Sentiment Analysis on Tweets about products of Apple and Google

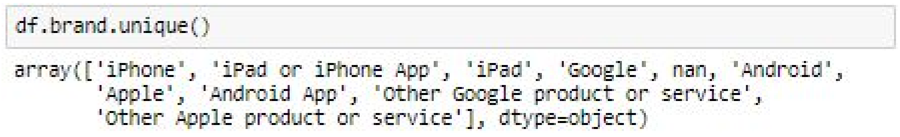
Project Description

The goal of this project is to build a model that can do sentiment analysis on Twitter messages. Sentiment analysis is a method for gauging attitude and/or opinions towards particular topics expressed in text. The analysis in this project, to be specific, is to determine whether the Twitter users' attitude towards products from two particular brands, Apple and Google, is positive or negative. Brands can use the results to monitor their reputation across social media. Furthermore, companies and brands can make improvements on products perfectly met customers' demands.

The data is retrieved from the website, Figure Eight. It provides many open datasets, which are free for everyone to download. The data downloaded as a csv file and it contains three columns -- tweets, brands tweets' emotions are directed at, and the emotions. After cleaning the data, I will compare two different methods, Bag of Words and TF-IDF to convert the tweets to numeric data so that it can be used to train the model. I will try two algorithms of classification after all what the model wants to achieve is to decide whether the tweet has a positive emotion or a negative one. The two algorithms are Multinomial Naive Bayes classification and Support Vector Machine classification. I also utilize the NLTK (Natural Language Toolkit) for word-stemming and word-lemmatization. The data set will be split into two parts based on different brands tweets are about. One will be used to build and tune the model. Another one will be used to test how generalized this model is.

Data Processing

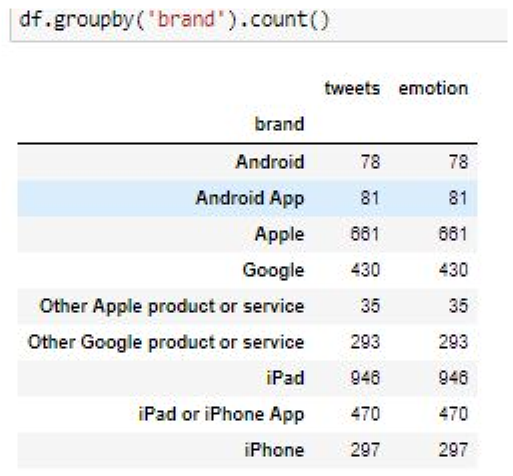
The data set has total 9093 rows and three columns, which are the Twitter messages, the target brands, and the opinions. The target brand column has 9 unique values. Take a closer look at those 9 unique values, I realize that it does not mean 9 unique brands rather two main brands and others:



```
df.brand.unique()
```

```
array(['iPhone', 'iPad or iPhone App', 'iPad', 'Google', nan, 'Android',
       'Apple', 'Android App', 'Other Google product or service',
       'Other Apple product or service'], dtype=object)
```

It is more clear see that most of the tweets are about products of service of brands, Google and Apple when the data is grouped by 'brand'. The data set will be partitioned into two parts based on the different target brands. For those tweets which have 'Android' and 'Android App' as brand, one can't tell which specific brand's products they are about, so they will be dropped. The model will be built using tweets about only Apple products. After the model is finalized, the data of Google products will be used to see whether the model can use on tweets about a different brand, in other words, to see how generalized the model is.



```
df.groupby('brand').count()
```

brand	tweets	emotion
Android	78	78
Android App	81	81
Apple	661	661
Google	430	430
Other Apple product or service	35	35
Other Google product or service	293	293
iPad	946	946
iPad or iPhone App	470	470
iPhone	297	297

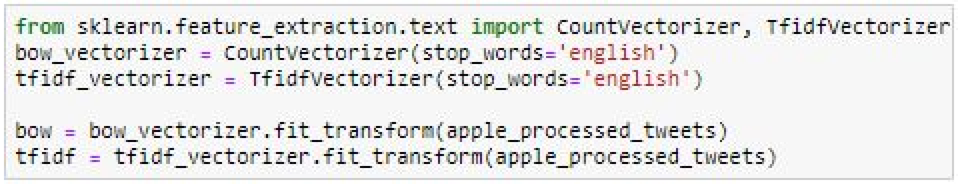
Moreover, this data set has four emotions: 'Negative emotion', 'Positive emotion', 'I can't tell', and 'No emotion toward brand or product'. Only two are the ones that matter in this project -- positive emotion and negative emotion. Thus, the samples with emotions other than positive or negative will be ignored.

