Final Python Report

Submitted on 14/12/18

Stephen O'Driscoll

R00146853

Stephen.odriscoll3@mycit.ie

Declaration

"By submitting this electronically, using an account & password only I have access to, I hereby certify that this material which I now submit for assessment is entirely my own work and has not been taken from the work of others, save and to the extent, that such work has been cited and acknowledged within the text of my work."

Introduction

The goal of this project is to build a learning algorithm that analyses several tweets from two documents, positive and negative and learns to classify unseen tweets from this analysis. This is done by building a Naïve Bayes Multinomial Model, then using Multinomial Text Classification to calculate probabilities of each unique word being related to either positive or negative.

We then test this algorithm on the data used to create it as well as another dataset to confirm the algorithm functions on unseen tweets. We do this by dividing each tweet into individual words and use our previous classifications to assign a probability that each word is positive or negative. We use logarithms to eliminate computational problems.

Naïve Bayes is a very fast algorithm and is one of the most successful known algorithms for learning to classify text documents. I explain every portion of my code below.

Main

My main method is called at the bottom of the program. It starts the program and calls three different methods twice to build my algorithm. These methods read in the document, process this text into a dictionary where key is every unique word and value is the frequency of that word (Naïve Bayes Multinomial Model) and finally we calculate a probability that each word is either positive or negative using Multinomial Text Classification and save this to another dictionary. This is our setup.

When this setup is complete we call the method analyse several times to classify tweets in other documents as positive or negative, display and return the accuracy with which it performs this task. We then visualize our chart at the end and display the time the program took to run.

```
def main():
    start_time = time.time()

    # Read in positive and negative training files and make them lower case
    doc_train_pos = read("train/trainPos.txt")
    doc_train_neg = read("train/trainNeg.txt")

    # Returns a dictionary with unique words as keys and their frequencies as values, also returns total words processed
    dict_count_pos, total_count_pos = process(doc_train_pos)
    dict_count_neg, total_count_neg = process(doc_train_neg)

    # Create a list of all unique words used in either positive or negative or both
    vocab_complete = set()
    vocab_complete.update(dict_count_pos.keys(), dict_count_neg.keys())

    # Use Naive Classification to calculate probability of each word being positive or negative
    dict_prob_pos = calculate(dict_count_pos, total_count_pos, vocab_complete)
    dict_prob_neg = calculate(dict_count_neg, total_count_neg, vocab_complete)

    # Display the number of positive and negative words as well as how long it took to get this far
    display_counts(total_count_pos, total_count_neg, start_time)

    print()     # Blank line

    # Calculate and display the accuracy for positive and negative training files
    accuracy_train_pos = analyse("Positive Training Analysis", doc_train_pos, dict_prob_pos, dict_prob_neg)
    accuracy_train_neg = analyse("Negative Training Analysis", doc_train_neg, dict_prob_pos, dict_prob_neg)

    print()     # Blank line
```

```
    # Calculate and display the accuracy for positive and negative testing files
    accuracy_test_pos = analyse("Positive Testing Analysis", read("test/testPos.txt"), dict_prob_pos, dict_prob_neg)
    accuracy_test_neg = analyse("Negative Testing Analysis", read("test/testNeg.txt"), dict_prob_pos, dict_prob_neg)

    # Create bar chart based on results of every accuracy test and print total time program has taken
    visualize(accuracy_train_pos, accuracy_train_neg, accuracy_test_pos, accuracy_test_neg)
    print("\n\nTotal time taken: " + str(time.time() - start_time))
    exit(0)
```

Analyse

Analyse takes in parameters for the title of the document its analysing, the document itself and both probability dictionaries to use to classify document tweets and display results. It records the time of each analysis and displays this as well. It then returns the accuracy with which it identified the tweets.

```
# Analyse each document and return final accuracy
def analyse(title, document, dict_prob_pos, dict_prob_neg):

    start_time = time.time()
    count_pos, count_neg = classify(document, dict_prob_pos, dict_prob_neg)
    return display(title, count_pos, count_neg, start_time)
```

Read

Opens file and makes all text lowercase to remove ambiguity.

```
# Reads files and makes all letters lower case
def read(directory):
    return open(directory).read().lower()
```

Clean

The pattern used here is [^\w].

\w means characters A-Z, a-z and 0-9.

^ means not.

