The social web Twitter is a rich source of social data that is a great starting point for social web mining because of its inherent openness for public consumption, clean and well-documented API, rich developer tooling, and broad appeal to users from every walk of life.

Twitter data is particularly interesting because tweets happen at the "speed of thought" and are available for consumption as they happen in near real time, represent the broadest cross-section of society at an international level, and are so inherently multifaceted.

Tweets and Twitter's "following" mechanism link people in a variety of ways, ranging from short (but often meaningful) conversational dialogues to interest graphs that connect people and the things that they care about. The list of different ways to use Twitter could be really long, and with 500 million of tweets per day, there's a lot of data to analyze and to play with.

This report will describe all the steps that we developed in order to analyze streaming data from Twitter using API. This project was developed under python 3 and we use Jypyter Notebook to run our code.

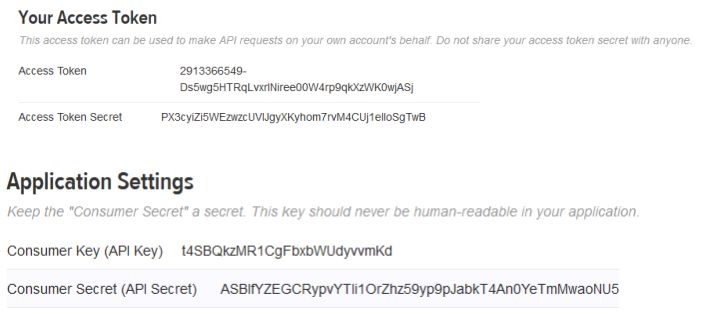
1) Twitter Stream API:

In order to have access to Twitter data programmatically, we need to create an app that interacts with the Twitter API. That's why the first step that we made in this project, we set up a twitter developer account so that you can get access to the twitter Application Program Interface (API). An API specifies how software components should interact. For our specific case, it specifies what kind of data you can collect from Twitter. We will need the API to collect large amount of tweets from Twitter automatically. Twitter API uses access token to authenticate you as a legitimate user of twitter and grant you access to the streaming tweets.

First, we sign up for a twitter account. Then, after we obtained an account, we are ready to set up a twitter developer account. Twitter provides an excellent short tutorial on this subject. The link to the tutorial is: https://twitter.com/signup?lang=ens

At the end of the tutorial you will obtain your consumer key, consumer secret, access token and access token secret, all credential needed for twitter API.

For this assignment, we need the consumer key, consumer secret, access token and access token secret. We will use these four keys to grant your analytic code access to the twitter API.



2) Collecting Data:

In order to collect data using API, python provides famous library called **tweepy** that allow us to interact with Twitter services and it is the most interesting and straightforward to use.

In order to authorize our app to access Twitter on our behalf, we need to use the OAuth interface:

```
In [1]: import tweepy
        from tweepy import OAuthHandler
        import json

        consumer_key = 't4SBQkzMR1CgFbxbWUdyvvmKd'
        consumer_secret = 'ASBlfYZEGCRypvYTli1OrZhz59yp9pJabkT4An0YeTmMwaoNU5'
        access_token = '2913366549-Ds5wg5HTRqLvxrINiree00W4rp9qkXzWK0wjASj'
        access_secret = 'PX3cyiZi5WEzwzcUVIJgyXKyhom7rvM4CUjlelloSgTwB'

        auth = OAuthHandler(consumer_key, consumer_secret)
        auth.set_access_token(access_token, access_secret)
        api = load_api()
```

Since we want to “keep the connection open”, and gather all the upcoming tweets about a particular event, the streaming API is what we need. We need to extend the StreamListener() to customise the way we process the incoming data. We can in this example how we gather all the new tweets with the **#1demayo** because we saw that is one of the trends in the moment that we wanted to collect data:

```
In [3]: import tweepy
        from tweepy import OAuthHandler
        import json
        from tweepy import Stream
        from tweepy.streaming import StreamListener

        consumer_key = 't4SBQkzMR1CgFbxbWUdyvvmKd'
        consumer_secret = 'ASBlfYZEGCRypvYTli1OrZhz59yp9pJabkT4An0YeTmMwaoNU5'
        access_token = '2913366549-Ds5wg5HTRqLvxrINiree00W4rp9qkXzWK0wjASj'
        access_secret = 'PX3cyiZi5WEzwzcUVIJgyXKyhom7rvM4CUjlelloSgTwB'

        auth = OAuthHandler(consumer_key, consumer_secret)
        auth.set_access_token(access_token, access_secret)


        class MyListener(StreamListener):

            def on_data(self, data):

                with open('trends.json', 'a') as f:
                    f.write(data)
                    return True


            def on_error(self, status):
                print(status)
                return True

        twitter_stream = Stream(auth, MyListener())
        twitter_stream.filter(track=['#1demayo'])
```