Now, move to clean the tweets and convert them to be numerically representable. The URLs, Twitter usernames (@something), any numbers, any punctuations, and any special characters will be removed. After taking out the URLs and before taking out any punctuations, the contractions of words will be removed at first since the contractions can result in misinterpreting the meaning of a phrase, especially in the case of negations if the punctuation, apostrophe, is just simply being removed. And since the main goal to use the package, contractions, is to avoid the misinterpreting any negations. Thus, phrases, such as " year's ", are not going to be expanded. For more information on this package, please see here. The technique used in the removals to match the URLs, usernames and so on is the regular expression library. After cleaning, the tweet messages will be tokenized and then join with a space between words. Furthermore, I create two functions to normalize the tweets even more by stemming words or lemmatizating words. The goal of both stemming and lemmatization is to get the root forms of derived words, but they differ in their approaches. Stemming usually just chops ff the ends of words. Lemmatization uses the morphological analysis of words. Ronald Wahome claims that the above two techniques may not work so well because they essentially shorten words to their base words and the Twitter messages are short messages by design. I still would like to try the two methods and to see whether they would make a distinct difference on improving my model. There many different implementations for stemming and lemmatization in python, I use the LancasterStemmer and the WordNetLemmatizer from the NLTK library.

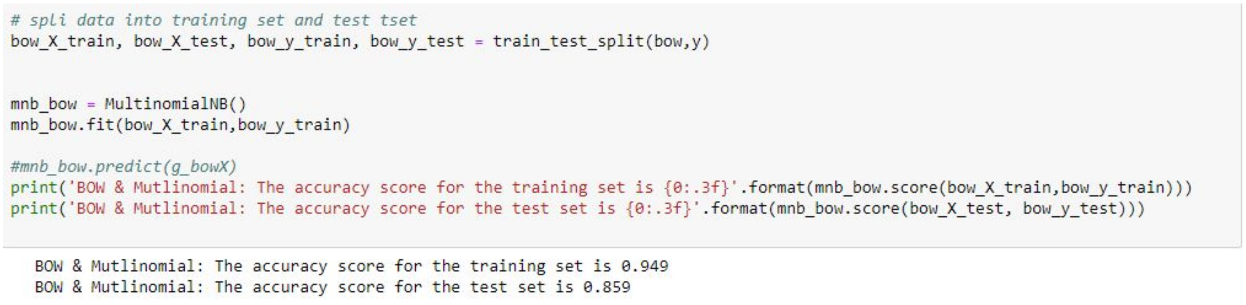
Last step in the data process section is to convert the cleaned tweets, which now are lists of words, to be numerically represented. I explore two techniques in this projects -- Bag-of-Words (BOW) and TF-IDF. Bag-of-Words approach is like to look at the counts of the words in each Twitter messages. The implementation, CountVectorizer, in sklearn this BOW approach. It counts the occurrences of words and builds a sparse matrix of the counts. TF-IDF is short for term Term Frequency - Inverse Document Frequency. It is the product of term frequency and inverse document frequency.The term frequency is simply words' counts. The inverse document frequency is a measure of how much information the word provides, i.e. if it is common or rare across all documents. TF-IDF is intended to reflect how important a word is to a document. In other words, if a word occurs in every tweets regardless the sentiment of the tweets, then this word give little information. Its TF-IDF score is 0 or very close to zero. The implementation in sklearn is called TfidVectorizer. These two methods will be compared during the model training process. Moreover, the target variable, emotion, needs converting to. It is transformed to a binary variable where 0 means negative emotion and 1 means positive emotion.

Train the Model

I will try two classification algorithms, the Multinomial Naive Bayes classification and the Support Vector Machine (SVM). Please recall that I have two approaches to convert the Twitter texts to numeric values, so that makes total four different combinations of text transformer and classifier.

