleveland on Saturday morning for Columbus, and had to driv
          e right back up I-71 that night when Chisenhall got hurt. Quipped Francon
          a: "The hope is that when he went back to Triple-A, he made some adjustme
          nts."',
           'Columbus is sitting all its best players tonight against the Preds beca
          use its not concerned about earning third place in the Metro.\n\nTranslat
          ion: Torts doesn’t want the Penguins in the first round. Though, he says
           differently. Whatever you say, dude 😂 https://t.co/genfFor9tw', (http
          s://t.co/genfFor9tw',)
           'Today, @MayorGinther sent a letter to Anthony Precourt, owner of @Colum
          busCrewSC in response to a letter he sent on March 16, 2018.\n \nYou can
           view the letter here https://t.co/BOTJgDH0F6\n\n#CrewSC (https://t.co/BO
          TJgDH0F6\n\n#CrewSC) #Columbus',
           'The power is in the people’s hands. Voter mobilization and training tod
          ay at #ADayofAction in Columbus Ohio #PowerToThePolls kickoff @OhioWomens
          March https://t.co/n6lXncDWZr', (https://t.co/n6lXncDWZr',)
           'The Bruins advance to their 34th NCAA Championship after winning the Co
          lumbus Regional. \n\nMeet recap: https://t.co/TU1QPKzaDY (https://t.co/TU
          1QPKzaDY) https://t.co/YMDEaAyGGa', (https://t.co/YMDEaAyGGa',)
           '#ADayofAction in Columbus is starting soon! #PowerToThePolls https://t.
          co/L04Uh9PgjG', (https://t.co/L04Uh9PgjG',)
           'Columbus Loses. So it’s Now OFFICIALLY #Flyers vs #Penguins First Roun
          d.  #BringItOn https://t.co/7jDu8iLGFY', (https://t.co/7jDu8iLGFY',)
           'Today in Columbus Circle. @CentralParkNYC #springinNYC https://t.co/0aw
          iX5UhK5', (https://t.co/0awiX5UhK5',)
```

```
 'THREE AND ZERO ONCE AGAIN! \n\n#eUCoD sweeps @TheRiseNation in the firs
t round of the #CWLProLeague!! \n\nThe guys are COOKING in Columbus.\n\n#
StandUnited #EUNWIN https://t.co/7b80SBd9kS'] (https://t.co/7b80SBd9kS'])
```

Storage

When working with unstructured data, we often have a need to aggregate and store it. Historically, relational database management systems have been used to store structured data and SQL has been used to interface with these systems. As the need to store unstructured data increased and traditional databases became less practical as storage solutions, NoSQL (https://en.wikipedia.org/wiki/NoSQL) databases began to be developed. There are a variety of types of NoSQL databases, each with its own benefits and drawbacks. One of these types is a document store (https://en.wikipedia.org/wiki/Document-oriented_database) where a *document* is associated with a key and stored in some sort of collection. Here, a *document* is usualy some semi-structured data such as JSON or XML data. Many document stores make use of the semi-structured nature of the documents to provided additional functionality such as querying for data within a document. There are many document store implementations (and many more NoSQL database implementations in general). As an example of working with a document store, we'll make use of TinyDB (https://tinydb.readthedocs.io/en/latest/index.html), a document store implemented entirely in Python. As part of this example, we'll create two stores - one for the review data and the other for the tweets we collected via the Twitter API.

Let's start by importing TinyDb and creating a data store for the twitter data.

In [109]:
```
import tinydb
db = tinydb.TinyDB('./data/documents.json')
```

TinyDB relies on an external file to persist data. This file will itself be a JSON file. Any data we add to the TinyDB document store will be converted to JSON and stored in the file. Like tables in a relational databases, many NoSQL have a means of partitioning data. TinyDB has a notion of tables (https://tinydb.readthedocs.io/en/latest/usage.html?highlight=table#tables) to do this but we must keep in mind that unlike a relational database, we cannot create relationships between tables. Let's create two tables, one for the Twitter data and one for the review data.

In [110]:
```
twitter_table = db.table('twitter')
```

Lab 7 In the cell below, create a table named *review* and store the result in a variable named `review_table`.

In [ ]:
```
review_table = db.table('review')
```

Let's iterate through our review data and store each review in the document store. To do this, we'll use the table's [*insert()* (https://tinydb.readthedocs.io/en/latest/api.html?highlight=insert#tinydb.database.Table.insert)](https://tinydb.readthedocs.io/en/latest/api.html?highlight=insert#tinydb.database.Table.insert) method. To simplify the use of the database, we'll iterate through the DataFrame storing review data, convert each row into a dictionary, and store the dictionary as the document.