We provide in addition to this report, the example of the data that we collected in just 2 minutes using this **#1demayo** and we can see from this example how we can collect very large data in just short time (we collected more than 8000 tweets in just 2 minutes). We can confirm here the claim that we can gather tons of tweets within a few minutes, it just depends on the search term. This is especially true for live events with a world-wide coverage (World Cups, Super Bowls, Academy Awards…).

We can collect huge data using this method if we use one of the trends of that that day, however we want to see more interesting results, that’s why we decided to analyze data which contains **#Trump** and see what information can be extracted from such data. We collected 236 tweets that contains which is good amount of data to be processed. The file containing the gathered data is attached with this report.

3) Text Pre-processing:

Now we have collected a number of tweets and stored them in JSON format.

```
In [108]: import json
          with open('trump-hashtag.json', 'r') as f:
              line = f.readline() # read only the first tweet/line
              tweet = json.loads(line) # load it as Python dict
              print(json.dumps(tweet, indent=4)) # pretty-print
```


```
{
    "created_at": "Mon May 01 15:20:23 +0000 2017",
    "id": 859064968578256897,
    "id_str": "859064968578256897",
    "text": "Megyn Kelly if crazy @megynkelly didn't cover me so much on her terrible show, her ratings would totally tan
k.",
    "source": "<a href=\"http://twitter.com/download/android\" rel=\"nofollow\">Twitter for Android</a>",
    "truncated": false,
    "in_reply_to_status_id": null,
    "in_reply_to_status_id_str": null,
    "in_reply_to_user_id": null,
    "in_reply_to_user_id_str": null,
    "in_reply_to_screen_name": null,
    "user": {
        "id": 728711314437181441,
        "id_str": "728711314437181441",
        "name": "Patria",
        "screen_name": "bomberos251",
        "location": "Distrito Capital, Venezuela",
```


The key attributes of the JSON format are the following:

created_at: the date of creation

id: the tweet identifier

text: the text of the tweet itself

favorite_count, retweet_count: the number of favourites and retweets

favorited, retweeted: boolean stating whether the authenticated user (you) have favourited or retweeted this tweet

lang: acronym for the language (e.g. “en” for english)

place, coordinates, geo: geo-location information if available

user: the author’s full profile

entities: list of entities like URLs, @-mentions, hashtags and symbols

in_reply_to_user_id: user identifier if the tweet is a reply to a specific user

in_reply_to_status_id: status identifier id the tweet is a reply to a specific status

…

In the first step of our analysis we decide to break the text down into words. Tokenisation [2] is one of the most basic steps in text analysis. The purpose of tokenisation is to split a stream of text into smaller units called tokens, usually words or phrases. While this is a well understood problem with several out-of-the-

box solutions from popular libraries, Twitter data pose some challenges because of the nature of the language.

In [109]:
```
from nltk.tokenize import word_tokenize

tweet = 'RT @marcobonzanini: just an example! :D http://example.com #NLP'
print(word_tokenize(tweet))
```

```
['RT', '@', 'marcobonzanini', ':', 'just', 'an', 'example', '!', ':', 'D', 'http', ':', '//example.com', '#', 'NLP']
```

We can notice that some peculiarities that are not captured by a general-purpose English tokeniser like the one from NLTK: @-mentions, emoticons, URLs and #hash-tags are not recognised as single tokens. Now, we will make a pre-processing chain that will consider these aspects of the language.

