Assignment 2: Naive Bayes

Welcome to week two of this specialization. You will learn about Naive Bayes. Concretely, you will be using Naive Bayes for sentiment analysis on tweets. Given a tweet, you will decide if it has a positive sentiment or a negative one. Specifically you will:

- Train a naive bayes model on a sentiment analysis task
- Test using your model
- Compute ratios of positive words to negative words
- Do some error analysis
- Predict on your own tweet

You may already be familiar with Naive Bayes and its justification in terms of conditional probabilities and independence.

- In this week's lectures and assignments we used the ratio of probabilities between positive and negative sentiment.
- This approach gives us simpler formulas for these 2-way classification tasks.

Important Note on Submission to the AutoGrader

Before submitting your assignment to the AutoGrader, please make sure you are not doing the following:

1. You have not added any *extra* `print` statement(s) in the assignment.
2. You have not added any *extra* code cell(s) in the assignment.
3. You have not changed any of the function parameters.
4. You are not using any global variables inside your graded exercises. Unless specifically instructed to do so, please refrain from it and use the local variables instead.
5. You are not changing the assignment code where it is not required, like creating *extra* variables.

If you do any of the following, you will get something like, `Grader Error: Grader feedback not found` (or similarly unexpected) error upon submitting your assignment. Before asking for help/debugging the errors in your assignment, check for these first. If this is the case, and you don't remember the changes you have made, you can get a fresh copy of the assignment by following these instructions.

Lets get started!

Load the cell below to import some packages. You may want to browse the documentation of unfamiliar libraries and functions.

Table of Contents

Importing Functions and Data

```python
from utils import process_tweet, lookup
import pdb
from nltk.corpus import stopwords, twitter_samples
import numpy as np
import pandas as pd
import nltk
import string
from nltk.tokenize import TweetTokenizer
from os import getcwd
import w2_unittest

nltk.download('twitter_samples')
nltk.download('stopwords')
```

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

```
True
```

If you are running this notebook in your local computer, don't forget to download the tweeter samples and stopwords from nltk.

```python
nltk.download('stopwords')
nltk.download('twitter_samples')
```

```
filePath = f"{getcwd()}/../tmp2/"
```

```

# get the sets of positive and negative tweets
all_positive_tweets = twitter_samples.strings('positive_tweets.json')
all_negative_tweets = twitter_samples.strings('negative_tweets.json')

# split the data into two pieces, one for training and one for testing (validation set)
test_pos = all_positive_tweets[4000:]
train_pos = all_positive_tweets[:4000]
test_neg = all_negative_tweets[4000:]
train_neg = all_negative_tweets[:4000]

train_x = train_pos + train_neg
test_x = test_pos + test_neg

# avoid assumptions about the length of all_positive_tweets
train_y = np.append(np.ones(len(train_pos)), np.zeros(len(train_neg)))
```

1 - Process the Data

For any machine learning project, once you've gathered the data, the first step is to process it to make useful inputs to your model.

- **Remove noise**: You will first want to remove noise from your data -- that is, remove words that don't tell you much about the content. These include all common words like 'I, you, are, is, etc...' that would not give us enough information on the sentiment.
- We'll also remove stock market tickers, retweet symbols, hyperlinks, and hashtags because they can not tell you a lot of information on the sentiment.
- You also want to remove all the punctuation from a tweet. The reason for doing this is because we want to treat words with or without the punctuation as the same word, instead of treating "happy", "happy?", "happy!", "happy," and "happy." as different words.
- Finally you want to use stemming to only keep track of one variation of each word. In other words, we'll treat "motivation", "motivated", and "motivate" similarly by grouping them within the same stem of "motiv-".

We have given you the function `process_tweet` that does this for you.

```
custom_tweet = "RT @Twitter @chapagain Hello There! Have a great day. :) #good #morning http://chapagain.com.np"

# print cleaned tweet
```

1.1 - Implementing your Helper Functions

To help you train your naive bayes model, you will need to compute a dictionary where the keys are a tuple (word, label) and the values are the corresponding frequency. Note that the labels we'll use here are 1 for positive and 0 for negative.

