Natural Language Processing: Lab 2 - Building and Visualizing word frequencies

In this lab, our goal of tweet sentiment analysis, is to build a dictionary where we can lookup how many times a word appears in the lists of **positive** or **negative tweets**. This will be very helpful when extracting the features of the dataset.

Necessary Libraries and Functions

Let us first import the required libraries:

In [9]:
```
import re                                   # Library for Regular Expression (RegEx) operations
import string                               # Library for String operations
import random                               # Python library to generate a pseudo - random integer
import matplotlib.pyplot as plt             # Visualization library
import numpy as np                          # Library for scientific computing and matrix operations

import nltk                                 # Python library for NLP
from nltk.corpus import twitter_samples     # Sample Twitter dataset from NLTK
from nltk.corpus import stopwords           # Module for stop - words that come with NLTK
from nltk.stem import PorterStemmer         # Module for Stemming
from nltk.tokenize import TweetTokenizer    # Module for Tokenizing strings
```

In [5]:
```
# download the samples from NLTK
nltk.download('twitter_samples')

# download the stopwords from NLTK
nltk.download('stopwords')
```

```
[nltk_data] Downloading package twitter_samples to
[nltk_data]     /home/julian/nltk_data...
[nltk_data]   Package twitter_samples is already up-to-date!
[nltk_data] Downloading package stopwords to /home/julian/nltk_data...
[nltk_data]   Package stopwords is already up-to-date!
```

Out[5]:
```
True
```

We are going to define some functions, whose functionality was evident in our previous Lab 1, and will be helpful during this lab.

Load data

The first operation that we will perform, will be the initialization of all the positive as well negative sentiment tweets, from the already downloaded twitter_samples file.

In [6]:
```
# Select the set of positive and negative tweets
all_positive_tweets = twitter_samples.strings('positive_tweets.json')
all_negative_tweets = twitter_samples.strings('negative_tweets.json')

# Import the english stop words list from NLTK
stopwords_english = stopwords.words('english')
```

preProcess_tweet()

The function that we are going to build: preProcess_tweet(), will be responsible for cleaning the text, tokenizes it into separate words, removes stopwords, and converts words to stems. The function will take as input parameter a tweet - data and will return a list of preprocessed tokens.

In [8]:
```
def preProcess_tweet(tweet):
    # Part 1: Clean redundancy
    # ------------------------
    # Remove re-tweets
    data = re.sub(r'^RT[\s]+', '', tweet)

    # Remove hyperlinks
    data = re.sub(r'https?://[^\s\n\r]+', '', data)

    # Remove hashtags by only removing the hash # sign from the word
    data = re.sub(r'#', '', data)

    # Part 2: Tokenize
    # ------------------------
    # Instantiate tokenizer class
    tokenizer = TweetTokenizer(preserve_case=False, strip_handles=True, reduce_len=True)

    # tokenize tweets
    data_tokens = tokenizer.tokenize(data)
```

```
    data_clean = []
    for word in data_tokens: # Go through every word in your tokens list
        if (word not in stopwords_english and  # remove stopwords
            word not in string.punctuation):  # remove punctuation
            data_clean.append(word)

    # Part 3: Stem
    # ------------------------
    # Instantiate stemming class
    stemmer = PorterStemmer()

    # Create an empty list to store the stems
    data_stem = []

    for word in data_clean:
        stem_word = stemmer.stem(word)  # stemming word
        data_stem.append(stem_word)  # append to the list

    return data_stem
```

Let us now test whether or not our implemented function runs in a proper desired way:

In [10]:
```
tweet1 = all_positive_tweets[random.randint(0,4999)]
print('\033[92m'+tweet1)
```

```
@jasmineshaddock Oooh how lovely! Hope you have a fantastic time :)
```

In [16]:
```
tweet2 = all_negative_tweets[random.randint(0,4999)]
print('\033[91m'+tweet2)
```

```
@kristynlopez97 you didn't get your vanilla latte though :(
```

In [17]:
```
preprocessed_tweet1 = preProcess_tweet(tweet1)
print('\033[92m'+tweet1)
print('\033[94m'+str(preprocessed_tweet1))
```

```
@jasmineshaddock Oooh how lovely! Hope you have a fantastic time :)
['oooh', 'love', 'hope', 'fantast', 'time', ':)']
```

In [18]:
```
preprocessed_tweet2 = preProcess_tweet(tweet2)
print('\033[91m'+tweet2)
print('\033[94m'+str(preprocessed_tweet2))
```

```
@kristynlopez97 you didn't get your vanilla latte though :(
['get', 'vanilla', 'latt', 'though', ':(']
```

Therefore we can see that our function is performing as expected.