In [5]:
```
import re

emoticons_str = r"""
    (?:
        [:=;] # Eyes
        [oO\-]? # Nose (optional)
        [D\)\]\(\]/\\OpP] # Mouth
    )"""

regex_str = [
    emoticons_str,
    r'<[^>]+>', # HTML tags
    r'(?:@[\w_]+)', # @-mentions
    r"(?:\#+[\w_]+[\w\'_\-]*[\w_]+)", # hash-tags
    r'http[s]?://(?:[a-z]|[0-9]|[$-_@.&amp;+]|[!*\(\),]|(?:%[0-9a-f][0-9a-f]))+', # URLs

    r'(?:(?:\d+,?)+(?:\.?\d+)?)', # numbers
    r"(?:[a-z][a-z'\-_]+[a-z])", # words with - and '
    r'(?:[\w_]+)', # other words
    r'(?:\S)' # anything else
]

tokens_re = re.compile(r'('+'|'.join(regex_str)+')', re.VERBOSE | re.IGNORECASE)
emoticon_re = re.compile(r'^'+emoticons_str+'$', re.VERBOSE | re.IGNORECASE)

def tokenize(s):
    return tokens_re.findall(s)

def preprocess(s, lowercase=False):
    tokens = tokenize(s)
    if lowercase:
        tokens = [token if emoticon_re.search(token) else token.lower() for token in tokens]
    return tokens

tweet = 'RT @marcobonzanini: just an example! :D http://example.com #NLP'
print(preprocess(tweet))
```

```
['RT', '@marcobonzanini', ':', 'just', 'an', 'example', '!', ':D', 'http://example.com', '#NLP']
```

We can see that now, @-mentions, emoticons, URLs and #hash-tags are now preserved as individual tokens.

4) Feature Extraction:

In order to extract the best features to understand the data and to take an idea about our data, we can perform is a simple word count [3]. In this way, we can observe what are the terms most commonly used in the data set.

We can use a custom tokeniser to split the tweets into a list of terms. The following code uses the preprocess() function described in Pre-processing part, in order to capture Twitter-specific aspects of the text, such as #hashtags, @-mentions, emoticons and URLs. In order to keep track of the frequencies while we are processing the tweets, we can use collections.Counter() which internally is a dictionary (term: count) with some useful methods like most_common():

```
import operator
import json
from collections import Counter

fname = 'trump.json'
with open(fname, 'r') as f:
    count_all = Counter()
    for line in f:
        tweet = json.loads(line)
        # Create a list with all the terms
        terms_all = [term for term in preprocess(tweet['text'])]
        # Update the counter
        count_all.update(terms_all)
    # Print the first 5 most frequent words
    print(count_all.most_common(30))
```

```
[('.', 294), (',', 133), ('the', 127), ('is', 114), ('!', 94), ('and', 87), ('a', 87), ("'", 74), ('to', 64), ('of', 57), (
'I', 52), ('Obama', 47), ('in', 46), ('The', 33), ('that', 32), ('terrible', 31), ('on', 29), ('stupid', 28), ('it', 27), (
'FAKE', 26), ('NEWS', 26), ('dumb', 25), ('so', 24), ('(', 24), (')', 24), ('not', 23), ('you', 22), ('or', 21), ('no', 20)
, ('are', 20)]
```

We can see from the results that the most frequent words are not meaningful. That's why we will try to clean the noise from these results.

In every language, some words are particularly common. While their use in the language is crucial, they don't usually convey a particular meaning, especially if taken out of context. This is the case of articles, conjunctions, some adverbs, etc. which are commonly called stop-words. Stop-word removal is one important step that should be considered during the pre-processing stages. One can build a custom list of stop-words, or use available lists (e.g. NLTK provides a simple list for English stop-words).

Given the nature of our data and our tokenisation, we should also be careful with all the punctuation marks and with terms like RT (used for re-tweets) and via (used to mention the original author of an article or a re-tweet), which are not in the default stop-word list.

```
from nltk.corpus import stopwords
import string

punctuation = list(string.punctuation)
stop = stopwords.words('english') + punctuation + ['rt', 'via']
fname = 'trump.json'
with open(fname, 'r') as f:
    count_all = Counter()
    for line in f:
        tweet = json.loads(line)
        terms_only = [term for term in preprocess(tweet['text'])
                      if term not in stop
                      and not term.startswith(('#', '@'))]
        count_all.update(terms_only)
    print(count_all.most_common(30))
```

```
[('I', 52), ('Obama', 47), ('The', 33), ('terrible', 31), ('stupid', 28), ('FAKE', 26), ('NEWS', 26), ('dumb', 25), ('peopl
e', 20), ('media', 20), ('totally', 19), ('total', 18), ('Russia', 17), ('really', 15), ('stupidity', 15), ('news', 15), ('
News', 15), ('know', 14), ('fake', 14), ('polls', 14), ('Fake', 14), ('haters', 13), ('never', 13), ('loser', 13), ('rating
s', 12), ('losers', 12), ('Just', 12), ('story', 12), ('worst', 12), ('show', 11)]
```

We remove all the noisy words and then we print the most common words and also all the words that have '#' and '@' and now we obtain more meaning words than previous ones even though they still some words that they can be removed manually like: 'I', 'The'.