You will also implement a lookup helper function that takes in the `freqs` dictionary, a word, and a label (1 or 0) and returns the number of times that word and label tuple appears in the collection of tweets.

For example: given a list of tweets `["i am rather excited", "you are rather happy"]` and the label 1, the function will return a dictionary that contains the following key-value pairs:

{ ("rather", 1): 2, ("happi", 1) : 1, ("excit", 1) : 1 }

- Notice how for each word in the given string, the same label 1 is assigned to each word.
- Notice how the words "i" and "am" are not saved, since it was removed by process_tweet because it is a stopword.
- Notice how the word "rather" appears twice in the list of tweets, and so its count value is 2.

Exercise 1 - count_tweets

Create a function `count_tweets` that takes a list of tweets as input, cleans all of them, and returns a dictionary.

- The key in the dictionary is a tuple containing the stemmed word and its class label, e.g. ("happi",1).
- The value the number of times this word appears in the given collection of tweets (an integer).

Hints

```python
# UNQ_C1 GRADED FUNCTION: count_tweets

def count_tweets(result, tweets, ys):
    '''
    Input:
        result: a dictionary that will be used to map each pair to its frequency
        tweets: a list of tweets
        ys: a list corresponding to the sentiment of each tweet (either 0 or 1)
    Output:
        result: a dictionary mapping each pair to its frequency
    '''
    ### START CODE HERE ###
    for y, tweet in zip(ys, tweets):
        for word in process_tweet(tweet):
            # define the key, which is the word and label tuple
            pair = (word, y)

            # if the key exists in the dictionary, increment the count
            if pair in result:
                result[pair] += 1
            # else, if the key is new, add it with count 1
            else:
                result[pair] = 1
    ### END CODE HERE ###

    return result
```

```python
# Testing your function

result = {}
tweets = ['i am happy', 'i am tricked', 'i am sad', 'i am tired', 'i am tired']
ys = [1, 0, 0, 0, 0]
count_tweets(result, tweets, ys)
```

```
{('happi', 1): 1, ('trick', 0): 1, ('sad', 0): 1, ('tire', 0): 2}
```

Expected Output: {('happi', 1): 1, ('trick', 0): 1, ('sad', 0): 1, ('tire', 0): 2}

```python
#Test your function
w2_unittest.test_count_tweets(count_tweets)
```

```
 All tests passed
```

2 - Train your Model using Naive Bayes

Naive bayes is an algorithm that could be used for sentiment analysis. It takes a short time to train and also has a short prediction time.

So how do you train a Naive Bayes classifier?

- The first part of training a naive bayes classifier is to identify the number of classes that you have.
- You will create a probability for each class. $P(D_{pos})$ is the probability that the document is positive. $P(D_{neg})$ is the probability that the document is negative. Use the formulas as follows and store the values in a dictionary:

$$P(D_{pos}) = \frac{D_{pos}}{D} \tag{1}$$

$$P(D_{neg}) = \frac{D_{neg}}{D} \tag{2}$$

Where D is the total number of documents, or tweets in this case, D_{pos} is the total number of positive tweets and D_{neg} is the total number of negative tweets.

Prior and Logprior

The prior probability represents the underlying probability in the target population that a tweet is positive versus negative. In other words, if we had no specific information and blindly picked a tweet out of the population set, what is the probability that it will be positive versus that it will be negative? That is the "prior".

The prior is the ratio of the probabilities $\frac{P(D_{pos})}{P(D_{neg})}$. We can take the log of the prior to rescale it, and we'll call this the logprior

$$\text{logprior} = log \left(\frac{P(D_{pos})}{P(D_{neg})} \right) = log \left(\frac{D_{pos}}{D_{neg}} \right)$$

.

Note that $log(\frac{A}{B})$ is the same as $log(A) - log(B)$. So the logprior can also be calculated as the difference between two logs:

$$\text{logprior} = \log(P(D_{pos})) - \log(P(D_{neg})) = \log(D_{pos}) - \log(D_{neg}) \tag{3}$$

Positive and Negative Probability of a Word

To compute the positive probability and the negative probability for a specific word in the vocabulary, we'll use the following inputs:

- $freq_{pos}$ and $freq_{neg}$ are the frequencies of that specific word in the positive or negative class. In other words, the positive frequency of a word is the number of times the word is counted with the label of 1.