Dictionaries

In Python, a dictionary is a mutable and indexed collection. It stores items as key-value pairs and uses hash tables underneath to allow practically constant time lookups. In NLP, dictionaries are **essential**, because it enables fast retrieval of items or containment checks even with thousands of entries in the collection.

A dictionary in Python is declared using curly brackets. Look at the next example:

In [31]:
```
dictionary = {'key1': 1, 'key2': 2}
```

The former line defines a dictionary with two entries. Keys and values can be almost any type (with a few restriction on keys), and in this case, we used strings. We can also use floats, integers, tuples, etc.

New entries can be inserted into dictionaries using square brackets. If the dictionary already contains the specified key, its value is overwritten.

In [33]:
```
# Add a new entry
dictionary['key3'] = -5

# Overwrite the value of key1
dictionary['key1'] = 0

print(dictionary)
```

```
{'key1': 0, 'key2': 2, 'key3': -5}
```

Performing dictionary lookups and retrieval are common tasks in NLP. There are two ways to do this:

- Using square bracket notation: This form is allowed if the lookup key is in the dictionary. It produces an error otherwise.
- Using the get method: This allows us to set a default value if the dictionary key does not exist.

Let us see these in action:

In [34]:
```
# Square bracket lookup when the key exist
```

```
print(dictionary['key2'])
```

```
2
```

However, if the key is missing, the operation produce an error

In [35]:
```
# The output of this line is intended to produce a KeyError
print(dictionary['key8'])
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
Input In [35], in <cell line: 2>()
      1 # The output of this line is intended to produce a KeyError
----> 2 print(dictionary['key8'])

KeyError: 'key8'
```

When using a square bracket lookup, it is common to use an if-else block to check for containment first (with the keyword in) before getting the item. On the other hand, you can use the .get() method if you want to set a default value when the key is not found. Let's compare these in the cells below:

In [36]:
```
# This prints a value
if 'key1' in dictionary:
    print("item found: ", dictionary['key1'])
else:
    print('key1 is not defined')
```

```
item found:  0
```

In [37]:
```
# Same as what you get with get
print("item found: ", dictionary.get('key1', -1))
```

```
item found:  0
```

In [38]:
```
# This prints a message because the key is not found
if 'key7' in dictionary:
    print(dictionary['key7'])
else:
    print('key does not exist!')
```

```
key does not exist!
```

In [39]:
```
# This prints -1 because the key is not found and we set the default to -1
```

```
print(dictionary.get('key7', -1))
```

```
-1
```

build_freqs()

Another very important function, is the build_freqs() function. A documentation of the code, can be viewed here.

As we will see by the code below, the function counts how often a word in the 'corpus' (the entire set of tweets) was associated with a positive label 1 or a negative label 0. It then builds the freqs dictionary, where each key is a (word,label) tuple, and the value is the count of its frequency within the corpus of tweets. Let us examine, how can we create the prementioned example as well as where the logic behind it stands.

In [26]:
```
# Concatenate the lists, 1st part is the positive tweets followed by the negative
tweets = all_positive_tweets + all_negative_tweets

# Let's see how many tweets we have
print("Number of tweets: ", len(tweets))
```

```
Number of tweets:  10000
```

Next, we will build an array of labels that matches the sentiments of our tweets. This data type works pretty much like a regular list but is optimized for computations and manipulation. The labels array will be composed of 10000 elements. The first 5000 will be filled with 1 labels denoting positive sentiments, and the next 5000 will be 0 labels denoting the opposite. We can do this easily with a series of operations provided by the numpy library:

- np.ones(#nrItems) - create an 1 x #nrItems array of 1's
- np.zeros(#nrItems) - create an 1 x #nrItems array of 0's
- np.append(#array1, #array2) - concatenate arrays 1 and 2

In [28]:
```
# Make a numpy array representing labels of the tweets
labels = np.append(np.ones((len(all_positive_tweets))), np.zeros((len(all_negative_tweets))))
```

In [48]:
```
# Convert np array to list since zip needs an iterable.
# The squeeze is necessary or the list ends up with one element.
# Also note that this is just a NOP if ys is already a list.
# .squeeze() documentation can be found in: https://numpy.org/doc/stable/reference/generated/numpy.squeeze.html

y_labelslist = np.squeeze(labels).tolist()
```

```
print(str(labels[0:3]))
print(str(y_labelslist[0:3]))

print('\n')
print(type(labels))
print(type(y_labelslist))

print('\n')
print(type(labels[0]))
print(type(y_labelslist[0]))
```

```
[1. 1. 1.]
[1.0, 1.0, 1.0]


<class 'numpy.ndarray'>
<class 'list'>


<class 'numpy.float64'>
<class 'float'>
```

