# Social Media Mining in Python

This chapter is all about social media. Though it's not directly related to NLTK / NLP, social data is also a very rich source of unstructured text data. So, as NLP enthusiasts, we should have the skill to play with social data. We will try to explore how to gather relevant data from some of the most popular social media platforms. We will cover how to gather data from Twitter, Facebook, and so on using Python APIs. We will explore some of the most common use cases in the context of social media mining, such as trending topics, sentiment analysis, and so on.

You already learned a lot of topics under the concepts of natural language processing and machine learning in the last few chapters. We will try to build some of the applications around social data in this chapter. We will also provide you with some of the best practices to deal with social data, and look at social data from the context of graph visualization.

There is a graph that underlies social media and most of the graph-based problems can be formulated as information flow problems and finding out the busiest node in the graph. Some of the problems such as trending topics, influencer detection, and sentiment analysis are examples of these. Let's take some of these use cases, and build some cool applications around these social networks.

By the end of this chapter,:

* You should be able to collect data from any social media using APIs.
* You will also learn to formulate the data in a structured format and how to build some amazing applications.
* Lastly, we will be able to visualize and gain meaningful insight out of social media.

## Data collection

The most important objective of this chapter is to gather data across some of the most common social networks. We will look mainly at Twitter and Facebook and try to give you enough details about the API and how to effectively use them to get relevant data. We will also talk about the data dictionary for scrapped data, and how we can build some cool apps using some of the stuff we learned so far.

### Twitter

We will start with one of the most popular and open social media that is completely public. This means that practically, you can gather entire Twitter stream, which is payable, while you can capture one percent of the stream for free. In the context of business, Twitter is a very rich resource of information such as public sentiments and emerging topics.

Let's get directly to face the main challenge of getting the tweets relevant to your use case.

The following is the repository of many Twitter libraries. These libraries are not verified by Twitter, but run on the Twitter API.

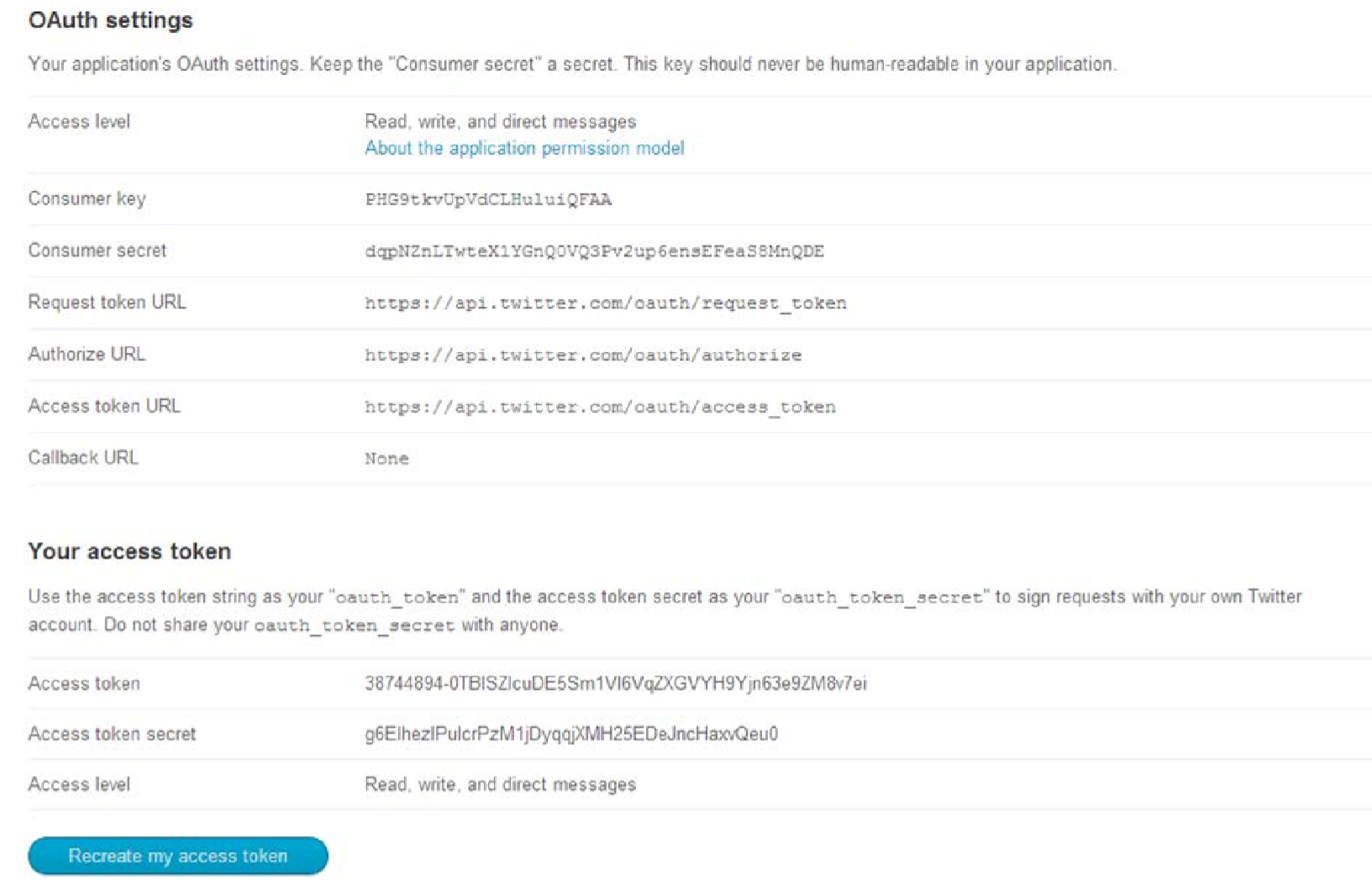
[https://dev.twitter.com/overview/api/twitterlibraries](https://dev.twitter.com/overview/api/twitter-libraries)