We can see here that in most of the tweets that contain #Trump, there is also other words that are very frequent like 'Obama' and this is because he is the previous president of the USA and also all of people who are with trump are not with Obama's orders. Also, in most of the tweets which have relation with trump, there is a lot of words that he uses a lot like 'terrible' and 'stupid'. We find also that people who talk about trump in their tweets, they put some words for his enemies which lost and this is proven by using the words 'losers' and 'loser'.

While the frequent words represent a clear topic, more often than not simple term frequencies don't give us a deep explanation of what the text is about. To put things in context, let's consider sequences of two terms.

The bigrams() function from NLTK will take a list of tokens and produce a list of tuples using adjacent tokens.

```
: from nltk.corpus import stopwords
  import string
  from nltk import bigrams

  punctuation = list(string.punctuation)
  stop = stopwords.words('english') + punctuation + ['rt', 'via']
  fname = 'trump.json'
  with open(fname, 'r') as f:
      count_all = Counter()
      for line in f:
          tweet = json.loads(line)
          terms_stop = [term for term in preprocess(tweet['text'])
                        if term not in stop
                        and not term.startswith(('#', '@'))]
          terms_bigram = bigrams(terms_stop)
          count_all.update(terms_bigram)
      print(count_all.most_common(30))
```

```
[(('FAKE', 'NEWS'), 26), (('CNN', 'fake'), 19), (('fake', 'news'), 14), (('Fake', 'News'), 13), (('NEWS', 'media'), 11), ((
'fake', 'negative'), 9), (('negative', 'impossible'), 9), (('impossible', 'watch'), 9), (('watch', 'terrible'), 9), (('terr
ible', 'panel'), 9), (('panel', 'angry'), 9), (('angry', 'haters'), 9), (('worst', 'go'), 9), (('go', 'dumb'), 9), (('dumb'
, 'one-sided'), 9), (('one-sided', 'panels'), 9), (('panels', 'podium'), 9), (('podium', 'speaker'), 9), (('speaker', 'Trum
p'), 9), (('laughing', 'stupidity'), 8), (('fake', 'worst'), 8), (('haters', 'losers'), 8), (('I', 'would'), 7), (('ABC', '
NBC'), 7), (('really', 'dumb'), 7), (('Washington', "Post's"), 6), (("Post's", 'Ruth'), 6), (('Ruth', 'Marcus'), 6), (('Mar
cus', 'terrible'), 6), (('terrible', 'today'), 6)]
```

In this part, we have discussed interesting terms from our dataset, by using simple term frequencies, stop-word removal and biograms.

Here, we are interested in the terms that occur together. This is mainly because the context gives us a better insight about the meaning of a term, supporting applications such as word disambiguation or semantic similarity. We discussed the option of using bigrams previously, but we want to extend the context of a term to the whole tweet.

We can refactor the code in order to capture the co-occurrences. We build a co-occurrence matrix com such that com[x][y] contains the number of times the term x has been seen in the same tweet as the term y:

While building the co-occurrence matrix, we don't want to count the same term pair twice, e.g. com[A][B] == com[B][A], so the inner for loop starts from i+1 in order to build a triangular matrix, while

sorted will preserve the alphabetical order of the terms.

