

Mining-the-Social-Web-2nd-Edition (/github/ptwobrussell/Mining-the-Social-Web-2nd-Edition/tree/master)
/ ipynb (/github/ptwobrussell/Mining-the-Social-Web-2nd-Edition/tree/master/ipynb)

Mining the Social Web, 2nd Edition

Chapter 1: Mining Twitter: Exploring Trending Topics, Discovering What People Are Talking About, and More

This IPython Notebook provides an interactive way to follow along with and explore the numbered examples from *Mining the Social Web (2nd Edition)* (<http://bit.ly/135dHfs>). The intent behind this notebook is to reinforce the concepts from the sample code in a fun, convenient, and effective way. This notebook assumes that you are reading along with the book and have the context of the discussion as you work through these exercises.

In the somewhat unlikely event that you've somehow stumbled across this notebook outside of its context on GitHub, [you can find the full source code repository here](http://bit.ly/16kGNyb) (<http://bit.ly/16kGNyb>).

Copyright and Licensing

You are free to use or adapt this notebook for any purpose you'd like. However, please respect the [Simplified BSD License](https://github.com/ptwobrussell/Mining-the-Social-Web-2nd-Edition/blob/master/LICENSE.txt) (<https://github.com/ptwobrussell/Mining-the-Social-Web-2nd-Edition/blob/master/LICENSE.txt>) that governs its use.

Twitter API Access

Twitter implements OAuth 1.0A as its standard authentication mechanism, and in order to use it to make requests to Twitter's API, you'll need to go to <https://dev.twitter.com/apps> (<https://dev.twitter.com/apps>) and create a sample application. There are four primary identifiers you'll need to note for an OAuth 1.0A workflow: consumer key, consumer secret, access token, and access token secret. Note that you will need an ordinary Twitter account in order to login, create an app, and get these credentials.



If you are taking advantage of the virtual machine experience for this chapter that is powered by Vagrant, you should just be able to execute the code in this notebook without any worries whatsoever about installing dependencies. If you are running the code from your own development environment, however, be advised that these examples in this chapter take

advantage of a Python package called `twitter` (<https://github.com/sixohsix/twitter>) to make API calls. You can install this package in a terminal with `pip` (<https://pypi.python.org/pypi/pip>) with the command `pip install twitter`, preferably from within a Python virtual environment (<https://pypi.python.org/pypi/virtualenv>).

Once installed, you should be able to open up a Python interpreter (or better yet, your IPython (<http://ipython.org/>) interpreter) and get rolling.

Example 1. Authorizing an application to access Twitter account data

In []:

```
import twitter

# XXX: Go to http://dev.twitter.com/apps/new to create an app and get values
# for these credentials, which you'll need to provide in place of these
# empty string values that are defined as placeholders.
# See https://dev.twitter.com/docs/auth/oauth for more information
# on Twitter's OAuth implementation.

CONSUMER_KEY = ''
CONSUMER_SECRET = ''
OAUTH_TOKEN = ''
OAUTH_TOKEN_SECRET = ''

auth = twitter.oauth.OAuth(OAUTH_TOKEN, OAUTH_TOKEN_SECRET,
                           CONSUMER_KEY, CONSUMER_SECRET)

twitter_api = twitter.Twitter(auth=auth)

# Nothing to see by displaying twitter_api except that it's now a
# defined variable

print twitter_api
```

Example 2. Retrieving trends

In []:

```

# The Yahoo! Where On Earth ID for the entire world is 1.
# See https://dev.twitter.com/docs/api/1.1/get/trends/place and
# http://developer.yahoo.com/geo/geoplanet/

WORLD_WOE_ID = 1
US_WOE_ID = 23424977

# Prefix ID with the underscore for query string parameterization.
# Without the underscore, the twitter package appends the ID value
# to the URL itself as a special case keyword argument.

world_trends = twitter_api.trends.place(_id=WORLD_WOE_ID)
us_trends = twitter_api.trends.place(_id=US_WOE_ID)

print world_trends
print
print us_trends

```

Example 3. Displaying API responses as pretty-printed JSON

In []:

```

import json

print json.dumps(world_trends, indent=1)
print
print json.dumps(us_trends, indent=1)

```

Example 4. Computing the intersection of two sets of trends

In []:

```

world_trends_set = set([trend['name']
                        for trend in world_trends[0]['trends']])

us_trends_set = set([trend['name']
                     for trend in us_trends[0]['trends']])

common_trends = world_trends_set.intersection(us_trends_set)

print common_trends

```

Example 5. Collecting search results

In []:

```

# Import unquote to prevent url encoding errors in next_results
from urllib import unquote

# XXX: Set this variable to a trending topic,
# or anything else for that matter. The example query below
# was a trending topic when this content was being developed
# and is used throughout the remainder of this chapter.

q = '#MentionSomeoneImportantForYou'

count = 100

# See https://dev.twitter.com/docs/api/1.1/get/search/tweets

search_results = twitter_api.search.tweets(q=q, count=count)

statuses = search_results['statuses']

# Iterate through 5 more batches of results by following the cursor
for _ in range(5):
    print "Length of statuses", len(statuses)
    try:
        next_results = search_results['search_metadata']['next_results']
    except KeyError, e: # No more results when next_results doesn't exist
        break

    # Create a dictionary from next_results, which has the following form:
    # ?max_id=313519052523986943&q=NCAA&include_entities=1
    kwargs = dict([ kv.split('=') for kv in unquote(next_results[1:]).split('&') ])

    search_results = twitter_api.search.tweets(**kwargs)
    statuses += search_results['statuses']

# Show one sample search result by slicing the list...
print json.dumps(statuses[0], indent=1)

```

Note: Should you desire to do so, you can load the same set of search results that are illustrated in the text of *Mining the Social Web* by executing the code below that reads a snapshot of the data and stores it into the same statuses variable as was defined above. Alternatively, you can choose to skip execution of this cell in order to follow along with your own data.