- N_{pos} and N_{neg} are the total number of positive and negative words for all documents (for all tweets), respectively.
- V is the number of unique words in the entire set of documents, for all classes, whether positive or negative.

We'll use these to compute the positive and negative probability for a specific word using this formula:

$$P(W_{pos}) = \frac{freq_{pos} + 1}{N_{pos} + V} \tag{4}$$
$$P(W_{neg}) = \frac{freq_{neg} + 1}{N_{neg} + V} \tag{5}$$

Notice that we add the "+1" in the numerator for additive smoothing. This wiki article explains more about additive smoothing.

Log likelihood

To compute the loglikelihood of that very same word, we can implement the following equations:

$$\text{loglikelihood} = \log \left(\frac{P(W_{pos})}{P(W_{neg})} \right) \tag{6}$$

Create `freqs` dictionary

- Given your `count_tweets` function, you can compute a dictionary called `freqs` that contains all the frequencies.
- In this `freqs` dictionary, the key is the tuple (word, label)
- The value is the number of times it has appeared.

We will use this dictionary in several parts of this assignment.

```
# Build the freqs dictionary for later uses
freqs = count_tweets({}, train_x, train_y)
```

Exercise 2 - train_naive_bayes

Given a freqs dictionary, `train_x` (a list of tweets) and a `train_y` (a list of labels for each tweet), implement a naive bayes classifier.

Calculate V

- You can then compute the number of unique words that appear in the `freqs` dictionary to get your V (you can use the `set` function).

Calculate $freq_{pos}$ and $freq_{neg}$

- Using your `freqs` dictionary, you can compute the positive and negative frequency of each word $freq_{pos}$ and $freq_{neg}$.

Calculate N_{pos}, and N_{neg}

- Using `freqs` dictionary, you can also compute the total number of positive words and total number of negative words N_{pos} and N_{neg}.

Calculate D, D_{pos}, D_{neg}

- Using the `train_y` input list of labels, calculate the number of documents (tweets) D, as well as the number of positive documents (tweets) D_{pos} and number of negative documents (tweets) D_{neg}.
- Calculate the probability that a document (tweet) is positive $P(D_{pos})$, and the probability that a document (tweet) is negative $P(D_{neg})$

Calculate the logprior

- the logprior is $log(D_{pos}) - log(D_{neg})$

Calculate log likelihood

- Finally, you can iterate over each word in the vocabulary, use your `lookup` function to get the positive frequencies, $freq_{pos}$, and the negative frequencies, $freq_{neg}$, for that specific word.
- Compute the positive probability of each word $P(W_{pos})$, negative probability of each word $P(W_{neg})$ using equations 4 & 5.

$$P(W_{pos}) = \frac{freq_{pos} + 1}{N_{pos} + V} \tag{4}$$
$$P(W_{neg}) = \frac{freq_{neg} + 1}{N_{neg} + V} \tag{5}$$

Note: We'll use a dictionary to store the log likelihoods for each word. The key is the word, the value is the log likelihood of that word).

