# Machine Learning Engineer Nanodegree Capstone Project

## Topic: Using Twitter Data for NLP and Sentiment Analysis

**Siddhesh Khedekar**

**November 19, 2018**

---

### Project Idea

The project goal is to build a sentiment analysis model with given training dataset. In this notebook, I make the steps towards developing an algorithm that could be used as part of a mobile or web application that analyzes the sentiment of a provided text. Later it can also be applied to tweets gathered through twitter API. Once the model is built, done training on the data and analyzing the validation set, at the end of this project, my code will accept the test set and predict its accuracy. The image below displays a sample output of my finished project.

In this real-world setting, I pieced together a series of models to perform different tasks; for instance, the algorithm that detects sentiment. No perfect algorithm exists, and there are many points of possible failure. My imperfect solution nonetheless did create a very fun experience for me!

---

### Navigational Keypoints

I broke the notebook into separate steps to help in understanding the project and to navigate the notebook as noted below.


- Step 0: Import Datasets and Data Preparation
- Step 1: Data Cleaning and Saving Cleaned Data as CSV
- Step 2: Explanatory Data Analysis and Visualisation of Zipf's Law
- Step 3: Data Split and Benchmark Selection
- Step 4: Feature Investigation and Extraction
- Step 5: Model Comparison and Creating a CNN
- Step 6: Training Model and Measuring Validation Accuracy
- Step 7: Testing Model and Getting the Final Result


---

### Step 0.1: Import Datasets

For the twitter sentiment analysis model, I have chosen a dataset of tweets labeled either positive or negative. The dataset for training the model is from "Sentiment140", a dataset originated from Stanford University. More info on the dataset can be found on its official website.

The dataset can be downloaded from the provided link. First look at the description of the dataset, the following information on each field can be found.

1. sentiment of the tweet (0 = negative, 4 = positive)
2. id of the tweet (3223)
3. date of the tweet (Sun Mar 12 13:38:45 UTC 2006)
4. query_string (qwe). NO_QUERY is set as value if there is no query
5. user that tweeted (dcrules)
6. text of the tweet (Marvel is cool)

In [1]:

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
plt.style.use('fivethirtyeight')

%matplotlib inline
%config InlineBackend.figure_format = 'retina'
```

## Step 0.2: Data Preparation

There are 1.6 million entries in the Dataset, with no null entries, especially for the "sentiment" column. 50% of the data is with a negative label and another 50% with a positive label. There's no skewness on the class division. By generally looking at some entries for each class, I concluded all the negative class is from the 0~799999th index, and the positive class entries start from 800000 to the end of the dataset.

I first started by dropping the columns that I don't need for the specific purpose of sentiment analysis namely for the four below.

- "id" column is the unique ID for each tweet
- "date" column is for date and time info for the tweet
- "query_string" column indicates whether the tweet has been collected with any particular query keyword, but for this column, 100% of the entries are with value "NO_QUERY"
- "user" column is the twitter username for the one who tweeted

In [125]:

```
df = pd.read_csv("./trainingandtestdata/training_data.csv",encoding="cp1252",header=None,
                 usecols=[0,5],names=['sentiment','text'])
df['sentiment'] = df['sentiment'].map({0: 0, 4: 1})
```

## Step 1.1: Data Cleaning

Before I move on to Explanatory data analysis and visualization. I had to get the data cleaning part done right. As a way of sanity check, I took a look at the length of the string in the text column in each entry. I did a plot pre_clean_len with box plot so that I can see the overall distribution of length of strings in each entry. This looked a bit strange since twitter's character limit is 140. But from the boxplot, some of the tweets are way more than 140 characters long.

#### Part 1: HTML decoding

Looking at the data I realized that the HTML encoding had not been converted to text and ended up in the text field as '&amp', '&quot', etc. Decoding HTML to general text was my first step in data cleaning. I used BeautifulSoup for this.

#### Part 2: @Mention

I also had to deal with @mention. Although @mention carries a certain information (which another user that the tweet mentioned), this information does not add value to build sentiment analysis model. With a little googling, I found out that twitter ID also allows the underscore symbol as a character can be used with ID. Except for the underscore symbol, only characters allowed are alphabets and numbers.

#### Part 3: URL links

I had to get rid of the URL links, same as with @mention, although they carry some information, for sentiment analysis purpose, this can be ignored. I recognized is that some of the URL links don't start with "HTTP", sometimes people paste a link in "www.aaaa.com" form. I need a regex pattern to catch the part of the URL that contains alphabets, numbers, periods, slashes and also if it contains any other special character such as "=", "_", "~", etc.</n>

#### Part 4: UTF-8 Byte Order Mark(BOM)

By looking at some of the entries in the data I could see some strange patterns of characters "\xef\xbf\xbd". After some intensive research, I understood these to be UTF-8 BOM. "The UTF-8 BOM is a sequence of bytes (EF BB BF) that enables the reader to recognize a file as being encoded in UTF-8." By decoding text with 'UTF-8-sig', this BOM will be substituted with Unicode unrecognizable special characters, then I can process this as "?"

#### Part 5: Hashtag / Numbers

A hashtag can contribute valuable information about the tweet. It might be a bit dangerous to get rid of all the text together with the hashtag. So I chose to leave the text intact and just remove the '#'. I will do this in the process of refining all the non-letter characters including numbers

including numbers.

**Part 6: Defining data cleaning function**

With above five data sanitation task, I will first define data cleaning function, and then will be implemented to the whole dataset. Tokenization, stemming/lemmatization, stop words will be dealt with the later stage when creating a matrix with either count vectorizer or Tfidf vectorizer.

A major issue I realized is that, during the cleaning process, negation words are split into two parts, and the 't' after the apostrophe vanishes when I filter tokens with length more than one syllable. Because of this words like "can't" end up as same as "can". This end case needs to be handled for the purpose of sentiment analysis.

In [126]:

```
import re
from bs4 import BeautifulSoup
from nltk.tokenize import WordPunctTokenizer
import warnings
warnings.simplefilter(action='ignore', category=UserWarning)

tok = WordPunctTokenizer()
pat1 = r'@[A-Za-z0-9_]+'
pat2 = r'https?://[^ ]+'
combined_pat = r'|'.join((pat1, pat2))
www_pat = r'www.[^ ]+'
negations_dic = {"isn't":"is not", "aren't":"are not", "wasn't":"was not", "weren't":"were not",
                "haven't":"have not","hasn't":"has not","hadn't":"had not","won't":"will not",
                "wouldn't":"would not", "don't":"do not", "doesn't":"does not","didn't":"did not",
                "can't":"can not","couldn't":"could not","shouldn't":"should not","mightn't":"might
not",
                "mustn't":"must not"}
neg_pattern = re.compile(r'\b(' + '|'.join(negations_dic.keys()) + r')\b')

def tweet_cleaner(text):
    soup = BeautifulSoup(text, 'html.parser')
    souped = soup.get_text()
    try:
        bom_removed = souped.decode("utf-8-sig").replace(u"\ufffd", "?")
    except:
        bom_removed = souped
    stripped = re.sub(combined_pat, '', bom_removed)
    stripped = re.sub(www_pat, '', stripped)
    lower_case = stripped.lower()
    neg_handled = neg_pattern.sub(lambda x: negations_dic[x.group()], lower_case)
    letters_only = re.sub("[^a-zA-Z]", " ", neg_handled)
    # During the letters_only process two lines above, it has created unnecessay white spaces,
    # I will tokenize and join together to remove unneccessary white spaces
    words = [x for x  in tok.tokenize(letters_only) if len(x) > 1]
    return (" ".join(words)).strip()
```

In [3]:

```
%%time

print ("Cleaning the tweets...\n")
clean_tweet_texts = []
for i in range(0,len(df)):
    if( (i+1)%100000 == 0 ):
        print ("Tweets %d of %d has been processed" % ( i+1, len(df) ))
    clean_tweet_texts.append(tweet_cleaner(df['text'][i]))
```

```
Cleaning the tweets...