```
In [121]: for index, row in reviews.iterrows():
              review_table.insert(row.to_dict())
```

We can get a count of all the documents in the table using the *len()* function.

```
In [122]: len(review_table)
```

```
Out[122]: 540
```

While we can access all the data in a table using the [*all()* (https://tinydb.readthedocs.io/en/latest/api.html?highlight=insert#tinydb.database.Table.insert)](https://tinydb.readthedocs.io/en/latest/api.html?highlight=insert#tinydb.database.Table.insert) method, it's often more convenient to perform queries to filter the data. To create queries we use the table's [*search()* (https://tinydb.readthedocs.io/en/latest/api.html?highlight=insert#tinydb.database.Table.search)](https://tinydb.readthedocs.io/en/latest/api.html?highlight=insert#tinydb.database.Table.search) method along with TinyDB's [*Query* (https://tinydb.readthedocs.io/en/latest/api.html?highlight=insert#tinydb.queries.Query)](https://tinydb.readthedocs.io/en/latest/api.html?highlight=insert#tinydb.queries.Query) class. First we create an instance of the *Query* class.

```
In [123]: Review = tinydb.Query()
```

The *Query* class allows us to access key/value pairs within the document using the dot operator. For example, we can refer to each review's "rating" key using `Review.rating` . Here we find all reviews there the rating is 5.0.

```
In [124]: five_stars = review_table.search(Review.rating == 5.0)
          len(five_stars)
```

```
Out[124]: 356
```

The results are iterable and support bracket notation.

Lab 8 In the cell below, use bracket notation to display the first result in `five_stars` .

```
In [171]: five_stars[0]
```

```
Out[171]: {'rating': 5.0,
            'text': "It was just a few weeks ago that I was bemoaning the fact that
           I did not scoop up an Echo back when they were being\n            release
           d last year. Not only could it be had at nearly half off the current pric
           e, but it came with a voice remote\n            and we could have had a y
           ear longer using this amazing device...Well, here I am, a year late to th
           e game and\n            I can now proudly say that I own both an Echo and
           this new little Tap! Can I just say that I am OBSESSED with\n
           all things Echo and Alexa. These are devices that you may not realize how
           much you'll use it until you've given\n            them a try.I have been
           spending the last few years getting our home automated. I currently use a
           combination\n            of Smartthings, IFTTT and Logitech Harmony to co
           ntrol the home. Alexa fits right into the middle of all that\n
           and gives us the ability to control our home via voice. For me, this is t
           he killer feature. I can integrate all\n            three parts of our ho
           me control right into Echo/Tap and make our house even smarter with very
           minimal setup.Read\n            more",
            'title': 'Tap Alexa on the go!'}
```

We can also search within text in a specific field in a document.

```
In [133]: amazing_reviews = review_table.search(Review.text.search("amazing"))
          len(amazing_reviews)
```

```
Out[133]: 24
```

Let's turn to the Twitter Data. Recall that each tweet and its metadata is stored in a list of statuses in `results` , the processed API response. We can iterate through the list and store each element as a document.

```
In [279]: for tweet in results['statuses']:
              twitter_table.insert(tweet)

          len(twitter_table)
```

```
Out[279]: 15
```

While we can reuse the existing instance of *Query* to query the Twitter table, we'll create a second one.