For each term, we then extract the 5 most frequent co-occurrent terms, creating a list of tuples in the form ((term1, term2), count):

```
from collections import defaultdict

com = defaultdict(lambda : defaultdict(int))

fname = 'trump.json'
with open(fname, 'r') as f:
    count_all = Counter()
    for line in f:
        tweet = json.loads(line)
        terms_only = [term for term in preprocess(tweet['text'])
                      if term not in stop
                      and not term.startswith(('#', '@'))]

        # Build co-occurrence matrix
        for i in range(len(terms_only)-1):
            for j in range(i+1, len(terms_only)):
                w1, w2 = sorted([terms_only[i], terms_only[j]])
                if w1 != w2:
                    com[w1][w2] += 1


com_max = []
# For each term, look for the most common co-occurrent terms
for t1 in com:
    t1_max_terms = sorted(com[t1].items(), key=operator.itemgetter(1), reverse=True)[:5]
    for t2, t2_count in t1_max_terms:
        com_max.append(((t1, t2), t2_count))
# Get the most frequent co-occurrences
terms_max = sorted(com_max, key=operator.itemgetter(1), reverse=True)
print(terms_max[:5])
```

```
[(('FAKE', 'NEWS'), 26), (('Obama', 'worst'), 24), (('CNN', 'fake'), 20), (('I', 'stupid'), 18), (('fake', 'news'), 14)]
```

Now we will look for a specific term and extract its most frequent co-occurrences. We simply need to modify the main loop including an extra counter, for example:

```
search_word = "Obama"
count_search = Counter()
fname = 'trump.json'
with open(fname, 'r') as f:
    count_all = Counter()
    for line in f:
        tweet = json.loads(line)
        terms_only = [term for term in preprocess(tweet['text'])
                      if term not in stop
                      ]
        if search_word in terms_only:
            count_search.update(terms_only)
print("Co-occurrence for %s:" % search_word)
print(count_search.most_common(20))
```

```
Co-occurrence for Obama:
[('Obama', 47), ('worst', 24), ('president', 7), ('terrible', 6), ('executive', 4), ('natural', 4), ('history', 4), ('never
', 4), ('terrorists', 4), ('country', 4), ('wants', 4), ("doesn't", 4), ('TERRIBLE', 3), ('PRESIDENT', 3), ('stupid', 3), (
'€', 3), ('™', 3), ('job', 3), ('overwhelmed', 2), ('happening', 2)]
```

We can see that the words related to ‘Obama’ have meaning and were expected. We expected to see that the followers of trump don’t like Obama’s attitude and this is obvious and we can see that by the words which are more frequent with ‘Obama’ like ‘president’ , ‘terrible’ , ‘stupid’.Also we can see that the followers of trump see a relation between Obama and terrorists by the frequent word ‘terrorists’.

```
search_word = "CNN"
count_search = Counter()
fname = 'trump.json'
with open(fname, 'r') as f:
    count_all = Counter()
    for line in f:
        tweet = json.loads(line)
        terms_only = [term for term in preprocess(tweet['text'])
                      if term not in stop
                      ]
        if search_word in terms_only:
            count_search.update(terms_only)
print("Co-occurrence for %s:" % search_word)
print(count_search.most_common(20))
```

```
Co-occurrence for CNN:
[('CNN', 23), ('fake', 20), ('negative', 11), ('impossible', 9), ('watch', 9), ('terrible', 9), ('panel', 9), ('angry', 9),
('haters', 9), ('worst', 9), ('go', 9), ('dumb', 9), ('one-sided', 9), ('panels', 9), ('podium', 9), ('speaker', 9), ('Trum
p', 9), ('polls', 8), ('election', 4), ('Just', 3)]
```

We can see that for the word ‘CNN’, based on the frequent words the followers of trump see that CNN is spreading ‘fake’ news and it is the ‘worst’ channel and also it is ‘one-sided’ channel that it wants to make influence on the ‘election’.

```
search_word = "news"
count_search = Counter()
fname = 'trump.json'
with open(fname, 'r') as f:
    count_all = Counter()
    for line in f:
        tweet = json.loads(line)
        terms_only = [term for term in preprocess(tweet['text'])
                      if term not in stop
                      ]
        if search_word in terms_only:
            count_search.update(terms_only)
print("Co-occurrence for %s:" % search_word)
print(count_search.most_common(20))
```

```
Co-occurrence for news:
[('news', 15), ('fake', 14), ('ABC', 10), ('NBC', 10), ('totally', 9), ('polls', 8), ('The', 7), ('two', 6), ('released', 6
), ('yesterday', 6), ('containing', 6), ('positive', 6), ('info', 6), ('wrong', 6), ('General', 6), ('E', 6), ('Watch', 6),
('story', 5), ('GThe', 3), ('Just', 3)]
```

The same idea can be extracted by extracting related words with 'news'.