So, we remove all characters that aren't letters or numbers.

```
# Returns a string with special characters removed. Ignores letters and numbers
def clean(to_clean):
    return re.sub(r'[^\w]', ' ', to_clean)
```

Process

Takes in a document, cleans it and splits it into individual words. It also removes all words of length 2 or less, I found this to be the most effective and efficient way to remove meaningless words. I also saw a very noticeable drop in accuracy when this was length three. It seems three letter words can be meaningful to the sentiment of tweets. This method then iterates through each word and records how many times each unique word appears by saving it to a dictionary where key is every unique word and value is that words frequency. This is our Multinomial Model

```
# returns a dictionary of unique words and their frequencies in a given document - Multinomial Model
def process(document):

    words = clean(document).split()         # Clean
    vocab = set(words)
    dictionary = dict.fromkeys(vocab, 0)    # put all unique words in dictionary with 0 as value
    total = 0

    # If a words length is 2 or less ignore. I experimented and found this was the best way to remove meaningless words
    # It has consistently given me the highest accuracy at a good speed
    for word in words:
        if len(word) > 2:
            count = dictionary.get(word)
            dictionary[word] = count + 1        # Get unique word and increment frequency at every occurrence
            total += 1

    # Send back processed words with total words checked
    return dictionary, total
```

Calculate

We calculate the probability that each word is either positive or negative using Multinomial Text Classification. The formula used here is (frequency of word + 1) / (total number of words + total number of unique words). We return our probability dictionary where the key is every unique word and the value is the probability that this word is positive/negative.

```
# Uses Naive Bayes Multinomial Text Classification to calculate probability each word is positive/negative
def calculate(dictionary, total_count, vocab):
    result = {}

    # (count(w,c) + 1) / (count(c) + |V|)
    for word in vocab:
        if word not in dictionary:
            result[word] = 1 / (total_count + len(vocab))

        else:
            result[word] = (dictionary[word] + 1) / (total_count + len(vocab))

    return result
```

Classify

We classify the tweets in a document as being positive or negative using probability dictionaries. This is done by checking each word against probability dictionaries for positive and negative and adding the log of these probabilities into an overall positive and negative score. We use logarithms to eliminate computational problems. We then return the count for how many tweets were identified as positive and how many were identified as negative. We also remove all words of length 2 or less, as discussed above I found this to be very effective at filtering out meaningless words.

```
# Decides whether a tweet is positive or negative based on it's algorithm
def classify(document, dict_prob_pos, dict_prob_neg):

    # Number of positive and negative tweets to be incremented when a decision is made
    count_pos = 0
    count_neg = 0
    tweets = document.split("\n")       # Split document into tweets

    # Iterate through each tweet
    for tweet in tweets:

        prob_pos = 1
        prob_neg = 1
        words = clean(tweet).split()    # Clean out special characters

        # Same as before, I experimented and found this worked best at cleaning out meaningless words
        for word in words:
            if len(word) > 2:

                # If word exists is positive or negative assign probability using log
                if word in dict_prob_pos:
                    word_prob_pos = dict_prob_pos[word]
                    prob_pos += log(word_prob_pos)

                if word in dict_prob_neg:
                    word_prob_neg = dict_prob_neg[word]
                    prob_neg += log(word_prob_neg)

        # If final positive score is greater than negative increment positive
        if prob_pos > prob_neg:
            count_pos += 1

        # Check the opposite as well. It is very unlikely but a tweet can be neutral if scores are the same
        # I have checked and this never occurs with the given files but I left the check in anyway
        elif prob_pos < prob_neg:
            count_neg += 1

    return count_pos, count_neg
```

Display

Calculate our percentage accuracy, display and return this value. Display how long my program took to perform the classification on this document.

```
# Simply display data
def display(title, count_pos, count_neg, start_time):

    total = count_pos + count_neg
    percent_pos = count_pos / total * 100
    percent_neg = count_neg / total * 100
    accuracy = 0.0

    print(title + ":")

    # If title starts with word Positive we know to display positive accuracy
    if title.startswith('P'):
        print("\tPositive Accuracy:" + str(percent_pos) + "%")
        accuracy = percent_pos

    # Else we display negative accuracy
    elif title.startswith('N'):
        print("\tNegative Accuracy:" + str(percent_neg) + "%")
        accuracy = percent_neg

    # Display the time it took for this portion of the file
    print("\tTime Taken: " + str(time.time() - start_time) + " seconds")
    return accuracy
```

Display Count

Display how many words are in the positive dictionary and how many are in the negative dictionary as well as how long it took to setup the program and make it ready to begin classifying.

```
# Display counts for positive and negative words and setup time
def display_counts(count_pos, count_neg, start_time):

    print("\nPositive Count: " + str(count_pos))
    print("Negative Count: " + str(count_neg))
    print("Time Taken to Read, Process and Calculate: " + str(time.time() - start_time) + " seconds")
```

Visualize

Display bar chart of how accurate each classification was with the document name on x-axis and percentage accuracy on y-axis.

```
# Display bar chart visualization
def visualize(accuracy_train_pos, accuracy_train_neg, accuracy_test_pos, accuracy_test_neg):

    data = ('Train Positive', 'Train Negative',
            'Test Positive', 'Test Negative')

    plot = [accuracy_train_pos, accuracy_train_neg,
            accuracy_test_pos, accuracy_test_neg]
    y_pos = np.arange(len(data))
    plt.bar(y_pos, plot, align='center', alpha=0.1)
    plt.xticks(y_pos, data)
    plt.ylabel("Percentage")
    plt.xlabel("Analysis")
    plt.title("Program Accuracy")
    plt.show()
```