```
In [137]: Tweet = tinydb.Query()
```

Recall that the `full_text` key is paired with the actual message. Let's search for tweets containing `Columbus` .

In [139]:
```
columbus_tweets = twitter_table.search(Tweet.full_text.search("Columbus"))
len(columbus_tweets)
```
Out[139]:
```
15
```

If there were no results, try `columbus` (with a lower-case c). Each of the the tweets should include either `Columbus` or `columbus` based on how we searched for tweets using the API.

Notice that the tweet data also includes a nested user data. Each user has a location. Most document stores support searching nested data. The following searches for documents where the user's location is `Columbus, OH` . The number of results will depend on the data that was returned by the API.

In [140]:
```
users_in_columbus = twitter_table.search(Tweet.user.location == "Columbus, C
len(users_in_columbus)
```
Out[140]:
```
2
```

Analysis

There are a wide variety of analyses that can be done with unstructured data. For example, we might want to search for the prevalence of certain features within images, determine the frequency at which keywords appear in documents, or determine if a relationship exists between social media activity, location, and historical events.

A simple type of analysis we could conduct is computing word frequency. For example, we could see which words appear most often among the product reviews. While there are libraries that can be used to do this, we can use tools in the standard library.

We'll start with a *defaultdict* (https://docs.python.org/3/library/collections.html#collections.defaultdict) that will keep words a keys with the number of times that word appears as its value. We can iterate through the collection of reviews and for each review's text, we can split it by spaces to iterate word by word.

In [272]:
```
from collections import defaultdict

# create a defaultdict that uses 0 as a default value
frequency = defaultdict(int)

for review in review_table:
    # strip beginning or ending whitespace
    text = review['text'].strip()

    # split the reivew into words
    words = text.split(' ')

    for word in words:
        # increment the word's count
        # strip whitespace and user lower case
        clean_word = word.strip().lower()

        # don't store empty strings
        if clean_word:
            frequency[clean_word] += 1
```

To view the most frequent words, we can use the *sorted()* function with the dictionaries *items()* (https://docs.python.org/3/tutorial/datastructures.html#looping-techniques) method, the operator *itemgetter()* (https://docs.python.org/3/library/operator.html?highlight=operator#operator.itemgetter) function to get the second item (the value), and the keyword argument `reverse=True` to sort in descending order.

In [276]:
```
from operator import itemgetter
top = sorted(frequency.items(), key=itemgetter(1), reverse=True)
```

We can display the top 25 words using `top` and slice notation.

```
In [280]: top[:25]
```

```
Out[280]: [('the', 1065),
           ('to', 750),
           ('and', 580),
           ('it', 576),
           ('i', 573),
           ('is', 468),
           ('a', 465),
           ('for', 298),
           ('this', 285),
           ('you', 229),
           ('my', 209),
           ('with', 203),
           ('have', 200),
           ('of', 193),
           ('great', 188),
           ('tap', 183),
           ('as', 177),
           ('that', 156),
           ('but', 149),
           ('sound', 142),
           ('speaker', 134),
           ('echo', 133),
           ('amazon', 133),
           ('on', 131),
           ('alexa', 126)]
```

With text data, another type of analysis we can conduct is sentiment analysis (https://en.wikipedia.org/wiki/Sentiment_analysis) where we try to determine the mood or attitude of an individual based on the content of the message produced by that individual. In it's basic form, sentiment analysis attempts to classify text as being positive, negative, or neutral. More advanced analysis attempts to classify messages further, possibly by assigning an emotional state such as "happy" or "angry".

Because of the flexibility offered by natural language to convey an idea, it can be difficult to reliably classify the sentiment of text. For example, a simple analysis focused on keywords might accurately classify text like "Pizza is delicious" as positive. However, a statement such as "I don't dislike rain" might be more difficult. The presence of "dislike" might cause an analysis algorithm to classify this statement as negative. An algorithm that accounts for negation might determine that "not" and "dislike" result in a positive statement. In reality, this might be a neutral statement indicating a lack of a preference against rain but not quite indicating a preference for it.

Organizations can use sentiment analysis to measure customer or public opinion. Product reviews can quickly be analyzed to determine if users like or dislike the product. A company might monitor social media for posts that include the company's name and categorize the content based on sentiment - a negative post could quickly be addressed.

TextBlob (https://textblob.readthedocs.io/en/dev/), a library with features for text processing and analysis, provides a relatively simple way of detecting sentiment in text.

Before we look at an example, we should understand the concepts of *subjectivity* and *polarity*. Subjectivity is the degree to which a collection of text represents an opinion as opposed to a statement that can be proven true or false. In TextBlob, subjectivity ranges from 0 to with 0 representing objective text and 1 representing subjective text. Subjective text can also have a polarity indicating whether the sentiment is negative, neutral, or positive. A negative sentiment is indicated by a polarity of -1, neutral by 0, and positive by 1. Note that a purely object statement will have a neutral sentiment.

To compute, subjectivity and polarity, we create an instance of the *TextBlob* class and use its *sentiment* property. First, we import the *TextBlob* class itself from the textblob module.