In []:

```
import json
statuses = json.loads(open('resources/ch01-twitter/data/MentionSomeoneImport

# The result of the list comprehension is a list with only one element that
# can be accessed by its index and set to the variable t
t = [ status
      for status in statuses
      if status['id'] == 316948241264549888 ] [0]

# Explore the variable t to get familiarized with the data structure...

print t['retweet_count']
print t['retweeted_status']

# Can you find the most retweeted tweet in your search results? Try do do it
```

Example 6. Extracting text, screen names, and hashtags from tweets

In []:

```
status_texts = [ status['text']
                  for status in statuses ]

screen_names = [ user_mention['screen_name']
                  for status in statuses
                  for user_mention in status['entities']['user_mentions'] ]

hashtags = [ hashtag['text']
              for status in statuses
              for hashtag in status['entities']['hashtags'] ]

# Compute a collection of all words from all tweets
words = [ w
          for t in status_texts
          for w in t.split() ]

# Explore the first 5 items for each...

print json.dumps(status_texts[0:5], indent=1)
print json.dumps(screen_names[0:5], indent=1)
print json.dumps(hashtags[0:5], indent=1)
print json.dumps(words[0:5], indent=1)
```

Example 7. Creating a basic frequency distribution from the words in tweets

In []:

```
from collections import Counter

for item in [words, screen_names, hashtags]:
    c = Counter(item)
    print c.most_common()[:10] # top 10
    print
```

Example 8. Using prettytable to display tuples in a nice tabular format

In []:

```
from prettytable import PrettyTable

for label, data in (('Word', words),
                   ('Screen Name', screen_names),
                   ('Hashtag', hashtags)):
    pt = PrettyTable(field_names=[label, 'Count'])
    c = Counter(data)
    [ pt.add_row(kv) for kv in c.most_common()[:10] ]
    pt.align[label], pt.align['Count'] = 'l', 'r' # Set column alignment
    print pt
```

Example 9. Calculating lexical diversity for tweets

In []:

```
# A function for computing lexical diversity
def lexical_diversity(tokens):
    return 1.0*len(set(tokens))/len(tokens)

# A function for computing the average number of words per tweet
def average_words(statuses):
    total_words = sum([ len(s.split()) for s in statuses ])
    return 1.0*total_words/len(statuses)

print lexical_diversity(words)
print lexical_diversity(screen_names)
print lexical_diversity(hashtags)
print average_words(status_texts)
```

Example 10. Finding the most popular retweets

In []:

```

retweets = [
    # Store out a tuple of these three values ...
    (status['retweet_count'],
     status['retweeted_status']['user']['screen_name'],
     status['text'])


    # ... for each status ...
    for status in statuses

    # ... so long as the status meets this condition.
    if status.has_key('retweeted_status')
]

# Slice off the first 5 from the sorted results and display each item in the

pt = PrettyTable(field_names=['Count', 'Screen Name', 'Text'])
[ pt.add_row(row) for row in sorted(retweets, reverse=True)[:5] ]
pt.max_width['Text'] = 50
pt.align= 'l'
print pt

```



Example 11. Looking up users who have retweeted a status

In []:

```

# Get the original tweet id for a tweet from its retweeted_status node
# and insert it here in place of the sample value that is provided
# from the text of the book

_retweets = twitter_api.statuses.retweets(id=317127304981667841)
print [r['user']['screen_name'] for r in _retweets]

```

Example 12. Plotting frequencies of words

In []:

```

word_counts = sorted(Counter(words).values(), reverse=True)

plt.loglog(word_counts)
plt.ylabel("Freq")
plt.xlabel("Word Rank")

```

Example 13. Generating histograms of words, screen names, and hashtags

In []:

```

for label, data in (('Words', words),
                   ('Screen Names', screen_names),
                   ('Hashtags', hashtags)):

    # Build a frequency map for each set of data
    # and plot the values
    c = Counter(data)
    plt.hist(c.values())

    # Add a title and y-label ...
    plt.title(label)
    plt.ylabel("Number of items in bin")
    plt.xlabel("Bins (number of times an item appeared)")

    # ... and display as a new figure
    plt.figure()

```

Example 14. Generating a histogram of retweet counts

In []:

```

# Using underscores while unpacking values in
# a tuple is idiomatic for discarding them

counts = [count for count, _, _ in retweets]

plt.hist(counts)
plt.title("Retweets")
plt.xlabel('Bins (number of times retweeted)')
plt.ylabel('Number of tweets in bin')

print counts

```

Note: This histogram gives you an idea of how many times tweets are retweeted with the x-axis defining partitions for tweets that have been retweeted some number of times and the y-axis telling you how many tweets fell into each bin. For example, a y-axis value of 5 for the "15-20 bin" on the x-axis means that there were 5 tweets that were retweeted between 15 and 20 times.

Here's another variation that transforms the data using the (automatically imported from numpy) log function in order to improve the resolution of the plot.

In []:

```
# Using underscores while unpacking values in  
# a tuple is idiomatic for discarding them  
  
counts = [count for count, _, _ in retweets]  
  
# Taking the Log of the *data values* themselves can  
# often provide quick and valuable insight into the  
# underlying distribution as well. Try it back on  
# Example 13 and see if it helps.  
  
plt.hist(log(counts))  
plt.title("Retweets")  
plt.xlabel('Log[Bins (number of times retweeted)]')  
plt.ylabel('Log[Number of tweets in bin]')  
  
print log(counts)
```