In [54]:
```
# Start with an empty dictionary and populate it by looping over all tweets
# as well as over all processed words in each tweet.

freqs = {}

for y_label, tweet in zip(y_labelslist, tweets):
        for word in preProcess_tweet(tweet):
            pair = (word, y_label)
            if pair in freqs:
                freqs[pair] += 1
            else:
                freqs[pair] = 1
```

In [73]:
```
iterations = 0;
for item in freqs:
    iterations+=1
    result=''
    result+='\''+item[0]+'\''+' appears '+ str(freqs[item])
    if(item[1]==1.0):
        result+=' in positive tweets'
    else:
```

```
        result+=' in negative tweets'
    print(result)
    if(iterations==5):
        break
```

```
'followfriday' appears 25 in positive tweets
'top' appears 32 in positive tweets
'engag' appears 7 in positive tweets
'member' appears 16 in positive tweets
'commun' appears 33 in positive tweets
```

Therefore, if we can summarize of the function, the result is the following:

In [76]:
```
def build_freqs(tweets_dataset, ylabels):

    # Convert np array to list since zip needs an iterable.
    # The squeeze is necessary or the list ends up with one element.
    # Also note that this is just a NOP if ys is already a list.
    ylabelslist = np.squeeze(ylabels).tolist()

    # Start with an empty dictionary and populate it by looping over all tweets
    # and over all processed words in each tweet.
    freqs = {}
    for y, tweet in zip(ylabelslist, tweets_dataset):
        for token in preProcess_tweet(tweet):
            pair = (token, y)
            if pair in freqs:
                freqs[pair] += 1
            else:
                freqs[pair] = 1

    return freqs
```

Table of word counts

We will select a set of words that we would like to visualize. It is better to store this temporary information in a table that is very easy to use later.

In [77]:
```
freqs = build_freqs(tweets,labels)
```

In [83]:
```
# Select some words to appear in the report. We will assume that each word is unique (i.e. no duplicates)
keys = ['happi', 'merri', 'nice', 'good', 'bad', 'sad', 'mad', 'best', 'pretti',
```

```
        '❤', ':)', ':(', '😒', '😬', '😄', '😍', '♛',
        'song', 'idea', 'power', 'play', 'magnific']

# List representing our table of word counts.
# Each element of data list, consist of a sublist with this pattern:
#     [<word>, <positive_count>, <negative_count>]
data = []

# loop through our selected words
for token in keys:

    # Initialize positive and negative counts
    pos = 0
    neg = 0

    # Retrieve number of positive counts
    # If token exists as a positively indexed token in our frequencies dictionary
    if (token, 1.0) in freqs:
        pos = freqs[(token, 1.0)]
    else:
        pos = 0

    # retrieve number of negative counts
    # If token exists as a positively indexed token in our frequencies dictionary
    if (token, 0.0) in freqs:
        neg = freqs[(token, 0.0)]
    else:
        neg = 0

    # append the word counts to the table
    data.append([token, pos, neg])
```

In [89]:
```
print(data)
```

```
[['happi', 212, 25], ['merri', 1, 0], ['nice', 99, 19], ['good', 238, 101], ['bad', 18, 73], ['sad', 5, 123], ['mad', 4, 11],
['best', 65, 22], ['pretti', 20, 15], ['❤', 29, 21], [':)', 3691, 2], [':(', 1, 4584], ['😒', 2, 3], ['😬', 0, 2], ['😄', 5,
1], ['😍', 5, 1], ['♛', 0, 210], ['song', 22, 27], ['idea', 27, 10], ['power', 7, 6], ['play', 46, 48], ['magnific', 2, 0]]
```