- You can then compute the loglikelihood: $log \left(\frac{P(W_{pos})}{P(W_{neg})} \right)$.

```python
# UNQ_C2 GRADED FUNCTION: train_naive_bayes

import numpy as np

def train_naive_bayes(freqs, train_x, train_y):
    '''
    Input:
        freqs: dictionary from (word, label) to how often the word appears
        train_x: a list of tweets
        train_y: a list of labels corresponding to the tweets (0,1)
    Output:
        logprior: the log prior
        loglikelihood: the log likelihood dictionary
    '''
    loglikelihood = {}
    logprior = 0

    ### START CODE HERE ###
    # vocabulary (set of unique words)
    vocab = set([pair[0] for pair in freqs.keys()])
    V = len(vocab)

    # initialize counts
    N_pos = 0
    N_neg = 0

    # count words by class
    for pair, count in freqs.items():
        if pair[1] > 0:
            N_pos += count
        else:
            N_neg += count

    # number of documents
    D = len(train_y)
    D_pos = np.sum(train_y)
    D_neg = D - D_pos

    # prior
    logprior = np.log(D_pos) - np.log(D_neg)

    # likelihoods
    for word in vocab:
        # frequency of word in positive and negative class
        freq_pos = freqs.get((word, 1), 0)
        freq_neg = freqs.get((word, 0), 0)

        # Laplace-smoothed likelihoods
        p_w_pos = (freq_pos + 1) / (N_pos + V)
        p_w_neg = (freq_neg + 1) / (N_neg + V)

        # log likelihood ratio
        loglikelihood[word] = np.log(p_w_pos) - np.log(p_w_neg)
    ### END CODE HERE ###

    return logprior, loglikelihood
```

```python
# UNQ_C3 (UNIQUE CELL IDENTIFIER, DO NOT EDIT)
logprior, loglikelihood = train_naive_bayes(freqs, train_x, train_y)
print(logprior)
print(len(loglikelihood))
```

```
0.0
9147
```

Expected Output:

0.0

9147

```python
# Test your function
w2_unittest.test_train_naive_bayes(train_naive_bayes, freqs, train_x, train_y)
```

```
 All tests passed
```

3 - Test your Naive Bayes

Now that we have the `logprior` and `loglikelihood`, we can test the naive bayes function by making predicting on some tweets!

Exercise 3 - naive_bayes_predict

Implement `naive_bayes_predict`.

Instructions: Implement the `naive_bayes_predict` function to make predictions on tweets.

- The function takes in the `tweet`, `logprior`, `loglikelihood`.
- It returns the probability that the tweet belongs to the positive or negative class.

- For each tweet, sum up loglikelihoods of each word in the tweet.
- Also add the logprior to this sum to get the predicted sentiment of that tweet.

$$p = logprior + \sum_i^N (loglikelihood_i)$$

Note

Note we calculate the prior from the training data, and that the training data is evenly split between positive and negative labels (4000 positive and 4000 negative tweets). This means that the ratio of positive to negative 1, and the logprior is 0.

The value of 0.0 means that when we add the logprior to the log likelihood, we're just adding zero to the log likelihood. However, please remember to include the logprior, because whenever the data is not perfectly balanced, the logprior will be a non-zero value.

```python
# UNQ_C4 GRADED FUNCTION: naive_bayes_predict

def naive_bayes_predict(tweet, logprior, loglikelihood):
    '''
    Input:
        tweet: a string
        logprior: a number
        loglikelihood: a dictionary of words mapping to numbers
    Output:
        p: the sum of all the loglikelihoods of each word in the tweet
           (if found in the dictionary) + logprior (a number)
    '''
    ### START CODE HERE ###
    # process the tweet to get a list of words
    word_l = process_tweet(tweet)

    # initialize probability as logprior
    p = logprior

    # add contributions from words
    for word in word_l:
        if word in loglikelihood:
            p += loglikelihood[word]
    ### END CODE HERE ###

    return p
```

```python
# UNQ_C5 (UNIQUE CELL IDENTIFIER, DO NOT EDIT)
my_tweet = 'She smiled.'
p = naive_bayes_predict(my_tweet, logprior, loglikelihood)
print('The expected output is', p)
```

```
The expected output is 1.5545735052281646
```

Expected Output:

- The expected output is around 1.55
- The sentiment is positive.

```python
# Test your function
w2_unittest.test_naive_bayes_predict(naive_bayes_predict)
```

```
 All tests passed
```

```python
# Experiment with your own tweet.
my_tweet = 'He laughed.'
p = naive_bayes_predict(my_tweet, logprior, loglikelihood)
print('The expected output is', p)
```

```
The expected output is -0.16819309251293824
```

Exercise 4 - test_naive_bayes

Implement test_naive_bayes.