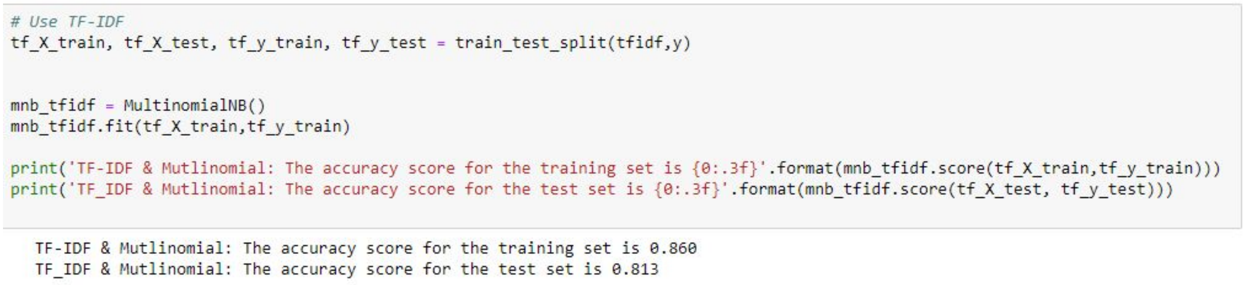


```
from sklearn.feature_extraction.text import CountVectorizer, TfidfVectorizer
bow_vectorizer = CountVectorizer(stop_words='english')
tfidf_vectorizer = TfidfVectorizer(stop_words='english')

bow = bow_vectorizer.fit_transform(apple_processed_tweets)
tfidf = tfidf_vectorizer.fit_transform(apple_processed_tweets)
```



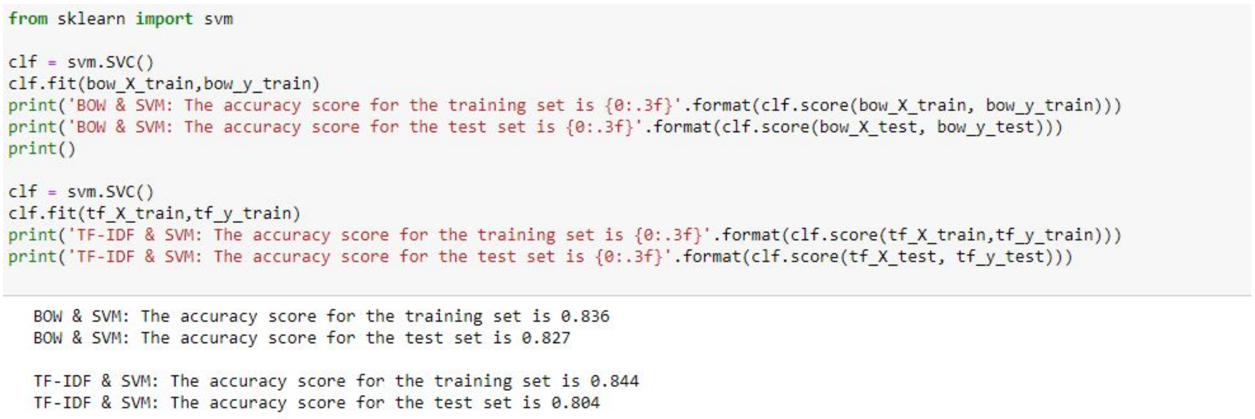
```
# spli data into training set and test tset
bow_X_train, bow_X_test, bow_y_train, bow_y_test = train_test_split(bow,y)


mnb_bow = MultinomialNB()
mnb_bow.fit(bow_X_train,bow_y_train)

#mnb_bow.predict(g_bowX)
print('BOW & Mutlinomial: The accuracy score for the training set is {0:.3f}'.format(mnb_bow.score(bow_X_train,bow_y_train)))
print('BOW & Mutlinomial: The accuracy score for the test set is {0:.3f}'.format(mnb_bow.score(bow_X_test, bow_y_test)))
```

```
BOW & Mutlinomial: The accuracy score for the training set is 0.949
BOW & Mutlinomial: The accuracy score for the test set is 0.859
```



```
# Use TF-IDF
tf_X_train, tf_X_test, tf_y_train, tf_y_test = train_test_split(tfidf,y)


mnb_tfidf = MultinomialNB()
mnb_tfidf.fit(tf_X_train,tf_y_train)

print('TF-IDF & Mutlinomial: The accuracy score for the training set is {0:.3f}'.format(mnb_tfidf.score(tf_X_train,tf_y_train)))
print('TF_IDF & Mutlinomial: The accuracy score for the test set is {0:.3f}'.format(mnb_tfidf.score(tf_X_test, tf_y_test)))
```

```
TF-IDF & Mutlinomial: The accuracy score for the training set is 0.860
TF_IDF & Mutlinomial: The accuracy score for the test set is 0.813
```