```
In [148]: from textblob import TextBlob
```

Let's look at some examples. First, an objective, neutral statement.

```
In [149]: TextBlob("Today is Monday.").sentiment

Out[149]: Sentiment(polarity=0.0, subjectivity=0.0)
```

Next, we have subjective, negative sentence.

```
In [153]: TextBlob("Mondays are the worst.").sentiment

Out[153]: Sentiment(polarity=-1.0, subjectivity=1.0)
```

We can also create a subjective, positive sentence.

Lab 9 In the cell below, display the polarity and subjectivity of a sentence with polarity equal to 1.

```
In [157]: TextBlob("Pizza is delicious.").sentiment

Out[157]: Sentiment(polarity=1.0, subjectivity=1.0)
```

Let's examine the sentiment of the tweets we've collected. For convenience, we'll create a DataFrame with three columns: text, polarity, and subjectivity.

In [166]:
```
sentiments = pd.DataFrame(columns=['text', 'polarity', 'subjectivity'])

for tweet in twitter_table:
    text = tweet['full_text']
    sentiment = TextBlob(text).sentiment
    sentiments = sentiments.append({
        'text': text,
        'polarity': sentiment.polarity,
        'subjectivity': sentiment.subjectivity
    }, ignore_index=True)

sentiments[['polarity', 'subjectivity']]
```

Out[166]:

	polarity	subjectivity
0	0.167677	0.647811
1	0.100000	0.666667
2	-0.487500	0.587500
3	0.375000	0.500000
4	0.333333	0.358974
5	0.285714	0.535714
6	0.071429	0.133929
7	0.210000	0.326667
8	0.000000	0.000000
9	0.000000	0.000000
10	0.500000	0.750000
11	0.000000	0.100000
12	-0.083333	0.277778
13	0.000000	0.000000
14	-0.031250	0.366667

We can see calculate the mean polarity and subjectivity.