Instructions:

- Implement `test_naive_bayes` to check the accuracy of your predictions.
- The function takes in your `test_x`, `test_y`, log_prior, and loglikelihood
- It returns the accuracy of your model.
- First, use `naive_bayes_predict` function to make predictions for each tweet in text_x.

```python
# UNQ_C6 GRADED FUNCTION: test_naive_bayes

import numpy as np

def test_naive_bayes(test_x, test_y, logprior, loglikelihood, naive_bayes_predict=naive_bayes_predict):
    """
    Input:
        test_x: A list of tweets
        test_y: the corresponding labels for the list of tweets
        logprior: the logprior
        loglikelihood: a dictionary with the loglikelihoods for each word
    Output:
        accuracy: (# of tweets classified correctly)/(total # of tweets)
    """
    ### START CODE HERE ###
    y_hats = []

    for tweet in test_x:
        # if the prediction is > 0 → predict positive
        if naive_bayes_predict(tweet, logprior, loglikelihood) > 0:
            y_hat_i = 1
        else:
            y_hat_i = 0
        y_hats.append(y_hat_i)

    # convert to numpy arrays
    y_hats = np.array(y_hats)
    test_y = np.array(test_y)

    # error = average absolute difference
    error = np.mean(np.abs(y_hats - test_y))

    # accuracy = 1 - error
    accuracy = 1 - error
    ### END CODE HERE ###

    return accuracy
```

```python
print("Naive Bayes accuracy = %0.4f" %
      (test_naive_bayes(test_x, test_y, logprior, loglikelihood)))
```

Expected Accuracy:

```
Naive Bayes accuracy = 0.9955
```

```python
# UNQ_C7 (UNIQUE CELL IDENTIFIER, DO NOT EDIT)
# Run this cell to test your function
for tweet in ['I am happy', 'I am bad', 'this movie should have been great.', 'great', 'great great', 'great great great', 'great g
    # print( '%s -> %f' % (tweet, naive_bayes_predict(tweet, logprior, loglikelihood)))
    p = naive_bayes_predict(tweet, logprior, loglikelihood)
#     print(f'{tweet} -> {p:.2f} ({p_category})')
    print(f'{tweet} -> {p:.2f}')
```

```
I am happy -> 2.14
I am bad -> -1.31
this movie should have been great. -> 2.11
great -> 2.13
great great -> 4.25
great great great -> 6.38
great great great great -> 8.50
```

Expected Output:

- I am happy -> 2.14
- I am bad -> -1.31
- this movie should have been great. -> 2.11
- great -> 2.13
- great great -> 4.25
- great great great -> 6.38
- great great great great -> 8.50

```python
# Feel free to check the sentiment of your own tweet below
my_tweet = 'you are bad :('
naive_bayes_predict(my_tweet, logprior, loglikelihood)
```

```
-8.843801112417253
```

```python
# Test your function
w2_unittest.unittest_test_naive_bayes(test_naive_bayes, test_x, test_y)
```

```
 All tests passed
```

4 - Filter words by Ratio of Positive to Negative Counts

- Some words have more positive counts than others, and can be considered "more positive". Likewise, some words can be considered more negative than others.

- One way for us to define the level of positiveness or negativeness, without calculating the log likelihood, is to compare the positive to negative frequency of the word.
  - Note that we can also use the log likelihood calculations to compare relative positivity or negativity of words.
- We can calculate the ratio of positive to negative frequencies of a word.
- Once we're able to calculate these ratios, we can also filter a subset of words that have a minimum ratio of positivity / negativity or higher.
- Similarly, we can also filter a subset of words that have a maximum ratio of positivity / negativity or lower (words that are at least as negative, or even more negative than a given threshold).

Exercise 5 - get_ratio

Implement get_ratio.

- Given the freqs dictionary of words and a particular word, use `lookup(freqs,word,1)` to get the positive count of the word.
- Similarly, use the `lookup` function to get the negative count of that word.
- Calculate the ratio of positive divided by negative counts

$$ratio = \frac{\text{pos\_words} + 1}{\text{neg\_words} + 1}$$

Where pos_words and neg_words correspond to the frequency of the words in their respective classes.