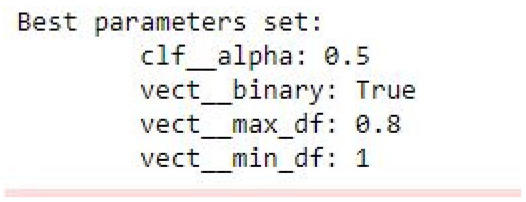
```
from sklearn import svm

clf = svm.SVC()
clf.fit(bow_X_train,bow_y_train)
print('BOW & SVM: The accuracy score for the training set is {0:.3f}'.format(clf.score(bow_X_train, bow_y_train)))
print('BOW & SVM: The accuracy score for the test set is {0:.3f}'.format(clf.score(bow_X_test, bow_y_test)))
print()

clf = svm.SVC()
clf.fit(tf_X_train,tf_y_train)
print('TF-IDF & SVM: The accuracy score for the training set is {0:.3f}'.format(clf.score(tf_X_train,tf_y_train)))
print('TF-IDF & SVM: The accuracy score for the test set is {0:.3f}'.format(clf.score(tf_X_test, tf_y_test)))
```

```
BOW & SVM: The accuracy score for the training set is 0.836
BOW & SVM: The accuracy score for the test set is 0.827

TF-IDF & SVM: The accuracy score for the training set is 0.844
TF-IDF & SVM: The accuracy score for the test set is 0.804
```

After comparing the accuracy score on training data and test data, the best combination is to convert the tweets by CountVectorizer and train the model with Multinomial Naive Bayes classifier.

Hyperparameter Tuning

Next, I do a hyperparameter tuning with a pipeline so that the transformation and the classification can be cross-validated together while setting different parameters. The tuning is about the maximum and minimum document frequency, terms have higher or lower frequency than which will be ignored when building the vocabulary during words converting. The additive smoothing in MultinomialNB is tuned as well. The technique used in the hyperparameter tuning is the GridSearchCV in sklearn. The set of best parameters given by the grid search is as shown below:



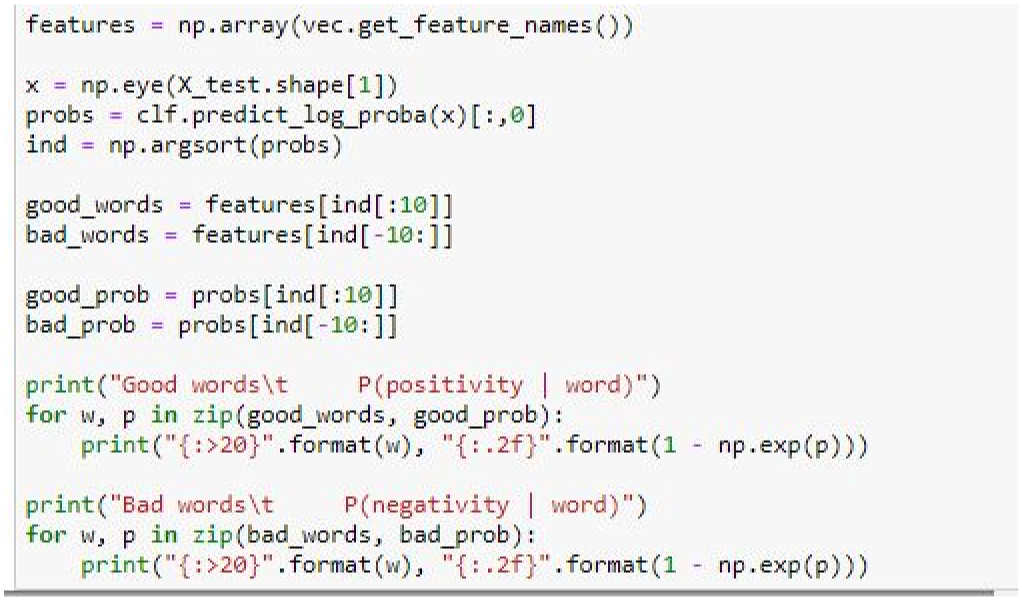
```
Best parameters set:
        clf__alpha: 0.5
        vect__binary: True
        vect__max_df: 0.8
        vect__min_df: 1
```

The best score is 0.867 on the training data set and 0.890 on the test data set. The test score is even a little bit better than the training'. It seems like a great result, but it is not really like that. I will explain why shortly.