Lab 10 In the cell below, calculate the mean subjectivity and polarity.

In [167]:
```
sentiments.mean()
```

Out[167]:
```
polarity        0.096071
subjectivity    0.350114
dtype: float64
```

We can also plot polarity and subjectivity. First, we need make sure plots are presented in the notebook itself.

In [168]:
```
%matplotlib inline
```

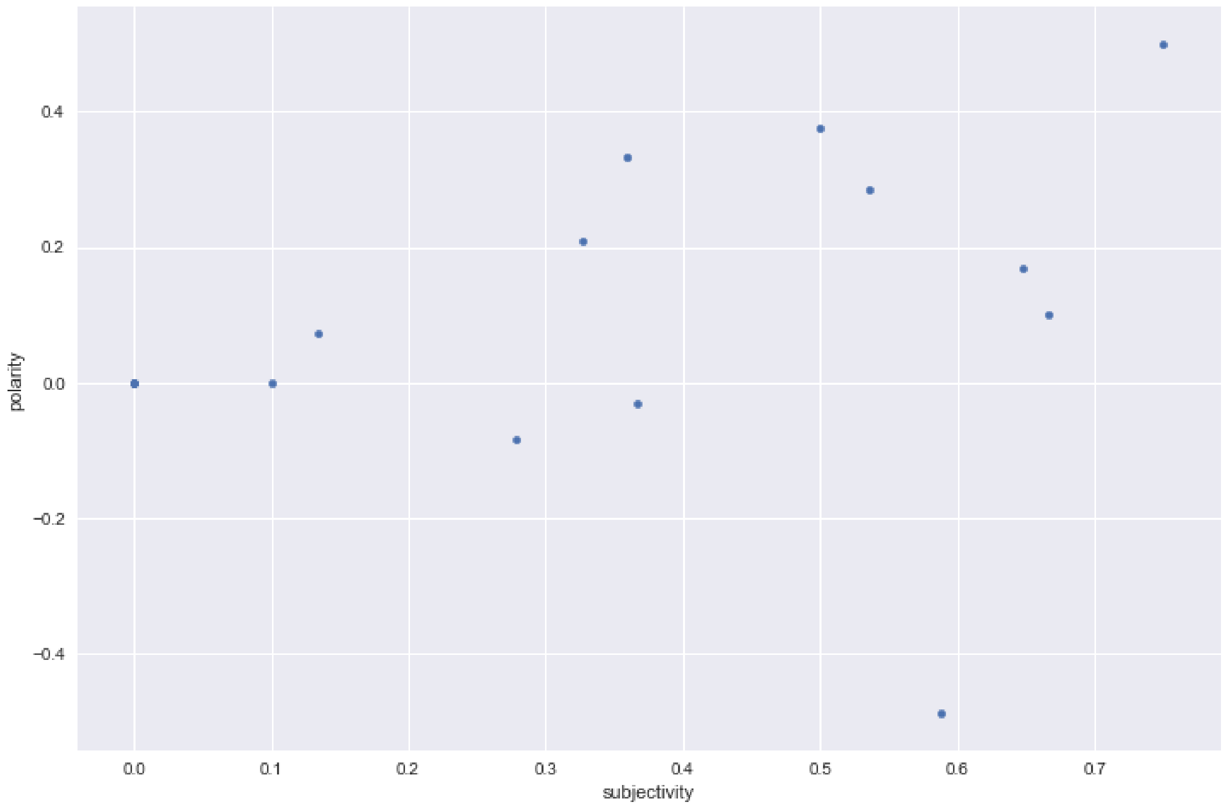
We can also use Seaborn for nicer looking plots.

In [169]:
```
import seaborn as sns
sns.set(rc={'figure.figsize':(12,8)})
```

We can use a scatter plot to see where the tweets fall in terms of polarity and subjectivity.

In [170]:
```
sentiments.plot.scatter(x='subjectivity', y='polarity')
```

Out[170]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x11ce7b898>
```



Let's look at the product reviews. We can again iterate through the review table and calculate the sentiment of each review. In addition to capturing the review text and sentiment, we will also keep track of the the numeric rating.

```
In [190]: sentiments = pd.DataFrame(columns=['text', 'rating',
                                             'polarity', 'subjectivity'])

          for review in review_table:
              text = review['text']
              sentiment = TextBlob(text).sentiment
              sentiments = sentiments.append({
                  'text': text,
                  'rating': review['rating'],
                  'polarity': sentiment.polarity,
                  'subjectivity': sentiment.subjectivity
              }, ignore_index=True)

          sentiments[['rating', 'polarity', 'subjectivity']].head()
```