There are more than 10 Python libraries there. So pick and choose the one you like. I generally use Tweepy and we will use it to run the examples in this book. Most of the libraries are wrappers around the Twitter API, and the parameters and signatures of all these are roughly the same.

The simplest way to install Tweepy is to install it using pip: **$ pip install tweepy**

The hard way is to build it from source. The GitHub link to Tweepy is: <https://github.com/tweepy/tweepy>.

To get Tweepy to work, you have to create a developer account with Twitter and get the access tokens for your application. Once you complete this, you will get your credentials and below these, the keys. Go through [https://apps.twitter.com/ app/new](https://apps.twitter.com/app/new) for registration and access tokens. The following snapshot shows the access tokens:



We will start with a very simple example to gather data using Twitter's streaming API. We are using Tweepy to capture the Twitter stream to gather all the tweets related to the given keywords:

**tweetdump.py >>>from tweepy.streaming import StreamListener**

**>>>from tweepy import OAuthHandler**

**>>>from tweepy import Stream**

**>>>import sys**

**>>>consumer\_key = 'ABCD012XXXXXXXXx'**

**>>>consumer\_secret = 'xyz123xxxxxxxxxxxxx'**

**>>>access\_token = '000000-ABCDXXXXXXXXXXX'**

**>>>access\_token\_secret ='XXXXXXXXXgaw2KYz0VcqCO0F3U4' >>>class StdOutListener(StreamListener):**

**>>> def on\_data(self, data):**

**>>> with open(sys.argv[1],'a') as tf:**

**>>> tf.write(data)**

**>>> return**

**>>> def on\_error(self, status):**

**>>> print(status) >>>if \_\_name\_\_ == '\_\_main\_\_':**

**>>> l = StdOutListener()**

**>>> auth = OAuthHandler(consumer\_key, consumer\_secret)**

**>>> auth.set\_access\_token(access\_token, access\_token\_secret)**

**>>> stream = Stream(auth, l)**

**>>> stream.filter(track=['Apple watch'])**

In the preceding code, we used the same code given in the example of Tweepy, with a little modification. This is an example where we use the streaming API of Twitter, where we track **Apple Watch**. Twitter's streaming API provides you the facility of conducting a search on the actual Twitter stream and you can consume a maximum of one percent of the stream using this API.