5) Sentiment Analysis:

The technique that we are going to work with was made by Peter Turney [1]. A lot of work has been done in Sentiment Analysis [4] since then, but the approach has still an interesting educational value. In particular, it is intuitive, simple to understand and to test, and most of all unsupervised, so it doesn't require any labelled data for training.

Firstly, we define the Semantic Orientation (SO) of a word as the difference between its associations with positive and negative words. In practice, we want to calculate "how close" a word is with terms like good and bad. The chosen measure of "closeness" is Pointwise Mutual Information (PMI), calculated as follows (t1 and t2 are terms):

$$\mathrm{PMI}(t_1,t_2) = log\left(\frac{P(t1 \wedge t2)}{P(t1).P(t2)}\right)$$

In Turney's paper, the SO of a word was calculated against excellent and poor, but of course we can extend the vocabulary of positive and negative terms. Using V+ and a vocabulary of positive terms and V- for the negative ones, the Semantic Orientation of a term t is hence defined as:

$$\mathrm{SO}(t)=\sum_{t' \in V}\left(+PMI(t,t') - \sum_{t' \in V}\left(+PMI(t,t')\right)\right)$$

We can build our own list of positive and negative terms, or we can use one of the many resources available on-line.

In order to compute P(t) (the probability of observing the term t) and P(t1 ^t2) (the probability of observing the terms t1 and t2 occurring together) we can calculate term frequencies and term co-occurrences like we did previously. Given the set of documents (tweets) D, we define the Document Frequency (DF) of a term as the number of documents where the term occurs. The same definition can be applied to co-occurrent terms. Hence, we can define our probabilities as:

$$P(t) == \frac{DF(t)}{IDI}$$

$$P(t1 \wedge t2) == \frac{DF(t1 \wedge t2)}{IDI}$$

Given two vocabularies for positive and negative terms that we define

```
positive_vocab = [
    'strong','good','GOOD','Good','good','good','good','good','good', 'nice', 'great', 'awesome', 'outstanding',
    'fantastic', 'terrific', ':)', ':-)', 'like', 'love',

]
negative_vocab = [
    'hell','dead','crime','racist','DISASTER','bad','angry','fake','FAKE','loser','TERRIBLE','stupid','stupidity',
    'dumb','worst', 'terrible', 'crap', 'useless', 'hate', ':(', ':-(','weak',

]
```