Tweets 100000 of 1600000 has been processed
Tweets 200000 of 1600000 has been processed
Tweets 300000 of 1600000 has been processed
Tweets 400000 of 1600000 has been processed
Tweets 500000 of 1600000 has been processed
Tweets 600000 of 1600000 has been processed
Tweets 700000 of 1600000 has been processed
Tweets 800000 of 1600000 has been processed
Tweets 900000 of 1600000 has been processed
Tweets 1000000 of 1600000 has been processed
Tweets 1100000 of 1600000 has been processed
Tweets 1200000 of 1600000 has been processed
```

```
Tweets 1200000 of 1600000 has been processed
Tweets 1300000 of 1600000 has been processed
Tweets 1400000 of 1600000 has been processed
Tweets 1500000 of 1600000 has been processed
Tweets 1600000 of 1600000 has been processed
Wall time: 5min 5s
```

### Step 1.2: Saving Cleaned Data as CSV

After cleaning 3,981 entries had null entries for the text column. This was strange, because the original dataset had no null entries, and if there are any null entries in the cleaned dataset, it must have happened during the cleaning process. By looking at these entries in the original data, I realized that the only text information they had was either twitter ID or it could have been URL address. Anyway, these are the info I decided to discard for the sentiment analysis, so I will drop these null rows, and update the data frame. After cleaning the tweets with the cleaner function, I took another look at the info() and head().

In [4]:

```
clean_df = pd.DataFrame(clean_tweet_texts,columns=['text'])
clean_df['target'] = df.sentiment
clean_df.to_csv('./trainingandtestdata/clean_tweet.csv',encoding='utf-8')

warnings.simplefilter(action='ignore', category=FutureWarning)
csv = './trainingandtestdata/clean_tweet.csv'
my_df = pd.read_csv(csv,index_col=0)
my_df.dropna(inplace=True)
my_df.reset_index(drop=True,inplace=True)
my_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1596041 entries, 0 to 1596040
Data columns (total 2 columns):
text      1596041 non-null object
target    1596041 non-null int64
dtypes: int64(1), object(1)
memory usage: 24.4+ MB
```

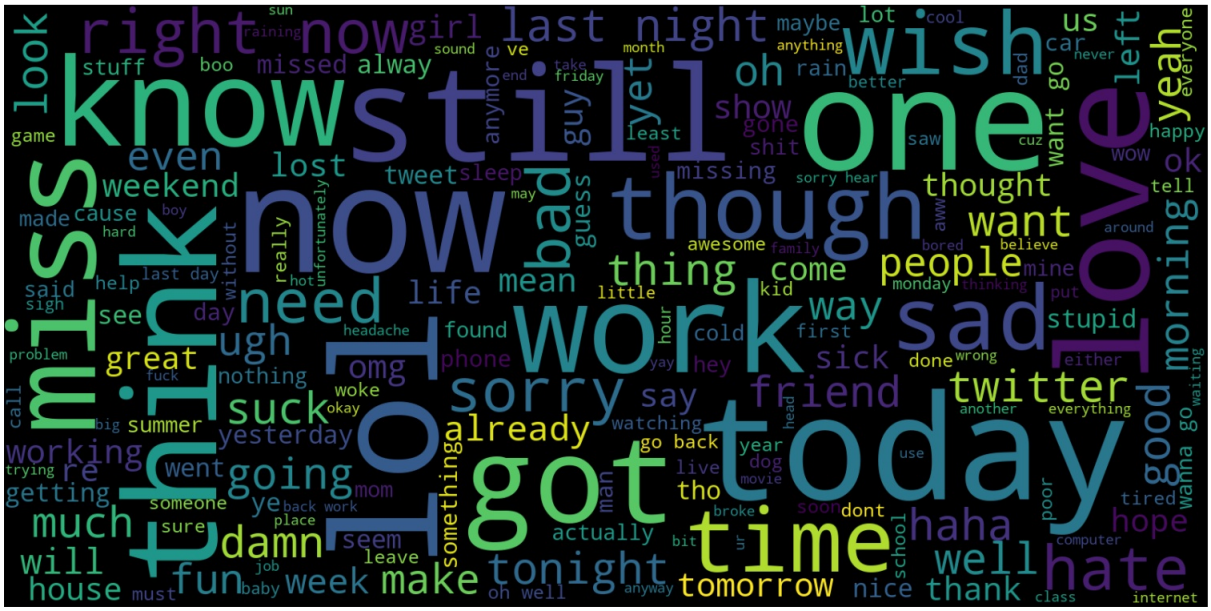
### Step 2.1: Explanatory Data Analysis using Word Cloud

A word cloud represents word usage in a document by resizing individual words proportionally to its frequency and then presenting them in a random arrangement. It is a very crude form of textual analysis, and it is often applied to situations where textual analysis is not appropriate, it does not provide a narrative and it gives no context summary of the data. But in the case of tweets, a textual analysis will do, and it provides a general idea of what kind of words are frequent in the corpus. For visualization, I used the python library word cloud

In [5]:

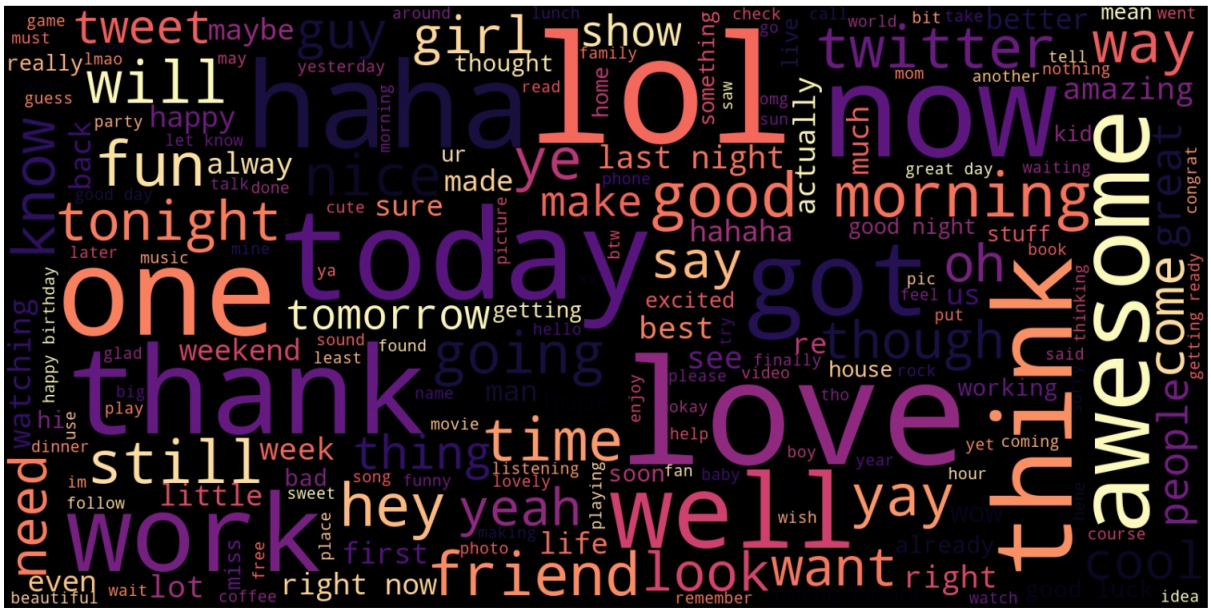
```
from wordcloud import WordCloud

neg_tweets = my_df[my_df.target == 0]
neg_string = []
for t in neg_tweets.text:
    neg_string.append(t)
neg_string = pd.Series(neg_string).str.cat(sep=' ')

pos_tweets = my_df[my_df.target == 1]
pos_string = []
for t in pos_tweets.text:
    pos_string.append(t)
pos_string = pd.Series(pos_string).str.cat(sep=' ')

wordcloud = WordCloud(width=1600, height=800,max_font_size=200).generate(neg_string)
plt.figure(figsize=(12,10))
plt.imshow(wordcloud, interpolation="bilinear")
plt.axis("off")
plt.show()
```

```
plt.show()
```




In [6]:

```
wordcloud = WordCloud(width=1600, height=800,max_font_size=200,colormap='magma').generate(pos_strin
g)
plt.figure(figsize=(12,10))
plt.imshow(wordcloud, interpolation="bilinear")
plt.axis("off")
plt.show()
```




### Step 2.2: Results from Word Cloud

Some of the big words can be interpreted quite neutral. I can see some of the words in a smaller size make sense to be in positive tweets or in the negative tweets. Interestingly, many words like "love" and "work" were quite big in the negative word cloud, but also quite big in the positive word cloud. It might imply that many people express negative sentiment towards them, but also many people are positive about them.

### Step 2.3: Preparation for Data Visualisation using Count Vectorizer

In order to implement data visualization of Zipf's Law, I need term frequency data. I selected count vectorizer to calculate the term frequencies. I implemented count vectorizer with stop words included, and not limiting the maximum number of terms. With count

vectorizer, I merely count the appearance of the words in each text. For example, let's say I have 3 documents in a corpus: "I love dogs", "I hate dogs and knitting", "Knitting is my hobby and my passion". If I build vocabulary from these three sentences and represent each document as count vectors, it will look like below picture.

In [7]:

```
from sklearn.feature_extraction.text import CountVectorizer

cvec = CountVectorizer()
cvec.fit(my_df.text)
```

Out[7]:

```
CountVectorizer(analyzer='word', binary=False, decode_error='strict',
        dtype=<class 'numpy.int64'>, encoding='utf-8', input='content',
        lowercase=True, max_df=1.0, max_features=None, min_df=1,
        ngram_range=(1, 1), preprocessor=None, stop_words=None,
        strip_accents=None, token_pattern='(?u)\\b\\w\\w+\\b',
        tokenizer=None, vocabulary=None)
```

In [8]:

```
neg_doc_matrix = cvec.transform(my_df[my_df.target == 0].text)
pos_doc_matrix = cvec.transform(my_df[my_df.target == 1].text)
neg_tf = np.sum(neg_doc_matrix,axis=0)
pos_tf = np.sum(pos_doc_matrix,axis=0)
neg = np.squeeze(np.asarray(neg_tf))
pos = np.squeeze(np.asarray(pos_tf))
term_freq_df = pd.DataFrame([neg,pos],columns=cvec.get_feature_names()).transpose()

term_freq_df.columns = ['negative', 'positive']
term_freq_df['total'] = term_freq_df['negative'] + term_freq_df['positive']
term_freq_df.sort_values(by='total', ascending=False).iloc[:10]
```

Out[8]:

| | negative | positive | total |
|---|---|---|---|
| **to** | 313162 | 252567 | 565729 |
| **the** | 257836 | 265998 | 523834 |
| **my** | 190775 | 125955 | 316730 |
| **it** | 157448 | 147786 | 305234 |
| **and** | 153958 | 149642 | 303600 |
| **you** | 103844 | 198245 | 302089 |
| **not** | 194724 | 86861 | 281585 |
| **is** | 133432 | 111191 | 244623 |
| **in** | 115542 | 101160 | 216702 |
| **for** | 98999 | 117369 | 216368 |

## Step 2.4: Zipf's Law Definition

It states that the vast majority are used very rarely, while a small number of words are used all the time. The frequency of any word is inversely proportional to its rank in the frequency table, given some corpus of natural language utterances. Thus the nth most frequent word will occur approximately 1/n times as often as the most frequent word.

The actual formula goes something like this

$$f(r) \propto \frac{1}{r^{\alpha}}$$

for

$$\alpha \approx 1$$

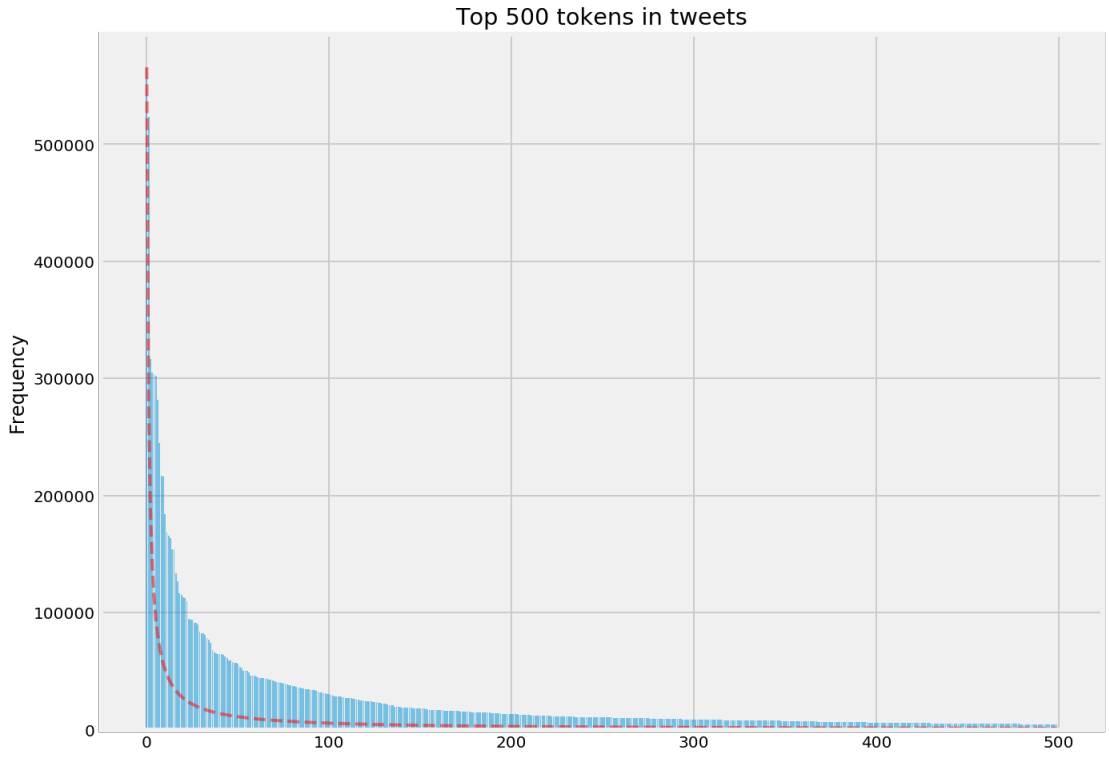
Below I plot the tweet tokens and their frequencies.

In [9]:

```
term_freq_df.to_csv('./trainingandtestdata/term_freq_df.csv',encoding='utf-8')

y_pos = np.arange(500)
plt.figure(figsize=(10,8))
s = 1
expected_zipf = [term_freq_df.sort_values(by='total', ascending=False)['total'][0]/(i+1)**s for i i
n y_pos]
plt.bar(y_pos, term_freq_df.sort_values(by='total', ascending=False)['total'][:500], align='center'
, alpha=0.5)
plt.plot(y_pos, expected_zipf, color='r', linestyle='--',linewidth=2,alpha=0.5)
plt.ylabel('Frequency')
plt.title('Top 500 tokens in tweets')
```

Out[9]:

```
Text(0.5,1,'Top 500 tokens in tweets')
```



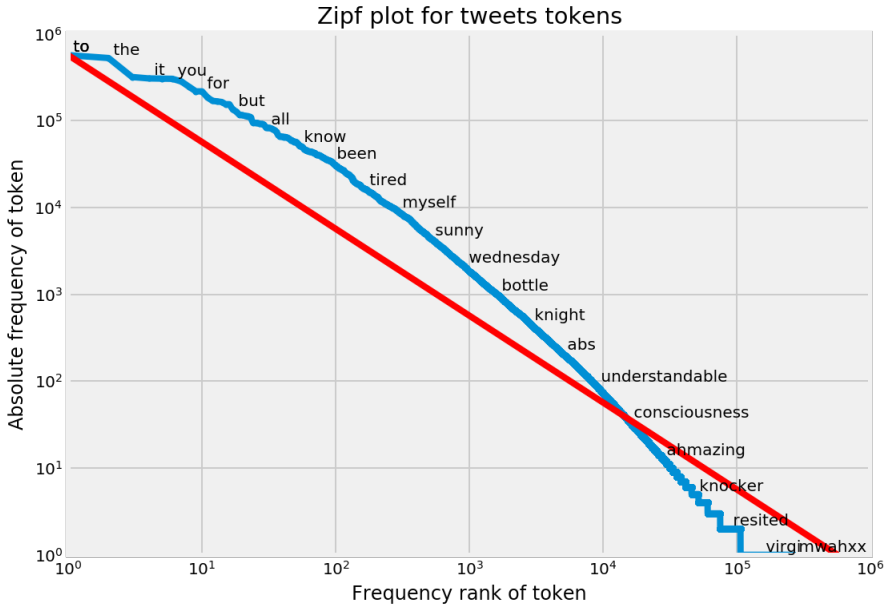

## Step 2.5: Results of Plain Plot

Y-axis is the "Sentiment140" dataset frequency. The x-axis is the rank of the frequency 500 ranks from left to the right. In most cases the actual observations do not strictly follow Zipf's distribution, but rather a near-Zipfian distribution. It looks like there is more area above the expected Zipf curve in higher ranked words. The log-log, with X-axis being log(rank), Y-axis being log(frequency) scale will yield a roughly linear line on the graph.

In [12]:

```
from pylab import *
warnings.simplefilter(action='ignore', category=UnicodeWarning)

counts = term_freq_df.total
tokens = term_freq_df.index
ranks = arange(1, len(counts)+1)
indices = argsort(-counts)
frequencies = counts[indices]
plt.figure(figsize=(8,6))
plt.ylim(1,10**6)
plt.xlim(1,10**6)
loglog(ranks, frequencies, marker=".")
plt.plot([1,frequencies[0]],[frequencies[0],1],color='r')
```

```
title("Zipf plot for tweets tokens")
xlabel("Frequency rank of token")
ylabel("Absolute frequency of token")
grid(True)
for n in list(logspace(-0.5, log10(len(counts)-2), 25).astype(int)):
    dummy = text(ranks[n], frequencies[n], " " + tokens[indices[n]], verticalalignment="bottom", ho
rizontalalignment="left")
```




### Step 2.6: Results of Log Plot

It is clearly seen again that the graph is a roughly linear curve, but at the lower ranks, we see the actual observation line lies below the expected linear line and deviating above the expected line on higher ranked words.

---

### Step 3.1: Train / Validate / Test Split

Before I train any model, I need to split the data. And considering the Here I chose to split the data into three sets: train, validation, and test. The ratio I decided to split my data is 98/1/1, 98% of the data as the training set, and 1% for the dev set, and the final 1% for the test set. The dataset has more than 1.5 million entries. This is more than enough to evaluate the model and refine the parameters.

- Train set: The portion of data to use for learning
- Validation set (Hold-out cross-validation set): The portion of data to use for fine-tuning the parameters of a classifier, and provide an unbiased evaluation of a model.
- Test set: The portion of data to use only to assess the performance of a final model.

In [13]:

```
from sklearn.model_selection  import train_test_split

x = my_df.text
y = my_df.target

SEED = 2000
x_train, x_validation_and_test, y_train, y_validation_and_test = train_test_split(x, y, test_size=.
02, random_state=SEED)
x_validation, x_test, y_validation, y_test = train_test_split(x_validation_and_test, y_validation_a
nd_test, test_size=.5, random_state=SEED)

x_train = x_train.head(16000)
x_validation = x_validation.head(2000)
x_test = x_test.head(2000)
y_train = y_train.head(16000)
y_validation = y_validation.head(2000)
y_test = y_test.head(2000)

print ("Train set has total {0} entries with {1:.2f}% negative, {2:.2f}% positive".format(
    len(x_train), (len(x_train[y_train == 0]) / (len(x_train)*1.))*100,
```

```
    (len(x_train[y_train == 1]) / (len(x_train)*1.))*100))
print ("Validation set has total {0} entries with {1:.2f}% negative, {2:.2f}% positive".format(
    len(x_validation),(len(x_validation[y_validation == 0]) / (len(x_validation)*1.))*100,
    (len(x_validation[y_validation == 1]) / (len(x_validation)*1.))*100))
print ("Test set has total {0} entries with {1:.2f}% negative, {2:.2f}% positive".format(
    len(x_test),(len(x_test[y_test == 0]) / (len(x_test)*1.))*100,
    (len(x_test[y_test == 1]) / (len(x_test)*1.))*100))
```

```
Train set has total 16000 entries with 50.59% negative, 49.41% positive
Validation set has total 2000 entries with 49.65% negative, 50.35% positive
Test set has total 2000 entries with 49.55% negative, 50.45% positive
```

### Step 3.2: Benchmark Selection 1

All the model performance will be compared to fit the same training and validation set. Baseline or benchmark provides a point of reference to compare when comparing various machine learning algorithms. Zero Rule (ZeroR) is one of the most popular baselines. It simply predicts the majority category (class). There is no predictability power in ZeroR but it is useful for determining a baseline performance as a benchmark for other classification methods.

### Step 3.3: Results of Benchmark 1

As I can see from the above validation set class division, the majority class is positive with 50.35%, which means if a classifier predicts positive for every validation data, it will get 50.35% accuracy.

### Step 3.4: Benchmark Selection 2

Another benchmark I wanted to compare the validation results with is TextBlob. This python library helps in processing textual data. Apart from other useful tools such as POS tagging, n-gram, The package has built-in sentiment classification. It is an out-of-the-box sentiment analysis tool. I will keep in mind of the accuracy I get from TextBlob sentiment analysis along with the null accuracy to see how my model is performing.

In [15]:

```
%%time
from textblob import TextBlob
from sklearn.metrics import accuracy_score
from sklearn.metrics import classification_report, confusion_matrix

tbresult = [TextBlob(i).sentiment.polarity for i in x_validation]
tbpred = [0 if n < 0 else 1 for n in tbresult]
```

```
Wall time: 1.32 s
```

In [16]:

```
conmat = np.array(confusion_matrix(y_validation, tbpred, labels=[1,0]))

confusion = pd.DataFrame(conmat, index=['positive', 'negative'],
                         columns=['predicted_positive','predicted_negative'])
print ("Accuracy Score: {0:.2f}%".format(accuracy_score(y_validation, tbpred)*100))
print ("-"*80)
print ("Confusion Matrix\n")
print (confusion)
print ("-"*80)
print ("Classification Report\n")
print (classification_report(y_validation, tbpred))
```

```
Accuracy Score: 62.75%
--------------------------------------------------------------------------------
Confusion Matrix

          predicted_positive  predicted_negative
positive                 908                  99
negative                 646                 347
--------------------------------------------------------------------------------
Classification Report

              precision    recall  f1-score   support

           0       0.78      0.35      0.48       993
```

```
              0       0.73      0.55      0.63       993
              1       0.58      0.90      0.71      1007