Words	Positive word count	Negative Word Count
glad	41	2
arriv	57	4
:(	1	3663

```python
# UNQ_C8 GRADED FUNCTION: get_ratio

def get_ratio(freqs, word):
    '''
    Input:
        freqs: dictionary containing the words

    Output: a dictionary with keys 'positive', 'negative', and 'ratio'.
        Example: {'positive': 10, 'negative': 20, 'ratio': 0.5}
    '''
    pos_neg_ratio = {'positive': 0, 'negative': 0, 'ratio': 0.0}
    ### START CODE HERE ###
    # lookup positive and negative counts
    pos_neg_ratio['positive'] = lookup(freqs, word, 1)
    pos_neg_ratio['negative'] = lookup(freqs, word, 0)

    # ratio (Laplace smoothed)
    pos_neg_ratio['ratio'] = (pos_neg_ratio['positive'] + 1) / (pos_neg_ratio['negative'] + 1)
    ### END CODE HERE ###
    return pos_neg_ratio
```

```python
get_ratio(freqs, 'happi')
```

```
{'positive': 162, 'negative': 18, 'ratio': 8.578947368421053}
```

```python
# Test your function
w2_unittest.test_get_ratio(get_ratio, freqs)
```

```
 All tests passed
```

Exercise 6 - get_words_by_threshold

Implement get_words_by_threshold(freqs,label,threshold)

- If we set the label to 1, then we'll look for all words whose threshold of positive/negative is at least as high as that threshold, or higher.
- If we set the label to 0, then we'll look for all words whose threshold of positive/negative is at most as low as the given threshold, or lower.
- Use the `get_ratio` function to get a dictionary containing the positive count, negative count, and the ratio of positive to negative counts.
- Append the `get_ratio` dictionary inside another dictinoary, where the key is the word, and the value is the dictionary `pos_neg_ratio` that is returned by the `get_ratio` function. An example key-value pair would have this structure:

```
{'happi':
  {'positive': 10, 'negative': 20, 'ratio': 0.524}
}
```

```
# UNQ_C9 GRADED FUNCTION: get_words_by_threshold

def get_words_by_threshold(freqs, label, threshold, get_ratio=get_ratio):
    '''
    Input:
        freqs: dictionary of words
        label: 1 for positive, 0 for negative
        threshold: ratio that will be used as the cutoff
    Output:
        word_list: dictionary of words meeting the condition
    '''
    word_list = {}

    ### START CODE HERE ###
    for key in freqs.keys():
        word, _ = key   # unpack word and its label (ignore label part)

        # get pos/neg ratio for the word
        pos_neg_ratio = get_ratio(freqs, word)

        # Positive case: ratio >= threshold
        if label == 1 and pos_neg_ratio['ratio'] >= threshold:
            word_list[word] = pos_neg_ratio

        # Negative case: ratio <= threshold
        elif label == 0 and pos_neg_ratio['ratio'] <= threshold:
            word_list[word] = pos_neg_ratio
    ### END CODE HERE ###
```

```
# Test your function: find negative words at or below a threshold
```

```
{':(': {'positive': 1, 'negative': 3675, 'ratio': 0.000544069640914037},
 ':-(': {'positive': 0, 'negative': 386, 'ratio': 0.002583979328165375},
 'zayniscomingbackonjuli': {'positive': 0, 'negative': 19, 'ratio': 0.05},
 '26': {'positive': 0, 'negative': 20, 'ratio': 0.047619047619047616},
 '>:(': {'positive': 0, 'negative': 43, 'ratio': 0.022727272727272728},
 'lost': {'positive': 0, 'negative': 19, 'ratio': 0.05},
 '♛': {'positive': 0, 'negative': 210, 'ratio': 0.004739336492890996},
 '》': {'positive': 0, 'negative': 210, 'ratio': 0.004739336492890996},
 'believ': {'positive': 0, 'negative': 35, 'ratio': 0.027777777777777776},
 'will': {'positive': 0, 'negative': 35, 'ratio': 0.027777777777777776},
 'justin': {'positive': 0, 'negative': 35, 'ratio': 0.027777777777777776},
 'ｓｅｅ': {'positive': 0, 'negative': 35, 'ratio': 0.027777777777777776},
 'ｍｅ': {'positive': 0, 'negative': 35, 'ratio': 0.027777777777777776}}
```

```
# Test your function; find positive words at or above a threshold
```

```
{'followfriday': {'positive': 23, 'negative': 0, 'ratio': 24.0},
 'commun': {'positive': 27, 'negative': 1, 'ratio': 14.0},
 ':)': {'positive': 2960, 'negative': 2, 'ratio': 987.0},
 'flipkartfashionfriday': {'positive': 16, 'negative': 0, 'ratio': 17.0},
 ':D': {'positive': 523, 'negative': 0, 'ratio': 524.0},
 ':p': {'positive': 104, 'negative': 0, 'ratio': 105.0},
 'influenc': {'positive': 16, 'negative': 0, 'ratio': 17.0},
 ':-)': {'positive': 552, 'negative': 0, 'ratio': 553.0},
 "here'": {'positive': 20, 'negative': 0, 'ratio': 21.0},
 'youth': {'positive': 14, 'negative': 0, 'ratio': 15.0},
 'bam': {'positive': 44, 'negative': 0, 'ratio': 45.0},
 'warsaw': {'positive': 44, 'negative': 0, 'ratio': 45.0},
 'shout': {'positive': 11, 'negative': 0, 'ratio': 12.0},
 ';)': {'positive': 22, 'negative': 0, 'ratio': 23.0},
 'stat': {'positive': 51, 'negative': 0, 'ratio': 52.0},
 'arriv': {'positive': 57, 'negative': 4, 'ratio': 11.6},
 'glad': {'positive': 41, 'negative': 2, 'ratio': 14.0},
 'blog': {'positive': 27, 'negative': 0, 'ratio': 28.0},
 'fav': {'positive': 11, 'negative': 0, 'ratio': 12.0},
 'fantast': {'positive': 9, 'negative': 0, 'ratio': 10.0},
 'fback': {'positive': 26, 'negative': 0, 'ratio': 27.0},
 'pleasur': {'positive': 10, 'negative': 0, 'ratio': 11.0},
 '←': {'positive': 9, 'negative': 0, 'ratio': 10.0},
 'aqui': {'positive': 9, 'negative': 0, 'ratio': 10.0}}
```