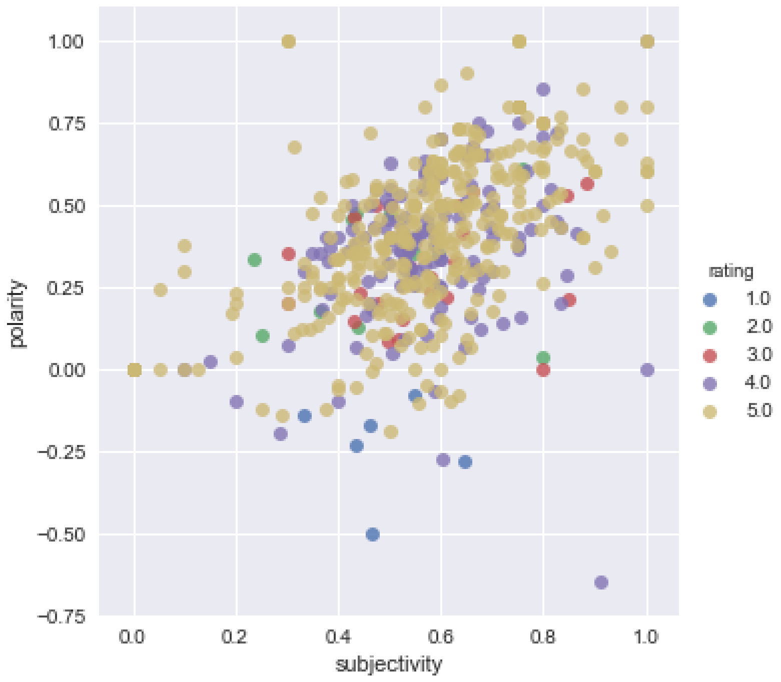
Out[190]:

	rating	polarity	subjectivity
0	5.0	0.049031	0.454416
1	5.0	0.119940	0.345933
2	5.0	0.279053	0.621882
3	5.0	0.251970	0.537121
4	5.0	0.380000	0.690000

Let's look at a scatter plot of polarity and subjectivity colored by rating.

```
In [203]: sns.lmplot(x='subjectivity', y="polarity", hue='rating',
                     data=sentiments, fit_reg=False)
```

```
Out[203]: <seaborn.axisgrid.FacetGrid at 0x125135240>
```



While it looks like subjectivity and polarity might be somewhat related, it doesn't look like these have much of an affect on rating. We can look at the correlation coefficients matrix and the pair plot for more insight.

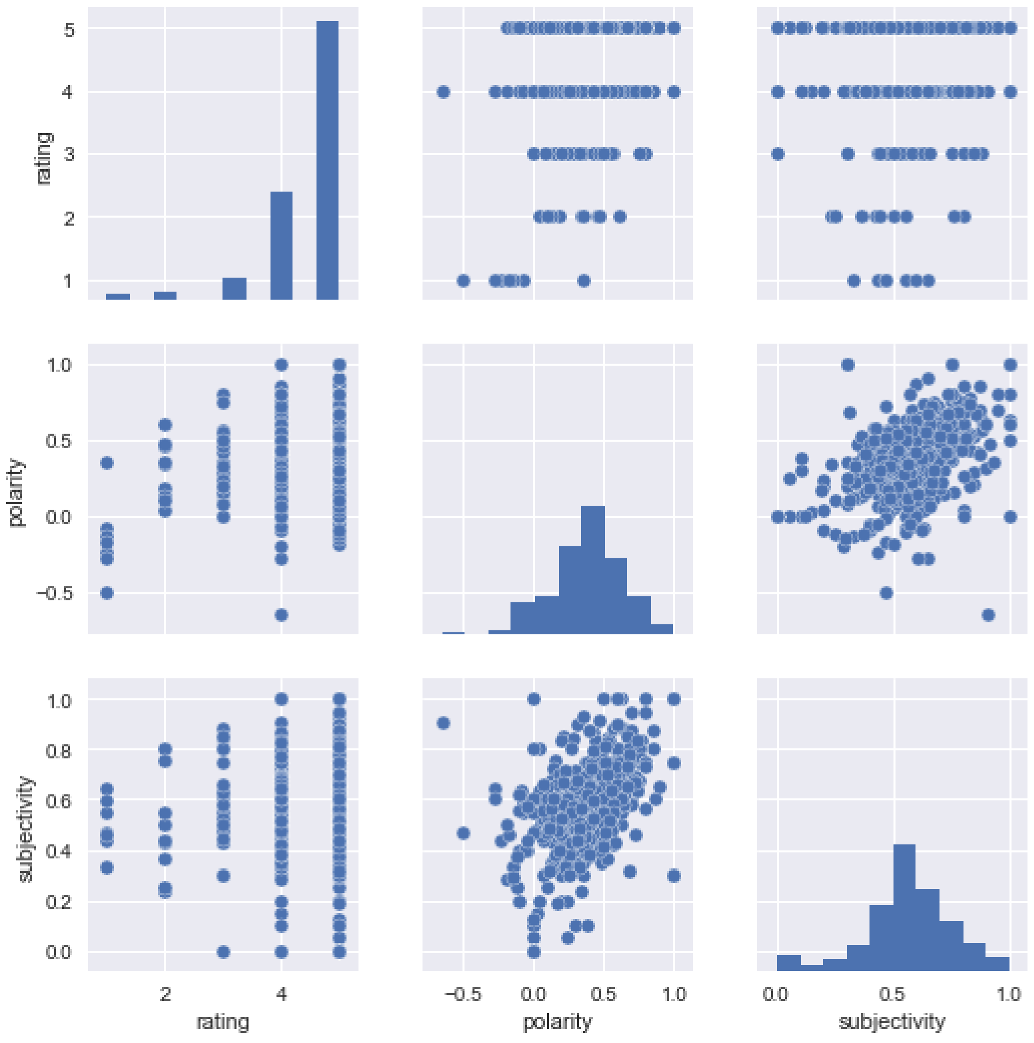
```
In [205]: sentiments.corr()
```

```
Out[205]:
```