Extract the Top 10 Strongest and Weakest Predictive Words

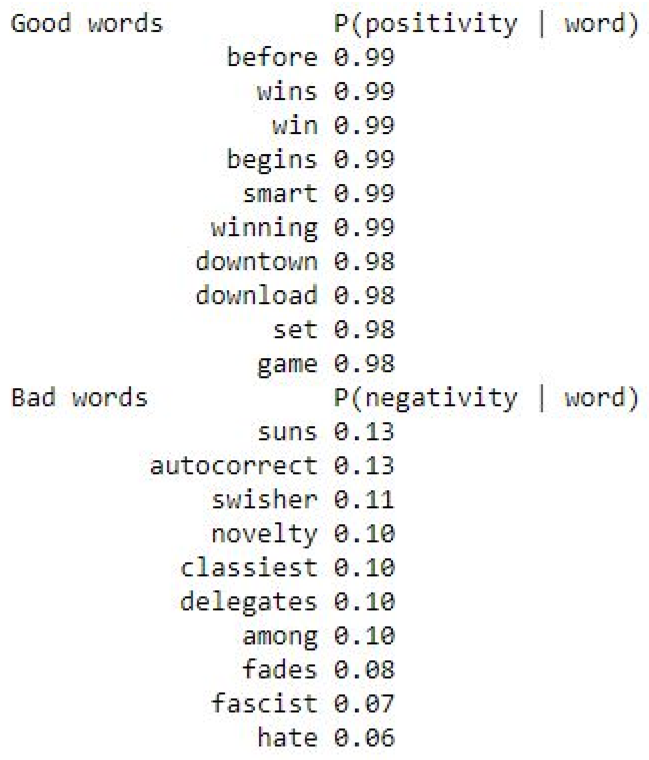
The best model is achieved after hyperparameter tuning done. I would like to extract the strongly and weakly predictive features. In other words, I want to see what words my model learns that indicate positivity the most and what words indicate negativity the most. I could not get the features directly from the pipeline after doing grid search.I did some research online about how to extract important features in text sentiment analysis, but all I could find is to extract them from the words converting matrix, in my case is the words count vector. Problem is that I couldn't get the vector with the best hyper-parameters from the pipeline. Thus, I have to do the words transformation and the classification all over again. Set the hyper-parameters to the best parameters obtained from the tuning process. Here comes one of the biggest issues I have in this project, the accuracy on the training data is 0.962 and on the test data is 0.842. Please recall the best scores for training and test data obtained from the tuning process are 0.863 and 0.875 respectively. I fail to figure why the scores on training set is this much different from test set when I break down the pipeline -- convert the words firstly and train the model with the best hyper-parameters. Despite of this inconsistency, the model still has a good score on the test data, so this model could achieve my initial goal.

Now come back to extract the top 10 strongest and weakest predictive words. I need to use the *get_feature_names* method from the CountVectorizer convector to get the actually words. An identity matrix is used to create a matrix that each row has exactly one word. Then, I use the trained MultinomialNB classifier to predict on this matrix. Finally, sort the rows by predicted probabilities, and pick the top and bottom 10 rows. Below is what my codes look like:



```
features = np.array(vec.get_feature_names())

x = np.eye(X_test.shape[1])
probs = clf.predict_log_proba(x)[:,0]
ind = np.argsort(probs)

good_words = features[ind[:10]]
bad_words = features[ind[-10:]]

good_prob = probs[ind[:10]]
bad_prob = probs[ind[-10:]]

print("Good words\t     P(positivity | word)")
for w, p in zip(good_words, good_prob):
    print("{:>20}".format(w), "{:.2f}".format(1 - np.exp(p)))

print("Bad words\t     P(negativity | word)")
for w, p in zip(bad_words, bad_prob):
    print("{:>20}".format(w), "{:.2f}".format(1 - np.exp(p)))
```