Notice the difference between the positive and negative ratios. Emojis like :(and words like 'me' tend to have a negative connotation. Other words like glad, community, arrives, tend to be found in the positive tweets.

```
# Test your function
```

```
 All tests passed
```

5 - Error Analysis

In this part you will see some tweets that your model missclassified. Why do you think the missclassifications happened? Were there any assumptions made by your naive bayes model?

```
# Some error analysis done for you
print('Truth Predicted Tweet')
for x, y in zip(test_x, test_y):
    y_hat = naive_bayes_predict(x, logprior, loglikelihood)
    if y != (np.sign(y_hat) > 0):
        print('%d\t%0.2f\t%s' % (y, np.sign(y_hat) > 0, ' '.join(
            process_tweet(x)).encode('ascii', 'ignore')))
```

```
Truth Predicted Tweet
1       0.00    b'truli later move know queen bee upward bound movingonup'
1       0.00    b'new report talk burn calori cold work harder warm feel better weather :p'
1       0.00    b'harri niall 94 harri born ik stupid wanna chang :D'
1       0.00    b'park get sunlight'
1       0.00    b'uff itna miss karhi thi ap :p'
0       1.00    b'hello info possibl interest jonatha close join beti :( great'
0       1.00    b'u prob fun david'
0       1.00    b'pat jay'
0       1.00    b'sr financi analyst expedia inc bellevu wa financ expediajob job job hire'
```

6 - Predict with your own Tweet

In this part you can predict the sentiment of your own tweet.

```
# Test with your own tweet - feel free to modify `my_tweet`
my_tweet = 'I am happy because I am learning :)'

p = naive_bayes_predict(my_tweet, logprior, loglikelihood)
print(p)
```

```
9.561469810952186
```

Congratulations on completing this assignment. See you next week!