	rating	polarity	subjectivity
rating	1.000000	0.215769	0.042229
polarity	0.215769	1.000000	0.512518
subjectivity	0.042229	0.512518	1.000000

```
In [204]: sns.pairplot(sentiments)
```

```
Out[204]: <seaborn.axisgrid.PairGrid at 0x125464dd8>
```



It doesn't appear that there is a strong relationship among `polarity` , `subjectivity` , and `rating` .

Lab Answers

1.
```
with open('./data/05-reviews.html') as infile:
    reviews = infile.readlines()

for line in reviews[:100]:
    print(line)
```

2.
```
divs = body.findall("div")
len(divs)
```

3.
```
div.attrib
```

4.
```
reviews = pd.DataFrame(columns=['title', 'rating', 'text'])

with open('./data/05-reviews.html') as infile:
    soup = BeautifulSoup(infile, 'html.parser')

for review_div in soup.find_all('div', {'class': 'review_contain
er'}):
    review_data = {
        'title': review_div.find('span', {'class': 'title'}).tex
t,
        'rating': float(review_div.find('span', {'class': 'ratin
g'}).text),
        'text': review_div.find('span', {'class': 'review'}).tex
t
    }
    reviews = reviews.append([review_data], ignore_index=True)

reviews.head()
```

5. *Answers will vary*

6.
```
results['statuses'][0]['full_text']
```

7.
```
review_table = db.table('review')
```

8.
```
five_stars[0]
```

9. *Answers will vary*

```
TextBlob("Pizza is delicious.").sentiment
```

10.
```
sentiments.mean()
```

Next Steps

We've spend a lot of time examining and exploring data, in the next units, we'll look at how we can report on the information we extract.

Resources and Further Reading

- How to Use Web APIs in Python 3 (https://www.digitalocean.com/community/tutorials/how-to-use-web-apis-in-python-3)
- Natural Language Processing with Python (http://www.nltk.org/book/)
- Python Data Analysis, 2nd Edition, by Fandango; Chapter 9 - Analyzing Textual Data and Social Media (Safari Books) (http://proquest.safaribooksonline.com.cscc.ohionet.org/book/programming/python/9781787127 data-analysis-second-edition/ch09_html?uicode=ohlink)

Exercises

1. When examining the polarity and subjectivity of the product reviews, we concluded that there wasn't a strong relationship between the two variables. Is the relationship stronger or weaker if we look at the correlation coefficient for a specific value for rating? Iterate through the unique values of the `rating` column and use this value to filter the DataFrame for records with only that value and calculate the correlation coefficients matrix for `subjectivity` and `polarity` for each rating.
2. Using the Twitter API, search for tweets containing a company or product of your choice. What is the average sentiment of tweets containing the search term?

```
In [ ]:
```