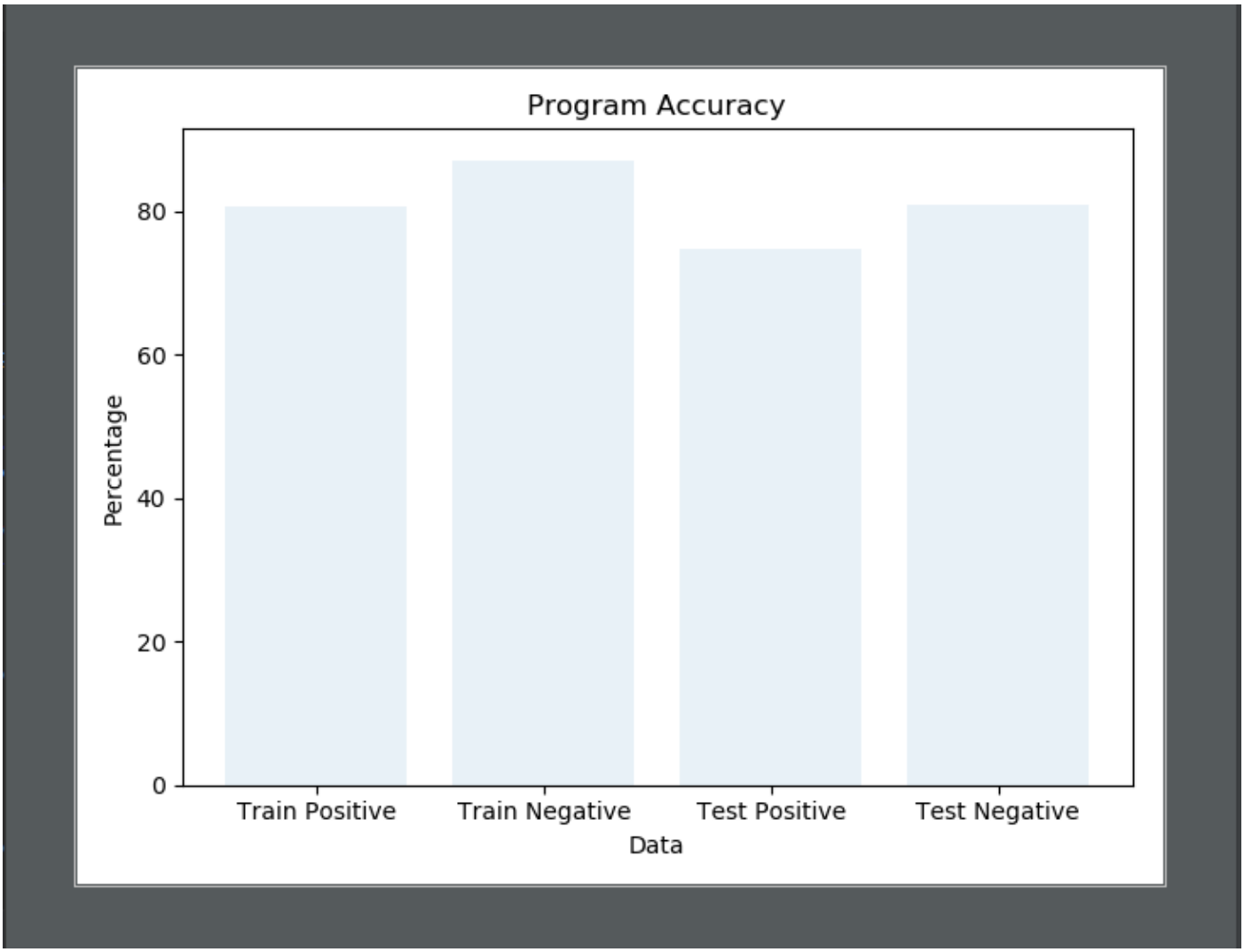
Output

My average accuracy when classifying the train files rounded to the nearest percent is: 84%.

My average accuracy when classifying the test files rounded to the nearest percent is: 78%.

```
Python 3.7.0 (v3.7.0:1bf9cc5093, Jun 27 2018, 04:59:51) [MSC v.1914 64 bit (AMD64)] on
>>> runfile('C:/Users/steph/Desktop/Year 3 - Semester 1/Programming for Data Analytics/

Positive Count: 4023880
Negative Count: 4101800
Time Taken to Read, Process and Calculate: 17.385955333709717 seconds

Positive Training Analysis:
    Positive Accuracy:80.76417955337226%
    Time Taken: 9.092191696166992 seconds
Negative Training Analysis:
    Negative Accuracy:87.10452773913806%
    Time Taken: 9.267363548278809 seconds

Positive Testing Analysis:
    Positive Accuracy:74.8%
    Time Taken: 0.031242847442626953 seconds
Negative Testing Analysis:
    Negative Accuracy:80.78078078078079%
    Time Taken: 0.015621662139892578 seconds

Total time taken: 35.95008587837219

Process finished with exit code 0
```

Bar Chart


Program Accuracy
Percentage
0
20
40
60
80
Train Positive
Train Negative
Test Positive
Test Negative
Data