In the preceding code, the main parts that you need to understand are the first and last four lines. In the initial lines, we are specifying the access tokens and other keys that we generated in the previous section. In the last four lines, we create a listener to consume the stream. In the last line, we use stream.filter to filter Twitter for keywords that we have put in the track. We can specify multiple keywords here. This will result in all the tweets that contain the term Apple Watch for our running example.

In the following example, we will load the tweets we have collected, and have a look at the tweet structure and how to extract meaningful information from it. A typical tweet JSON structure looks similar to:

**{**

**"created\_at":"Wed May 13 04:51:24 +0000 2015",**

**"id":598349803924369408,**

**"id\_str":"598349803924369408",**

**"text":"Google launches its first Apple Watch app with News & Weather http:\/\/t.co\/o1XMBmhnH2",**

**"source":"\u003ca href=\"http:\/\/ifttt.com\" rel=\"nofollow\"\ u003eIFTTT\u003c\/a\u003e", "truncated":false,**

**"in\_reply\_to\_status\_id":null,**

**"user":{**

**"id":1461337266,**

**"id\_str":"1461337266",**

**"name":"vestihitech \u0430\u0432\u0442\u043e\u043c\u0430\u0442",**

**"screen\_name":"vestihitecha",**

**"location":"",**

**"followers\_count":20,**

**"friends\_count":1,**

**"listed\_count":4,**

**""statuses\_count":7442,**

**"created\_at":"Mon May 27 05:51:27 +0000 2013",**

**"utc\_offset":14400,**

**},**

**,**

**"geo":{ "latitude" : 51.4514285, "longitude"=-0.99**

**}**

**"place":"Reading, UK",**

**"contributors":null,**

**"retweet\_count":0,**

**"favorite\_count":0,**

**"entities":{**

**"hashtags":["apple watch", "google"**

**],**

**"trends":[**

**],**

**"urls":[**

**{**

**"url":"http:\/\/t.co\/o1XMBmhnH2",**

**"expanded\_url":"http:\/\/ift.tt\/1HfqhCe",**

**"display\_url":"ift.tt\/1HfqhCe",**

**"indices":[**

**66,**

**88**

**]**

**}**

**],**

**"user\_mentions":[**

**],**

**"symbols":[**

**]**

**},**

**"favorited":false,**

**"retweeted":false,**

**"possibly\_sensitive":false,**

**"filter\_level":"low",**

**"lang":"en",**

**"timestamp\_ms":"1431492684714"**

**}**

**]**

## Data extraction

Some of the most commonly used fields of interest in data extraction are:

* text: This is the content of the tweet provided by the user
* user: These are some of the main attributes about the user, such as username, location, and photos
* Place: This is where the tweets are posted, and also the geo coordinates
* Entities: Effectively, these are the hashtags and topics that a user attaches to his / her tweets

Every attribute in the previous figure can be a good use case for some of the social mining exercises done in practice. Let's jump onto the topic of how we can get to these attributes and convert them to a more readable form, or how we can process some of these:

**Source**: tweetinfo.py

**>>>import json**

**>>>import sys**

**>>>tweets = json.loads(open(sys.argv[1]).read())**

**>>>tweet\_texts = [ tweet['text']\**

**for tweet in tweets ] >>>tweet\_source = [tweet ['source'] for tweet in tweets]**

**>>>tweet\_geo = [tweet['geo'] for tweet in tweets]**

**>>>tweet\_locations = [tweet['place'] for tweet in tweets]**

**>>>hashtags = [ hashtag['text'] for tweet in tweets for hashtag in tweet['entities']['hashtags'] ] >>>print tweet\_texts**

**>>>print tweet\_locations**

**>>>print tweet\_geo**

**>>>print hashtags**

The output of the preceding code will give you, as expected, four lists in which all the tweet content is in tweet\_texts and the location of the tweets and hashtags.