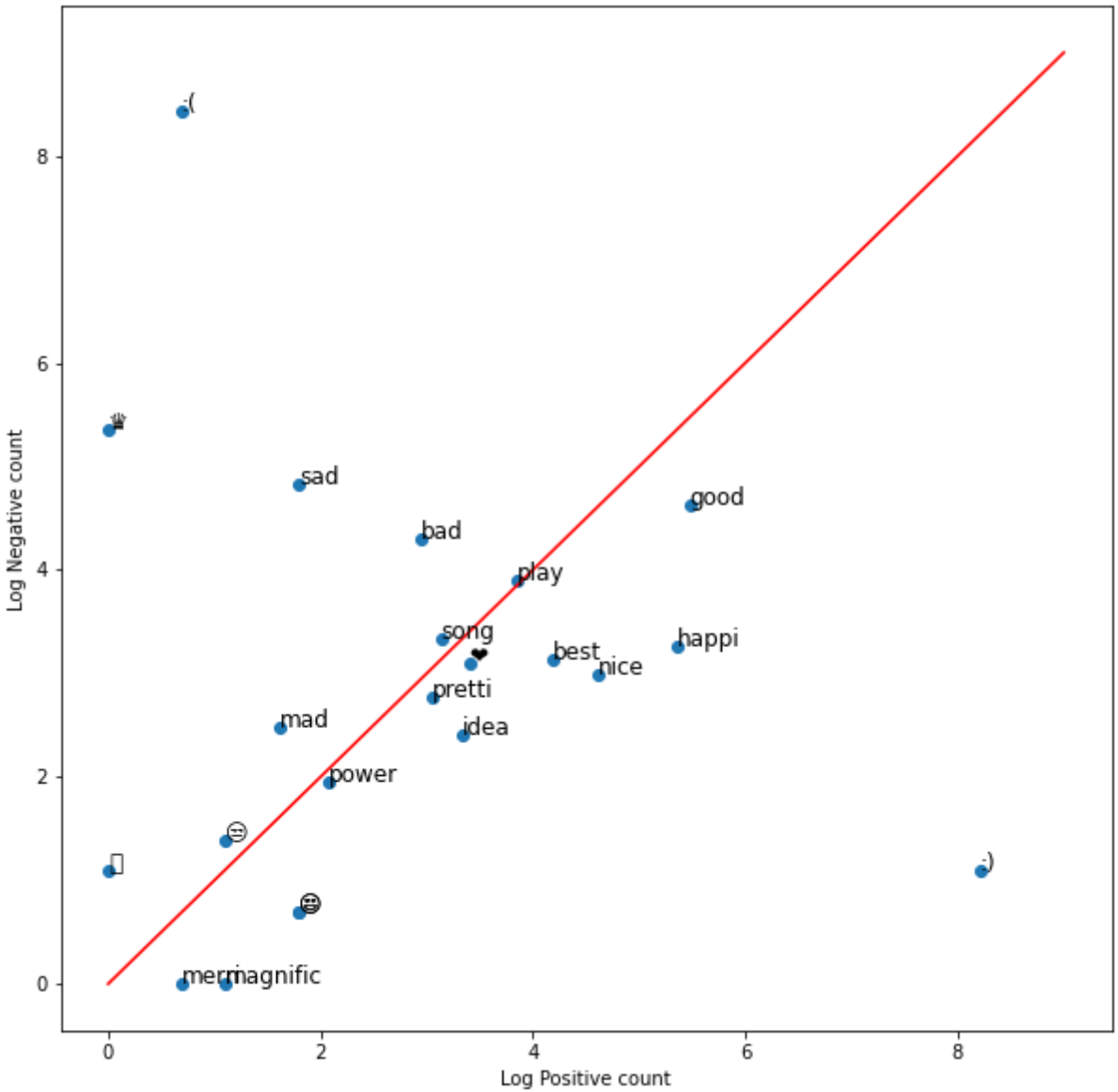
Visualization

We can then use a scatter plot to inspect this table visually. Instead of plotting the raw counts exact number, we will plot it in the logarithmic scale, to take into account the wide discrepancies between the raw counts. Example: ':)' has 3691 counts marked positively while only 2 in marked

negatively.

The red line marks the boundary between positive and negative areas. Tokens close to the red line can be classified as neutral.

In [88]:
```
fig, ax = plt.subplots(figsize = (10, 10))

# convert positive raw counts to logarithmic scale. we add 1 to avoid log(0)
x = np.log([x[1] + 1 for x in data])

# do the same for the negative counts
y = np.log([x[2] + 1 for x in data])

# Plot a dot for each pair of words
ax.scatter(x, y)

# assign axis labels
plt.xlabel("Log Positive count")
plt.ylabel("Log Negative count")

# Add the word as the label at the same position as you added the points just before
for i in range(0, len(data)):
    ax.annotate(data[i][0], (x[i], y[i]), fontsize=12)

ax.plot([0, 9], [0, 9], color = 'red') # Plot the red line that divides the 2 areas.
plt.show()
```



This chart is straightforward to interpret. It shows that emoticons ':)' and ':(' are very important for sentiment analysis. Thus, we should not let preprocessing steps get rid of these symbols!

Furthermore, it seems that the meaning of the crown symbol to be very negative!