    micro avg       0.63      0.63      0.63      2000
    macro avg       0.68      0.63      0.60      2000
weighted avg       0.68      0.63      0.60      2000
```

### Step 3.5: Results of Benchmark 2

TextBlob sentiment analysis yielded 62.75% accuracy on the validation set, which is 12.74% more accurate than null accuracy (50.35%).

### Step 3.6: Explaining the Confusion Matrix and Classification Report

Before moving forward, let me briefly explain about confusion matrix and classification report. In order to evaluate the performance of a model, there are many different metrics that can be used. Here I will talk in case of binary classification, in which the target variable only has two classes to be predicted. In the case of this project, the classes are either "negative" or "positive". One obvious measure of performance can be accuracy. It is the number of times the model predicted correctly for the class over the number of the whole data set. But in case of classification, this can be broken down further. Below is a representation of confusion matrix.

In the above matrix, each row represents the instances in an actual class while each column represents the instances in a predicted class, and it can be also presented swapping rows and columns (column for the actual class, row for predicted class). So the accuracy (ACC) I mentioned above can be expressed as below.

$$ACC = \frac{TruePositive + TrueNegative}{Positive + Negative} = \frac{TruePositive + TrueNegative}{TruePositive + FalsePositive + TrueNegative + FalseNegative}$$

When the distribution of the classes in data is well balanced, accuracy can give you a good picture of how the model is performing. But when you have skewed data, for example, one of the class is dominant in your data set, then accuracy might not be enough to evaluate your model. Let's say you have a dataset which contains 80% positive class, and 20% negative class. This means that by predicting every data into the positive class, the model will get 80% accuracy. In this case, you might want to explore further into the confusion matrix and try different evaluation metrics. There can be 9 different metrics, just from the combination of numbers from confusion matrix, but I will talk about two of them in particular, and another metric which combines these two.

"Precision" (also called Positive Predictive Value) tells you what proportion of data predicted as positive actually is positive. In other words, the proportion of True Positive in the set of all positive predicted data.

$$PPV(Precision) = \frac{TruePositive}{TruePositive + FalsePositive}$$

"Recall" (also called Sensitivity, Hit Rate, True Positive Rate) tells you what proportion of data that actually is positive were predicted positive. In other words, the proportion of True Positive in the set of all actual positive data.

$$TPR(Recall) = \frac{TruePositive}{Positive} = \frac{TruePositive}{TruePositive + FalseNegative}$$

Below is the image of confusion matrix of cancer diagnose. If you think of "cancer" as positive class, "no cancer" as a negative class, the image explains well how to think of precision and recall in terms of the confusion matrix.

And finally, the F1 score is the harmonic mean of precision and recall. The harmonic mean is a specific type of average, which is used when dealing with averages of units, like rates and ratios. So by calculating the harmonic mean of the two metrics, it will give you a good idea of how the model is performing both in terms of precision and recall. The formula is as below

$$F1 = 2 \cdot \frac{Precision \cdot Recall}{Precision + Recall}$$

---

### Step 4.1: Feature Investigation using CountVec

If the size of the corpus gets big, the number of vocabulary gets too big to process. With my 1.5 million tweets, if I build vocabulary without limiting the number of vocabulary, I will have more than 260,000 vocabularies. This means that the shape of training data will be around 1,500,000 x 260,000, this sounds too big to train various different models with. So I decided to limit the number of vocabularies, but I also wanted to see how the performance varies depending on the number of vocabularies.

Another thing I wanted to explore is stopwords. Stop Words are words which do not contain important significance, such as "the", "of",

etc. It is often assumed that removing stopwords is a necessary step, and will improve the model performance. But I wanted to see for myself if this is really the case. So I ran the same test with and without stop words and compared the result. In addition, I also defined my custom stopwords list, which contains top 10 most frequent words in the corpus: "to", "the", "my", "it", "and", "you", "not", "is", "in", "for".

A model I chose to evaluate different count vectors is the logistic regression. It is one of the linear models, so computationally scalable to big data, compared to models like KNN or random forest. And once I have the optimal number of features and make a decision on whether to remove stop words or not, then I will try different models with the chosen number of vocabularies' count vectors.

### Step 4.2: Feature Investigation using Unigram/Bigram/Trigram

According to Wikipedia, "n-gram is a contiguous sequence of n items from a given sequence of text or speech". In other words, n-grams are simply all combinations of adjacent words or letters of length n that you can find in your source text. Below picture represents well how n-grams are constructed out of source text. In this project, I will extend the bag-of-words to trigrams, and see if it affects the performance.



### Step 4.3: Feature Investigation using Word2Vec

Before I jump into doc2vec, it will be better to first start by word2vec. "Word2vec is a group of related models that are used to produce word embeddings. These models are shallow, two-layer neural networks that are trained to reconstruct linguistic contexts of words."

Word2vec is not a single algorithm but consists of two techniques – CBOW(Continuous bag of words) and Skip-gram model. Both of these techniques learn weights which act as word vector representations. With a corpus, CBOW model predicts the current word from a window of surrounding context words, while Skip-gram model predicts surrounding context words given the current word. In Gensim package, you can specify whether to use CBOW or Skip-gram by passing the argument "sg" when implementing Word2Vec. By default (sg=0), CBOW is used. Otherwise (sg=1), skip-gram is employed.

For example, let's say we have the following sentence: "I love dogs". CBOW model tries to predict the word "love" when given "I", "dogs" as inputs, on the other hand, Skip-gram model tries to predict "I", "dogs" when given the word "love" as input. But what's used as word vectors are actually not the predicted results from these models but the weights of the trained models. By extracting the weights, such a vector comes to represent in some abstract way the ‘meaning’ of a word. Below picture represents more formally how these two models work.



### Step 4.4: Feature Investigation using Doc2Vec

Then what is doc2vec? Doc2vec uses the same logic as word2vec, but apply this to the document level. According to Mikolov et al. (2014), "every paragraph is mapped to a unique vector, represented by a column in matrix D and every word is also mapped to a unique vector, represented by a column in matrix W. The paragraph vector and word vectors are averaged or concatenated to predict the next word in a context...The paragraph token can be thought of as another word. It acts as a memory that remembers what is missing from the current context – or the topic of the paragraph." https://cs.stanford.edu/~quocle/paragraph_vector.pdf

**DM:**

This is the Doc2Vec model analogous to CBOW model in Word2vec. The paragraph vectors are obtained by training a neural network on the task of inferring a centre word based on context words and a context paragraph.

**DBOW:**

This is the Doc2Vec model analogous to Skip-gram model in Word2Vec. The paragraph vectors are obtained by training a neural network on the task of predicting a probability distribution of words in a paragraph given a randomly-sampled word from the paragraph.

I implemented Doc2Vec model using a Python library, Gensim. In case of DM model, I implemented averaging and concatenating. This is inspired by the research paper from Le and Mikolov (2014). In their paper, they have implemented DM model in two different way, using average calculation process for the paragraph matrix, and concatenating calculation method for the paragraph matrix. This has also been shown in Gensim's tutorial.

Below are the methods I used to get the vectors for each tweet.

In [34]:

```
from tqdm import tqdm
```

```
tqdm.pandas(desc="progress-bar")
import gensim
from gensim.models.word2vec import Word2Vec
from gensim.models.doc2vec import TaggedDocument
import multiprocessing
from sklearn import utils
```

### Step 5.1: Model Comparison using Features

1. DBOW (Distributed Bag of Words)
2. DMC (Distributed Memory Concatenated)
3. DMM (Distributed Memory Mean)
4. DBOW + DMC
5. DBOW + DMM

With above vectors, I fitted the model and evaluate the result on the validation set. I compared different combinations and input only the best combination to my model.

In [35]:

```
def labelize_tweets_ug(tweets,label):
    result = []
    prefix = label
    for i, t in zip(tweets.index, tweets):
        result.append(TaggedDocument(t.split(), [prefix + '_%s' % i]))
    return result
```

### Step 5.2: Preparation for Convolutional Neural Network

In order to feed to a CNN, I have to not only feed each word vector to the model, but also in a sequence which matches the original tweet.

For example, let's say I have a sentence as below.

"I love cats"

And let's assume that I have a 2-dimensional vector representation of each word as follows:

I: [0.3, 0.5] love: [1.2, 0.8] cats: [0.4, 1.3]

With the above sentence, the dimension of the vector I have for the whole sentence is 3 X 2 (3: number of words, 2: number of vector dimension).

But there is one more thing I need to consider. A neural network model will expect all the data to have the same dimension, but in case of different sentences, they will have different lengths. This can be handled with padding.

Let's say I have our second sentence as below.

"I love dogs too"

with the below vector representation of each word:

I: [0.3, 0.5], love: [1.2, 0.8], dogs: [0.8, 1.2], too: [0.1, 0.1]

The first sentence had 3X2 dimension vectors, but the second sentence has a 4X2 dimension vector. Our neural network won't accept these as inputs. By padding the inputs, I decide the maximum length of words in a sentence, then zero pads the rest, if the input length is shorter than the designated length. In the case where it exceeds the maximum length, then it will also truncate either from the beginning or from the end. For example, let's say I decide our maximum length to be 5.

Then by padding, the first sentence will have 2 more 2-dimensional vectors of all zeros at the start or the end (you can decide this by passing an argument), and the second sentence will have 1 more 2-dimensional vector of zeros at the beginning or the end. Now I have 2 same dimensional (5X2) vectors for each sentence, and I can finally feed this to a model.

In [36]:

```
all_x = pd.concat([x_train,x_validation,x_test])
all_x_w2v = labelize_tweets_ug(all_x, 'all')
```

### Step 5.3: Creating a Convolutional Neural Network

Let's say I have a sentence as follows:

"I love cats and dogs"

With word vectors (let's assume I have 200-dimensional word vectors for each word), the above sentence can be represented in 5X200 matrix, one row for each word. I remember I added zeros to pad a sentence in the above where I prepared the data to feed to an embedding layer? If our decided word length is 45, then the above sentence will have 45X200 matrix, but with all zeros in the first 40 rows. Keeping this in mind, let's take a look at how CNN works on image data.



In the above GIF, I have one filter (kernel matrix) of 3X3 dimension, convolving over the data (image matrix) and calculate the sum of element-wise multiplication result, and record the result on a feature map (output matrix). If I imagine each row of the data is for a word in a sentence, then it would not be learning efficiently since the filter is only looking at a part of a word vector at a time. The above CNN is so-called 2D Convolutional Neural Network since the filter is moving in 2-dimensional space.

What I do with text data represented in word vectors is making use of 1D Convolutional Neural Network. If a filter's column width is as same as the data column width, then it has no room to stride horizontally, and only stride vertically. For example, if our sentence is represented in 45X200 matrix, then a filter column width will also have 200 columns, and the length of row (height) will be similar to the concept of n-gram. If the filter height is 2, the filter will stride through the document computing the calculation above with all the bigrams, if the filter height is 3, it will go through all the trigrams in the document, and so on.

If a 2X200 filter is applied with stride size of 1 to 45X200 matrix, I will get 44X1 dimensional output. In the case of 1D Convolution, the output width will be just 1 in this case(number of filter=1). The output height can be easily calculated with below formula (assuming that your data is already padded).

$$OutputHeight = \frac{H - F_h}{S} + 1$$

where

H: input data height

Fh: filter height

S: stride size

In [37]:

```
cores = multiprocessing.cpu_count()
model_ug_cbow = Word2Vec(sg=0, size=100, negative=5, window=2, min_count=2, workers=cores, alpha=0.
065, min_alpha=0.065)
model_ug_cbow.build_vocab([x.words for x in tqdm(all_x_w2v)])
```

```
100%|██████████| 20000/20000 [00:00<00:00, 1000036.72it/s]
```

In [38]:

```
%%time
for epoch in range(30):
    model_ug_cbow.train(utils.shuffle([x.words for x in tqdm(all_x_w2v)]),
total_examples=len(all_x_w2v), epochs=1)
    model_ug_cbow.alpha -= 0.002
    model_ug_cbow.min_alpha = model_ug_cbow.alpha
```

```
100%|██████████| 20000/20000 [00:00<00:00, 1250109.24it/s]
100%|██████████| 20000/20000 [00:00<00:00, 1666919.96it/s]
100%|██████████| 20000/20000 [00:00<00:00, 1249960.22it/s]
100%|██████████| 20000/20000 [00:00<00:00, 999179.08it/s]
100%|██████████| 20000/20000 [00:00<00:00, 1249848.47it/s]
100%|██████████| 20000/20000 [00:00<00:00, 2241025.86it/s]
100%|██████████| 20000/20000 [00:00<00:00, 642790.43it/s]
100%|██████████| 20000/20000 [00:00<00:00, 1186490.71it/s]
100%|██████████| 20000/20000 [00:00<00:00, 1249997.47it/s]
100%|██████████| 20000/20000 [00:00<00:00, 2500404.78it/s]
100%|██████████| 20000/20000 [00:00<00:00, 1666721.24it/s]
100%|██████████| 20000/20000 [00:00<00:00, 1666754.36it/s]
100%|██████████| 20000/20000 [00:00<00:00, 2499585.22it/s]
100%|██████████| 20000/20000 [00:00<00:00, 1666456.36it/s]
```

```
100%|██████████| 20000/20000 [00:00<00:00, 1666357.04it/s]
100%|██████████| 20000/20000 [00:00<00:00, 2499063.96it/s]
100%|██████████| 20000/20000 [00:00<00:00, 1250053.35it/s]
100%|██████████| 20000/20000 [00:00<00:00, 1666688.12it/s]
100%|██████████| 20000/20000 [00:00<00:00, 1872122.84it/s]
100%|██████████| 20000/20000 [00:00<00:00, 1135975.08it/s]
100%|██████████| 20000/20000 [00:00<00:00, 1196543.57it/s]
100%|██████████| 20000/20000 [00:00<00:00, 1458355.73it/s]
100%|██████████| 20000/20000 [00:00<00:00, 1666456.36it/s]
100%|██████████| 20000/20000 [00:00<00:00, 2500032.19it/s]
100%|██████████| 20000/20000 [00:00<00:00, 2500032.19it/s]
100%|██████████| 20000/20000 [00:00<00:00, 1666754.36it/s]
100%|██████████| 20000/20000 [00:00<00:00, 2499957.68it/s]
100%|██████████| 20000/20000 [00:00<00:00, 1666522.57it/s]
100%|██████████| 20000/20000 [00:00<00:00, 1666456.36it/s]
100%|██████████| 20000/20000 [00:00<00:00, 1249811.23it/s]
```

```
Wall time: 6.67 s
```

In [39]:

```
model_ug_sg = Word2Vec(sg=1, size=100, negative=5, window=2, min_count=2, workers=cores, alpha=0.06
5, min_alpha=0.065)
model_ug_sg.build_vocab([x.words for x in tqdm(all_x_w2v)])
```

```
100%|██████████| 20000/20000 [00:00<00:00, 1250016.09it/s]
```

In [40]:

```
%%time
for epoch in range(30):
    model_ug_sg.train(utils.shuffle([x.words for x in tqdm(all_x_w2v)]),
total_examples=len(all_x_w2v), epochs=1)
    model_ug_sg.alpha -= 0.002
    model_ug_sg.min_alpha = model_ug_sg.alpha
```

```
100%|██████████| 20000/20000 [00:00<00:00, 1000072.48it/s]
100%|██████████| 20000/20000 [00:00<00:00, 1249885.72it/s]
100%|██████████| 20000/20000 [00:00<00:00, 1666721.24it/s]
100%|██████████| 20000/20000 [00:00<00:00, 1279843.77it/s]
100%|██████████| 20000/20000 [00:00<00:00, 906709.90it/s]
100%|██████████| 20000/20000 [00:00<00:00, 1249960.22it/s]
100%|██████████| 20000/20000 [00:00<00:00, 1249494.76it/s]
100%|██████████| 20000/20000 [00:00<00:00, 1293360.67it/s]
100%|██████████| 20000/20000 [00:00<00:00, 2499138.41it/s]
100%|██████████| 20000/20000 [00:00<00:00, 1666688.12it/s]
100%|██████████| 20000/20000 [00:00<00:00, 1665265.41it/s]
100%|██████████| 20000/20000 [00:00<00:00, 1285354.34it/s]
100%|██████████| 20000/20000 [00:00<00:00, 959059.76it/s]
100%|██████████| 20000/20000 [00:00<00:00, 1025765.54it/s]
100%|██████████| 20000/20000 [00:00<00:00, 2501224.88it/s]
100%|██████████| 20000/20000 [00:00<00:00, 1666655.01it/s]
100%|██████████| 20000/20000 [00:00<00:00, 2499808.68it/s]
100%|██████████| 20000/20000 [00:00<00:00, 2499659.70it/s]
100%|██████████| 20000/20000 [00:00<00:00, 2498989.51it/s]
100%|██████████| 20000/20000 [00:00<00:00, 1666555.68it/s]
100%|██████████| 20000/20000 [00:00<00:00, 955531.15it/s]
100%|██████████| 20000/20000 [00:00<00:00, 2499585.22it/s]
100%|██████████| 20000/20000 [00:00<00:00, 971837.30it/s]
100%|██████████| 20000/20000 [00:00<00:00, 1161824.88it/s]
100%|██████████| 20000/20000 [00:00<00:00, 2499212.87it/s]
100%|██████████| 20000/20000 [00:00<00:00, 2499957.68it/s]
100%|██████████| 20000/20000 [00:00<00:00, 2499510.74it/s]
100%|██████████| 20000/20000 [00:00<00:00, 2502568.02it/s]
100%|██████████| 20000/20000 [00:00<00:00, 2499138.41it/s]
100%|██████████| 20000/20000 [00:00<00:00, 2500032.19it/s]
```