In the code, we are just loading a JSON output generated using json. loads(). I would recommend you to use an online tool such as Json Parser (<http://json.parser.online.fr/>) to get an idea of what your JSON looks like and what are its attributes (key and value).

Next, if you look, there are different levels in the JSON, where some of the attributes such as text have a direct value, while some of them have more nested information. This is the reason you see, where when we are looking at hashtags, we have to iterate one more level, while in case of text, we just fetch the values. Since our file actually has a list of tweets, we have to iterate that list to get all the tweets, while each tweet object will look like the example tweet structure.

### Trending topics

Now, if we look for trending topics in this kind of a setup. One of the simplest ways to find them could be to look for frequency distribution of words across tweets. We already have a list of tweet\_text that contains the tweets:

**>>>import nltk**

**>>>from nltk import word\_tokenize,sent\_tokenize**

**>>>from nltk import FreqDist**

**>>>tweets\_tokens = []**

**>>>for tweet in tweet\_text:**

**>>> tweets\_tokens.append(word\_tokenize(tweet))**

**>>>Topic\_distribution = nltk.FreqDist(tweets\_tokens)**

**>>>Freq\_dist\_nltk.plot(50, cumulative=False)**

One other more complex way of doing this could be the use of the part of speech tagger that you learned in *Chapter 3*, *Part of Speech Tagging*. The theory is that most of the time, topics will be nouns or entities. So, the same exercise can be done like this. In the preceding code, we read every tweet and tokenize it, and then use POS as a filter to only select nouns as topics:

**>>>import nltk**

**>>>Topics = [] >>>for tweet in tweet\_text:**

**>>> tagged = nltk.pos\_tag(word\_tokenize(tweet))**

**>>> Topics\_token = [word for word,pos in ] in tagged if pos in**

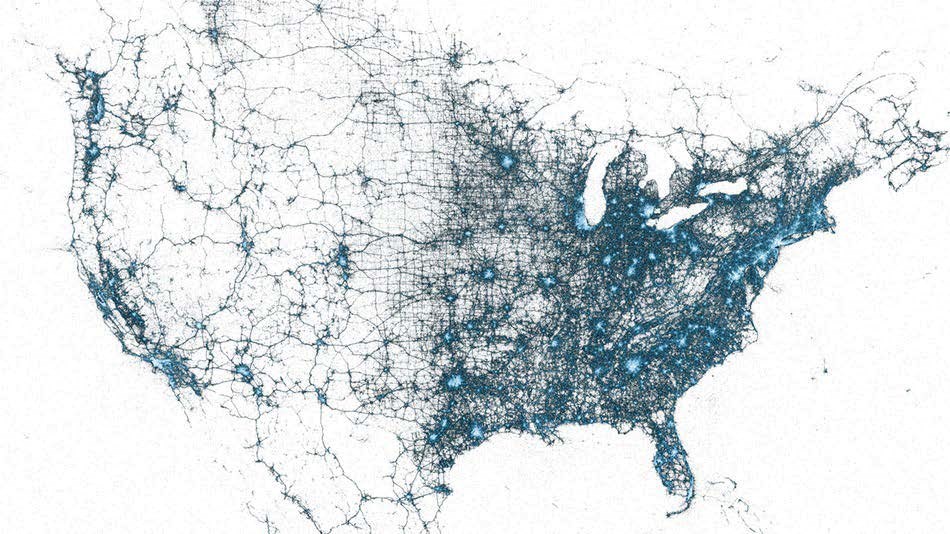
**['NN','NNP']**

**>>> print Topics\_token**

If we want to see a much cooler example, we can gather tweets across time and then generate plots. This will give us a very clear idea of the trending topics. For example, the data we are looking for is "Apple Watch". This word should peak on the day when Apple launched Apple Watch and the day they started selling it. However, it will be interesting to see what kind of topics emerged apart from those, and how they trended over time.