We can compute the PMI of each pair of terms, and then compute the Semantic Orientation as described above. The Semantic Orientation of a term will have a positive (negative) value if the term is often associated with terms in the positive (negative) vocabulary. The value will be zero for neutral terms, e.g. the PMI's for positive and negative balance out, or more likely a term is never observed together with other terms in the positive/negative vocabularies.

```
import operator
import json
from collections import Counter
from nltk import bigrams
import math

fname = 'trump.json'
with open(fname, 'r') as f:
    count_all = Counter()
    for line in f:
        tweet = json.loads(line)

        terms_only = [term for term in preprocess(tweet['text'])
                  if term not in stop
                  ]

        count_all.update(terms_only)

n_docs=236
p_t = {}
p_t_com = defaultdict(lambda : defaultdict(int))

for term, n in count_all.items():
    p_t[term] = n / n_docs
    for t2 in com[term]:
        p_t_com[term][t2] = com[term][t2] / n_docs


positive_vocab = [
    'strong','good','GOOD','Good','good','good','good','good','good', 'nice', 'great', 'awesome', 'outstanding',
    'fantastic', 'terrific', ':)', ':-)', 'like', 'love',

]
negative_vocab = [
    'hell','dead','crime','racist','DISASTER','bad','angry','fake','FAKE','loser','TERRIBLE','stupid','stupidity',
    'dumb','worst', 'terrible', 'crap', 'useless', 'hate', ':(', ':-(','weak',

]
```

```
positive_vocab = [
    'strong','good','GOOD','Good','good','good','good','good','good', 'nice', 'great', 'awesome', 'outstanding',
    'fantastic', 'terrific', ':)', ':-)', 'like', 'love',

]
negative_vocab = [
    'hell','dead','crime','racist','DISASTER','bad','angry','fake','FAKE','loser','TERRIBLE','stupid','stupidity',
    'dumb','worst', 'terrible', 'crap', 'useless', 'hate', ':(', ':-(','weak',

]


pmi = defaultdict(lambda : defaultdict(int))
for t1 in p_t:
    for t2 in com[t1]:
        denom = p_t[t1] * p_t[t2]
        pmi[t1][t2] = math.log2(p_t_com[t1][t2] / denom)

semantic_orientation = {}
for term, n in p_t.items():
    positive_assoc = sum(pmi[term][tx] for tx in positive_vocab)
    negative_assoc = sum(pmi[term][tx] for tx in negative_vocab)
    semantic_orientation[term] = positive_assoc - negative_assoc
semantic_sorted = sorted(semantic_orientation.items(),
                         key=operator.itemgetter(1),
                         reverse=True)
top_pos = semantic_sorted[:20]
top_neg = semantic_sorted[-20:]
print(top_pos)
print(top_neg)
```

```
[('disgrace', 44.27119624193678), ('broadcasting', 44.27119624193678), ('Fox', 31.145234932383936), ('News', 24.48345683461
1957), ('dumb', 12.039270443804615), ('getting', 9.00128754586046), ('We', 7.2734330023940075), ('Emmys', 6.297680548640685
), ('especially', 5.560714954474479), ('dead', 5.560714954474479), ('along', 5.560714954474479), ('done', 4.975752453753323
), ('immediately', 4.7127180479195285), ('fathers', 4.7127180479195285), ('Fathers', 4.7127180479195285), ('enjoyable', 4.7
127180479195285), ('Memorial', 4.7127180479195285), ('extend', 4.7127180479195285), ('best', 4.7127180479195285), ('date',
4.7127180479195285)]
[('T', -7.0579269666428), ('V', -7.0579269666428), ('Cheri', -7.206865385455018), ('Jacobus', -7.206865385455018), ('begged
', -7.206865385455018), ("CNN's", -8.045114315458406), ('bankruptcy', -8.322342602874954), ('billion', -8.322342602874954),
('Obama', -8.39922614453013), ('ALWAYS', -8.595361097281371), ('BE', -8.595361097281371), ('In', -9.322342602874954), ('Bry
ant', -9.322342602874954), ('Gumbel', -9.322342602874954), ('CBS', -9.322342602874954), ('bldgs', -9.322342602874954), ('cl
ue', -9.715662909133995), ('CNN', -10.634753671987621), ('If', -11.280173148631015), ('10', -13.035060650794481)]
```

We can conclude using the sentiment analysis that as we discussed before most of followers of Donald Trump have some words that are always related to bad vocabulary like ‘Obama’ and ‘CNN’ and this validate that they have a bad attitude about Obama and CNN these days and stay talk about them.

Conclusion:

The PMI-based approach has been introduced as simple and intuitive, but of course it has some limitations and this is can extracted by the results which contain some results that have not meaning. The semantic scores are calculated on terms, meaning that there is no notion of “entity” or “concept” or “event”. For example, it would be nice to aggregate and normalise all the references to the team names, e.g. #TRUMP,Donald and Trump should all contribute to the semantic orientation of the same entity. Moreover, do the opinions on the individual teams also contribute to the overall opinion on a match?

Some aspects of natural language are also not captured by this approach, more notably modifiers and negation: how do we deal with phrases like not bad (this is the opposite of just bad) or very good (this is stronger than just good)?

This project of mining Twitter data with Python helped us to implement a simple approach for Sentiment Analysis, based on the computation of a semantic orientation score which tells us whether a term is more closely related to a positive or negative vocabulary. The intuition behind this approach is fairly simply and it can be implemented using Pointwise Mutual Information as a measure of association. The approach has of course some limitations, but it was very beneficial for us to get more familiar with such a new technique for us.

References:

1. Peter D. Turney , Thumbs Up or Thumbs Down? Semantic Orientation Applied to Unsupervised Classification of Reviews. National Research Council of Canada, 2002
2. Dr.S.Vijayarani et al, Preprocessing Techniques for Text Mining - An Overview, International Journal of Computer Science & Communication Networks
3. Helena Ahonen et al, Applying Data Mining Techniques in Text Analysis
4. Walaa Medha et al, Sentiment analysis algorithms and applications: A survey, Ain Shams Engineering Journal , 2014