```
Wall time: 9.82 s
```

Let's first load the Word2Vec models to extract word vectors from. I have saved the Word2Vec models I trained in the previous post, and can easily be loaded with "KeyedVectors" function in Gensim. I have two different Word2Vec models, one with CBOW

(Continuous Bag Of Words) model, and the other with a skip-gram model. I won't go into detail about how CBOW and skip-gram differs, but you can refer to my previous post if you want to know a bit more in detail.

I am constructing a sort of dictionary I can extract the word vectors from. Since I have two different Word2Vec models, below "embedding_index" will have concatenated vectors of the two models. For each model, I have 100 dimension vector representation of the words, and by concatenating each word will have 200 dimension vector representation.

Now I have our reference to word vectors ready, but I still haven't prepared data to be in the format I have explained at the start of the post. Keras' 'Tokenizer' will split each word in a sentence, then I can call 'texts_to_sequences' method to get the sequential representation of each sentence. I also need to pass 'num_words' which is a number of vocabularies you want to use, and this will be applied when you call 'texts_to_sequences' method. This might be a bit counter-intuitive since if you check the length of all the word index, it will not be the number of words you defined, but the actual screening process happens when you call 'texts_to_sequences' method.

Each word is represented as a number, and I can see that the number of words in each sentence is matching the length of numbers in the "sequences". I can later make connections of which word each number represents. But I still didn't pad our data, so each sentence has varying length. Let's deal with this.

The maximum number of words in a sentence within the training data is 40. Let's decide the maximum length to be a bit longer than this, let's say 45.

As you can see from the padded sequences, all the data now transformed to have the same length of 45, and by default, Keras zero-pads at the beginning, if a sentence length is shorter than the maximum length. If you want to know more in detail, please check the Keras documentation on sequence preprocessing. https://keras.io/preprocessing/sequence/

There's still one more thing left to do before I can feed the sequential text data to a model. When I transformed a sentence into a sequence, each word is represented by an integer number. Actually, these numbers are where each word is stored in the tokenizer's word index. Keeping this in mind, let's build a matrix of these word vectors, but this time I will use the word index number so that our model can refer to the corresponding vector when fed with integer sequence.

As a sanity check, if the embedding matrix has been generated properly. In the above, when I saw the first five entries of the training set, the first entry was "hate you", and the sequential representation of this was [137, 6]. Let's see if 6th embedding matrix is as same as vectors for the word 'you'.

Now I am ready with the data preparation. Before I jump into CNN, I would like to test one more thing (sorry for the delay). When I feed this sequential vector representation of data, I will use the Embedding layer in Keras. With Embedding layer, I can either pass pre-defined embedding, which I prepared as 'embedding_matrix' above, or Embedding layer itself can learn word embeddings as the whole model trains. And another possibility is I can still feed the pre-defined embedding but make it trainable so that it will update the values of vectors as the model trains.

Feeding pre-trained word vectors for an embedding layer to update is like providing the first initialization guideline to the embedding layer so that it can learn more efficiently the task-specific word vectors. But the result is somewhat counterintuitive, and in this case, it turns out that it is better to force the embedding layer to learn from scratch.

In [41]:

```
model_ug_cbow.save('./model/w2v_model_ug_cbow.word2vec')
model_ug_sg.save('./model/w2v_model_ug_sg.word2vec')
```

In [43]:

```
from gensim.models import KeyedVectors
model_ug_cbow = KeyedVectors.load('./model/w2v_model_ug_cbow.word2vec')
model_ug_sg = KeyedVectors.load('./model/w2v_model_ug_sg.word2vec')
```

In [44]:

```
len(model_ug_cbow.wv.vocab.keys())
```

Out[44]:

```
8265
```

In [46]:

```
from keras.preprocessing.text import Tokenizer
from keras.preprocessing.sequence import pad_sequences

tokenizer = Tokenizer(num_words=100000)
tokenizer.fit_on_texts(x_train)
```

```
sequences = tokenizer.texts_to_sequences(x_train)
```

In [47]:

```
len(tokenizer.word_index)
```

Out[47]:

```
17332
```

In [49]:

```
for x in x_train[:5]:
    print (x)
```

```
your not pregnant oh no what shame
cleaning the bathroom
feeling left out you never recommend anything to me
home sick what the hell wonder if it ll mutate into swine flu
your tweet reminded me that game was the shit
```

In [50]:

```
sequences[:5]
```

Out[50]:

```
[[41, 7, 2323, 81, 35, 46, 715],
 [626, 2, 2324],
 [164, 213, 30, 5, 156, 1700, 340, 1, 12],
 [79, 184, 46, 2, 416, 663, 73, 6, 65, 7150, 214, 756, 541],
 [41, 206, 2672, 12, 14, 251, 22, 2, 358]]
```

In [51]:

```
length = []
for x in x_train:
    length.append(len(x.split()))
```

In [52]:

```
max(length)
```

Out[52]:

```
31
```

In [53]:

```
x_train_seq = pad_sequences(sequences, maxlen=45)
print('Shape of data tensor:', x_train_seq.shape)
```

```
Shape of data tensor: (16000, 45)
```

In [54]:

```
x_train_seq[:5]
```

Out[54]:

```
array([[   0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,
           0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,
           0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,
           0,    0,    0,    0,    0,   41,    7, 2323,   81,   35,   46,
         715],
       [   0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,
           0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,
           0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,
```

```
          0,    0,    0,    0,    0,    0,    0,    0,    0,  626,    2,
       2324],
       [   0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,
           0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,
           0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,
           0,    0,    0,  164,  213,   30,    5,  156, 1700,  340,    1,
          12],
       [   0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,
           0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,
           0,    0,    0,    0,    0,    0,    0,    0,    0,    0,   79,
         184,   46,    2,  416,  663,   73,    6,   65, 7150,  214,  756,
         541],
       [   0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,
           0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,
           0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,
           0,    0,    0,   41,  206, 2672,   12,   14,  251,   22,    2,
         358]])
```

In [55]:

```
sequences_val = tokenizer.texts_to_sequences(x_validation)
x_val_seq = pad_sequences(sequences_val, maxlen=45)
```

In [56]:

```
num_words = 100000
embedding_matrix = np.zeros((num_words, 200))
for word, i in tokenizer.word_index.items():
    if i >= num_words:
        continue
    embedding_vector = embeddings_index.get(word)
    if embedding_vector is not None:
        embedding_matrix[i] = embedding_vector
```

In [57]:

```
np.array_equal(embedding_matrix[6] ,embeddings_index.get('you'))
```

Out[57]:

```
False
```

In [58]:

```
from keras.models import Sequential, Model
from keras.callbacks import ModelCheckpoint
from keras.layers.embeddings import Embedding
from keras.layers import Dense, Dropout, Flatten, Input, concatenate, Activation, Conv1D, GlobalMax
Pooling1D
```

### Step 5.4: Fine Tuning the Convolutional Neural Network



Basically, the above structure is implementing what I have done with bigram filters, but not only to bigrams but also to trigrams and fourgrams. However this is not linearly stacked layers, but parallel layers. And after convolutional layer and max-pooling layer, it simply concatenated max pooled result from each of bigram, trigram, and fourgram, then build one output layer on top of them.

The model I defined is basically as same as the above picture, but the differences are that I added one fully connected hidden layer with dropout just before the output layer, and also my output layer will have just one output node with Sigmoid activation instead of two.

So far I have only used Sequential model API of Keras, and this worked fine with all the previous models I defined above since the structures of the models were only linearly stacked. But as you can see from the above picture, the model I am about to define has parallel layers which take the same input but do their own computation, then the results will be merged. In this kind of neural network structure, we can use Keras functional API.

Keras functional API can handle multi-input, multi-output, shared layers, shared input, etc. It is not impossible to define these types of models with Sequential API, but when you want to save the trained model, functional API enables you to simply save the model and load, but with sequential API it is difficult.

load, but with sequential API it is difficult.

In [59]:

```
tweet_input = Input(shape=(45,), dtype='int32')

tweet_encoder = Embedding(100000, 200, weights=[embedding_matrix], input_length=45, trainable=True)
(tweet_input)
bigram_branch = Conv1D(filters=100, kernel_size=2, padding='valid', activation='relu', strides=1)(t
weet_encoder)
bigram_branch = GlobalMaxPooling1D()(bigram_branch)
trigram_branch = Conv1D(filters=100, kernel_size=3, padding='valid', activation='relu', strides=1)(
tweet_encoder)
trigram_branch = GlobalMaxPooling1D()(trigram_branch)
fourgram_branch = Conv1D(filters=100, kernel_size=4, padding='valid', activation='relu', strides=1)
(tweet_encoder)
fourgram_branch = GlobalMaxPooling1D()(fourgram_branch)

merged = concatenate([bigram_branch, trigram_branch, fourgram_branch], axis=1)
merged = Dense(256, activation='relu')(merged)
merged = Dropout(0.2)(merged)
merged = Dense(1)(merged)
output = Activation('sigmoid')(merged)
model = Model(inputs=[tweet_input], outputs=[output])
model.compile(loss='binary_crossentropy',
                  optimizer='adam',
                  metrics=['accuracy'])
model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         (None, 45)                0
_________________________________________________________________
embedding_1 (Embedding)      (None, 45, 200)           20000000
_________________________________________________________________
conv1d_1 (Conv1D)            (None, 44, 100)           40100
_________________________________________________________________
conv1d_2 (Conv1D)            (None, 43, 100)           60100
_________________________________________________________________
conv1d_3 (Conv1D)            (None, 42, 100)           80100
_________________________________________________________________
global_max_pooling1d_1 (GlobalM (None, 100)            0
_________________________________________________________________
global_max_pooling1d_2 (GlobalM (None, 100)            0
_________________________________________________________________
global_max_pooling1d_3 (GlobalM (None, 100)            0
_________________________________________________________________
concatenate_1 (Concatenate)  (None, 300)               0
                                                       global_max_pooling1d_1[0][0]
                                                       global_max_pooling1d_2[0][0]
                                                       global_max_pooling1d_3[0][0]
_________________________________________________________________
dense_1 (Dense)              (None, 256)               77056
_________________________________________________________________
dropout_1 (Dropout)          (None, 256)               0
_________________________________________________________________
dense_2 (Dense)              (None, 1)                 257
_________________________________________________________________
activation_1 (Activation)    (None, 1)                 0
=================================================================
Total params: 20,257,613
Trainable params: 20,257,613
Non-trainable params: 0
_________________________________________________________________
```

## Step 6.1: Training Model

I use the model I defined on the training data and I save the weights every time they improve.

In [61]:

```
filepath="./model/CNN_best_weights.{epoch:02d}-{val_acc:.4f}.hdf5"
checkpoint = ModelCheckpoint(filepath, monitor='val_acc', verbose=1, save_best_only=True, mode='max
')
```

```
')

model.fit(x_train_seq, y_train, batch_size=32, epochs=5,
                    validation_data=(x_val_seq, y_validation), callbacks = [checkpoint])
```

```
Train on 16000 samples, validate on 2000 samples
Epoch 1/5
16000/16000 [==============================] - 940s 59ms/step - loss: 0.5621 - acc: 0.7061 - val_l
oss: 0.4982 - val_acc: 0.7685

Epoch 00001: val_acc improved from -inf to 0.76850, saving model to ./model/CNN_best_weights.01-0.
7685.hdf5
Epoch 2/5
16000/16000 [==============================] - 950s 59ms/step - loss: 0.4293 - acc: 0.8031 - val_l
oss: 0.4794 - val_acc: 0.7740

Epoch 00002: val_acc improved from 0.76850 to 0.77400, saving model to
./model/CNN_best_weights.02-0.7740.hdf5
Epoch 3/5
16000/16000 [==============================] - 897s 56ms/step - loss: 0.2903 - acc: 0.8803 - val_l
oss: 0.5759 - val_acc: 0.7575

Epoch 00003: val_acc did not improve from 0.77400
Epoch 4/5
16000/16000 [==============================] - 896s 56ms/step - loss: 0.1369 - acc: 0.9483 - val_l
oss: 0.7922 - val_acc: 0.7385

Epoch 00004: val_acc did not improve from 0.77400
Epoch 5/5
16000/16000 [==============================] - 891s 56ms/step - loss: 0.0802 - acc: 0.9701 - val_l
oss: 1.0242 - val_acc: 0.7450

Epoch 00005: val_acc did not improve from 0.77400
```

Out[61]:

```
<keras.callbacks.History at 0x226923a3908>
```

### Step 6.2: Measuring Validation Accuracy

I load the best weights the model created from the training process and test it on the validation data. I evaluate and summerize the loss and accuracy

In [62]:

```
from keras.models import load_model

loaded_CNN_model = load_model('./model/CNN_best_weights.02-0.7740.hdf5')
loaded_CNN_model.evaluate(x=x_val_seq, y=y_validation)
```

```
2000/2000 [==============================] - 1s 541us/step
```

Out[62]:

```
[0.479428031206131, 0.774]
```

### Step 6.3: Result Summary

The best validation accuracy is 77.40%, slightly better than all the other ones I tried before. I could even define a deeper structure with more hidden layers, or even make use of the multi-channel approach, or try different pool size to see how the performance differs, but I will stop here for now.

---

### Step 7.1: Testing Model Evaluation with Test Set

So far I have tested the model on the validation set to decide the feature extraction tuning and model comparison. Now I will finally check the final result with the test set. I will plot the ROC curve of the model Word2Vec + CNN.

In [82]:

```
sequences_test = tokenizer.texts_to_sequences(x_test)
x_test_seq = pad_sequences(sequences_test, maxlen=45)
loaded_CNN_model.evaluate(x=x_test_seq, y=y_test)
```

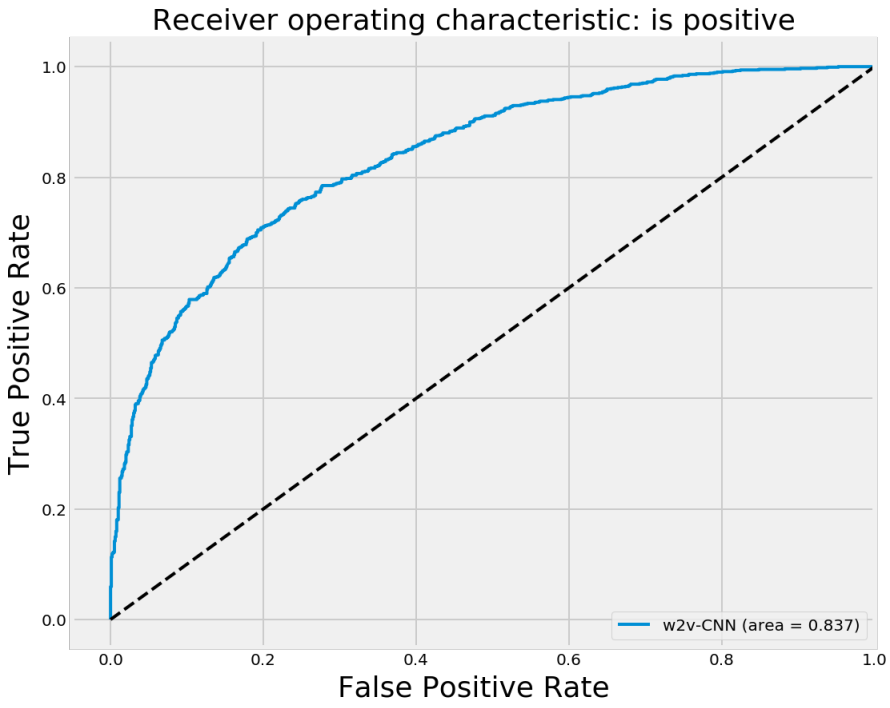
```
2000/2000 [==============================] - 2s 1ms/step
```

Out[82]:

```
[0.49982174110412597, 0.7545]
```

In [83]:

```
from sklearn.metrics import roc_curve, auc

yhat_cnn = loaded_CNN_model.predict(x_test_seq)

fpr_cnn, tpr_cnn, threshold = roc_curve(y_test, yhat_cnn)
roc_auc_nn = auc(fpr_cnn, tpr_cnn)
plt.figure(figsize=(8,7))
plt.plot(fpr_cnn, tpr_cnn, label='w2v-CNN (area = %0.3f)' % roc_auc_nn, linewidth=2)
plt.plot([0, 1], [0, 1], 'k--', linewidth=2)
plt.xlim([-0.05, 1.0])
plt.ylim([-0.05, 1.05])
plt.xlabel('False Positive Rate', fontsize=18)
plt.ylabel('True Positive Rate', fontsize=18)
plt.title('Receiver operating characteristic: is positive', fontsize=18)
plt.legend(loc="lower right")
plt.show()
```




### Step 7.2: Result Summary

| Model | Validation Set Accuracy | Test Set Accuracy | ROC AUC |
|---|---|---|---|
| Word2Vec + CNN | 77.74% | 75.45% | 0.84 |