## Geovisualization

One of the other common application of social media is geo-based visualization. In the tweet structure, we saw attributes named geo, longitude, and latitude. Once you have access to these values, it is very easy to use some of the common visualization libraries, such as **D3**, to come up with something like this:



This is just an example of what we can achieve with these kind of visualizations; this was the visualization of a tweet in the U.S. We can clearly see the areas of increased intensity in eastern places such as New York. Now, a similar analysis done by a company on the customers can give a clear insight about which are some of the most popular places liked by our customer base. We can text mine these tweets for sentiment, and then we can infer insights about customers as to in which states they are not happy with the company and so on.

### Influencers detection

Detection of important nodes in the social graph that has a lot of importance is another great problem in the context of social graphs. So, if we have millions of tweets streaming about our company, then one important use case would be to gather the most influential customers in the social media, and then target them for branding, marketing, or improving customer engagement.

In the case of Twitter, this goes back to the graph theory and concept of PageRank, where in a given node, if the ratio of outdegree and indegree is high, then that node is an influencer. This is very intuitive since people who have more followers than the number of people they follow are typically, influencers. One company, **KLOUT,** [(https://klout.com/)](https://klout.com/) has been focusing on a similar problem. Let's write a very basic and intuitive algorithm to calculate Klout's score:

**>>>klout\_scores = [ (tweet['user']['followers\_count]/ tweet['user']**

**['friends\_count'],tweet['user']) for tweet in tweets ]**

Some of the examples where we worked on Twitter will hold exactly the same modification of content field. We can build a trending topic example with Facebook posts. We can also visualize Facebook users and post on the **geomap** and influencer kind of use cases. In fact, in the next section, we will see a variation of this in the context of Facebook.

### Facebook

Facebook is a bit more personal, and somewhat private social network. Facebook does not allow you to gather the feeds/posts of the user simply for security and privacy reasons. So, Facebook's graph API has a limited way of accessing the feeds of the given page. I would recommend you to go to [https://developers.facebook. com/docs/graph-api/using-graph-api/v2.3](https://developers.facebook.com/docs/graph-api/using-graph-api/v2.3) for better understanding.

The next question is how to access the Graph API using Python and how to get started with it. There are many wrappers written over Facebook's API, and we will use one the most common Facebook SDK: **$ pip install facebook-sdk**

You can also install it through: <https://github.com/Pythonforfacebook/facebook-sdk>.

The next step is to get the access token for the application while Facebook treats every API call as an application. Even for this data collection step, we will pretend to be an application.

To get your token, go to: <https://developers.facebook.com/tools/explorer>.

We are all set now! Let's start with one of the most widely used Facebook graph APIs. In this API, Facebook provides a graph-based search for pages, users, events, places, and so on. So, the process of getting to the post becomes a two-stage process, where we have to look for a specific pageid / userid related to our topic of interest, and then we will be able to access the feeds of that page. One simple use case for this kind of an exercise could be to use the official page of a company and look for customer complaints. The way to go about this is:

**>>>import facebook**

**>>>import json**

**>>>fo = open("fdump.txt",'w')**

**>>>ACCESS\_TOKEN = 'XXXXXXXXXXX' # https://developers.facebook.com/tools/ explorer >>>fb = facebook.GraphAPI(ACCESS\_TOKEN)**

**>>>company\_page = "326249424068240"**

**>>>content = fb.get\_object(company\_page)**

**>>>fo.write(json.dumps(content))**

The code will attach the token to the Facebook Graph API and then we will make a REST call to Facebook. The problem with this is that we have to have the ID of the given page with us beforehand. The code which will attach the token is as follows:

**"website":"www.dimennachildrenshistorymuseum.org",**

**"can\_post":true,**

**"category\_list":[**

**{**

**"id":"244600818962350",**

**"name":"History Museum"**

**},**

**{**

**"id":"187751327923426",**

**"name":"Educational Organization"**

**}**

**],**

**"likes":1793,**

**},**

**"id":"326249424068240",**

**"category":"Museum/art gallery",**

**"has\_added\_app":false,**

**"talking\_about\_count":8,**

**"location":{**

**"city":"New York",**

**"zip":"10024",**

**"country":"United States",**

**"longitude":-73.974413,**

**"state":"NY",**

**"street":"170 Central Park W",**

**"latitude":40.779236**

**},**

**"is\_community\_page":false,**

**"username":"nyhistorykids",**

**"description":"The first-ever museum bringing American history to life through the eyes of children, where kids plus history equals serious fun! Kids of all ages can practice their History Detective skills at the DiMenna Children's History Museum and:\n\n\u2022 discover the past through six historic figure pavilions\n\n\u2022!", "hours":{**

**""thu\_1\_close":"18:00"**

**},**

**"phone":"(212) 873-3400",**

**"link":"https://www.facebook.com/nyhistorykids",**

**"price\_range":"$ (0-10)",**

**"checkins":1011,**

**"about":"The DiMenna Children' History Museum is the first-ever museum bringing American history to life through the eyes of children. Visit it inside the New-York Historical Society!", "name":"New-York Historical Society DiMenna Children's History Museum",**

**"cover":{**

**"source":"https://scontent.xx.fbcdn.net/hphotos-xpf1/t31.0-8/s720x720/104**

**9166\_672951706064675\_339973295\_o.jpg",**

**"cover\_id":"672951706064675",**

**"offset\_x":0,**

**"offset\_y":54,**

**"id":"672951706064675"**

**},**

**"were\_here\_count":1011,**

**"is\_published":true**

**},**

Here, we showed a similar schema for the Facebook data as we did for Twitter, and now we can see what kind of information is required for our use case. In most of the cases, the user post, category, name, about, and likes are some of the important fields. In this example, we are showing a page of a museum, but in a more businessdriven use case, a company page has a long list of posts and other useful information that can give some great insights about it.

Let's say I have a Facebook page for my organization xyz.org and I want to know about the users who complained about me on the page; this is good for a use case such as complaint classification. The way to achieve the application now is simple enough. You need to look for a set of keywords in fdump.txt, and it can be as complex as scoring using a text classification algorithm we learned in *Chapter 6, Text Classification*.

The other use case could be to look for a topic of interest, and then to look for the resulting pages for open posts and comments. This is exactly analogous to searching using the graph search bar on your Facebook home page. However, the power of doing this programmatically is that we can conduct these searches and then each page can be recursively parsed for use comments. The code for searching user data is as follows: User search

**>>>fb.request("search", {'q' : 'nitin', 'type' : 'user'}) Place based on the nearest location.**

**>>>fb.request("search", {'q' : 'starbucks', 'type' : 'place'}) Look for open pages.**

**>>>fb.request("search", {'q' : 'Stanford university', 'type' : page}) Look for event matching to the key word.**

**>>>fb.request("search", {'q' : 'beach party', 'type' : 'event'})**

Once we have dumped all the relevant data into a structured format, we can apply some of the concepts we learned when we went through the topics of NLP and machine learning. Let's pick the same use case of finding posts, that will mostly be complaints on a Facebook page.

I assume that we now have the data in the following format:

|  |  |
| --- | --- |
| **Userid** | **FB Post** |
| XXXX0001 | The product was pathetic and I tried reaching out to your customer care, but nobody responded |
| XXXX002 | Great work guys |
| XXXX003 | Where can I call to get my account activated ??? Really bad service |

We will go back to the same example we had in *Chapter 6*, *Text Classification*, where we built a text classifier to detect whether the **SMS** (text message) was spam. Similarly, we can create training data using this kind of data, where from the given set of posts, we will ask manual taggers to tag the comments that are complaints and the ones that are not. Once we have significant training data, we can build the same text classifier:

**fb\_classification.py >>>from sklearn.feature\_extraction.text import TfidfVectorizer**

**>>>vectorizer = TfidfVectorizer(min\_df=2, ngram\_range=(1, 2), stop\_ words='english', strip\_accents='unicode', norm='l2') >>>X\_train = vectorizer.fit\_transform(x\_train)**

**>>>X\_test = vectorizer.transform(x\_test)**

**>>>from sklearn.linear\_model import SGDClassifier**

**>>>clf = SGDClassifier(alpha=.0001, n\_iter=50).fit(X\_train, y\_train)**

**>>>y\_pred = clf.predict(X\_test)**

Let's assume that these three are the only samples. We can tag 1st and 3rd to be classified as complaints, while 2nd will not be a complaint. Although we will build a vectorizer of unigram and bigram in the same way, we can actually build a classifier using the same process. I ignored some of the preprocessing steps here. You can use the same process as discussed in *Chapter 6*, *Text Classification*. In some of the cases, it will be hard/expensive to get training data like this. In some of these cases, we can apply either an unsupervised algorithm, such as text clustering or topic modeling. The other way is to use some different dataset that is openly available and build model on that and apply it here. For example, in the same use case, we can crawl some of the customer complaints available on the Web and use that as training data for our model. This can work as a good proxy for labeled data.

### Influencer friends

One other use case of social media could be finding out the most influencer in your social graph. In our case, it could be finding out a clear node that has a vast amount of inlinks and outlinks will be the influencer in the graph.

The same problem in the context of business can be finding out the most influential customers, and targeting them to market our products.

The code for the Influencer friends is as follows:

**>>>friends = fb.get\_connections("me", "friends")["data"]**

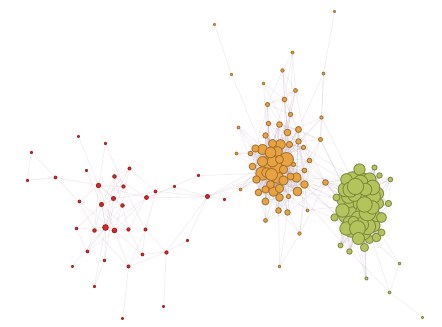
**>>>print friends >>>for frd in friends:**

**>>> print fb.get\_connections(frd["id"],"friends")**

Once you have a list of all your friends and mutual friends, you can create a data structure like this:

|  |  |  |
| --- | --- | --- |
| **source node** | **destination node** | **link\_exist** |
| Friend 1 | Friend 2 | 1 |
| Friend 1 | Friend 3 | 1 |
| Friend 2 | Friend 3 | 0 |
| Friend 1 | Friend 4 | 1 |

This a kind of data structure that can be used to generate a network, and is a very good way of visualizing the social graph. I used D3 here, but python also has a library called **NetworkX** (<https://networkx.github.io/>) that can be used to generate graph visualization, as shown in the following graph. To generate a visualization, you need to arrive at a adjacency matrix that can be created based on the bases of the preceding information about who is the friend of whom.



Visualization of a sample network in D3

## Summary

In this chapter, we touched upon some of the most popular social networks. You learned how to get data using Python. You understood the structure and kind of attributes data has. We explored different options provided by the API.

We explored some of the most common use cases in the context of social media mining. We touched upon the use cases about trending topics, influencer detection, information flow, and so on. We visualized some of these use cases. We also applied some of the learnings from the previous chapter, where we used NLTK to get some of the topic and entity extraction, while in scikit-learn we classified some of the complaints.

In conclusion, I would suggest that you look for some of the same use cases in context of some other social networks and try to explore them. The great part of these social networks is that all of them have a data API, and most of them are open enough to do some interesting analysis. If you apply the same learning you did in this chapter, you need to understand the API, how to get the data, and then how to apply some of the concepts we learned in the previous chapters. I hope that after learning all this, you will come up with more use cases, and some interesting analysis of social media.