And below is what the top 10 most positive words and most negative words are:



```
Good words          P(positivity | word)
              before 0.99
                wins 0.99
                 win 0.99
              begins 0.99
               smart 0.99
             winning 0.99
            downtown 0.98
            download 0.98
                 set 0.98
                game 0.98
Bad words           P(negativity | word)
                suns 0.13
         autocorrect 0.13
             swisher 0.11
             novelty 0.10
           classiest 0.10
           delegates 0.10
               among 0.10
               fades 0.08
             fascist 0.07
                hate 0.06
```

The outcomes are not exactly as good as when looking at the top 10 positive words, anyone will agree that they would think these words mean positive or negative. For instance, set, before, and begins are listed in the top 10 strongest predictive words, but when I think of them, I don't really have any emotion towards them. Some words in the weakest predictive words make few sense, such as classiest, and novelty. These three words definitely indicate positiveness to me. I am not sure why my model would learn that they are the top 10 negative words, unless they are used sarcastically very often in my data set. Moreover, I notice that both win and wins appear in the top 10 strongest predictive words list, but clearly win and wins mean the same thing. Thus, I think I should try word stemming and lemmatization and see if that could help improve my model.

Try Word Stemming and Lemmatization

I use the two functions defined earlier to get the word stems as well as verb lemma and noun lemma as my data. Then I perform the transformation and the classification process on the stems and lemma as before. Here are the results:

```
vec = CountVectorizer(binary=grid_search.best_params_['vect__binary'],
                      min_df=grid_search.best_params_['vect__min_df'],
                      max_df=grid_search.best_params_['vect__max_df'])
stem_X = vec.fit_transform(stems)

X_train, X_test, y_train, y_test = train_test_split(stem_X,y,random_state=38)
clf = MultinomialNB(alpha=grid_search.best_params_['clf__alpha']).fit(X_train,y_train)

training_accuracy = clf.score(X_train, y_train)
test_accuracy = clf.score(X_test, y_test)

print("Accuracy on training data: {0:2f}".format(training_accuracy))
print("Accuracy on test data:     {0:2f}".format(test_accuracy))
```

```
Accuracy on training data: 0.955975
Accuracy on test data:     0.869640
```

```
vec = CountVectorizer(binary=grid_search.best_params_['vect__binary'],
                      min_df=grid_search.best_params_['vect__min_df'],
                      max_df=grid_search.best_params_['vect__max_df'])
verb_lemma_X = vec.fit_transform(verb_lemma)

X_train, X_test, y_train, y_test = train_test_split(verb_lemma_X,y,random_state=38)
clf = MultinomialNB(alpha=grid_search.best_params_['clf__alpha']).fit(X_train,y_train)

training_accuracy = clf.score(X_train, y_train)
test_accuracy = clf.score(X_test, y_test)

print("Accuracy on training data: {0:2f}".format(training_accuracy))
print("Accuracy on test data:     {0:2f}".format(test_accuracy))
```

```
Accuracy on training data: 0.959977
Accuracy on test data:     0.842196
```

```
vec = CountVectorizer(binary=grid_search.best_params_['vect__binary'],
                      min_df=grid_search.best_params_['vect__min_df'],
                      max_df=grid_search.best_params_['vect__max_df'])
noun_lemma_X = vec.fit_transform(noun_lemma)

X_train, X_test, y_train, y_test = train_test_split(noun_lemma_X,y,random_state=38)
clf = MultinomialNB(alpha=grid_search.best_params_['clf__alpha']).fit(X_train,y_train)

training_accuracy = clf.score(X_train, y_train)
test_accuracy = clf.score(X_test, y_test)

print("Accuracy on training data: {0:2f}".format(training_accuracy))
print("Accuracy on test data:     {0:2f}".format(test_accuracy))
```

```
Accuracy on training data: 0.965695
Accuracy on test data:     0.869640
```

The orders are accuracy score of word stems followed by score of verb lemma, which is followed by score of noun lemma. Below is a summary of the scores:

	Accuracy Score on	
	Training Data	Test Data
Word Stems	0.956	0.870
Verb Lemmas	0.960	0.842
Noun Lemmas	0.966	0.870

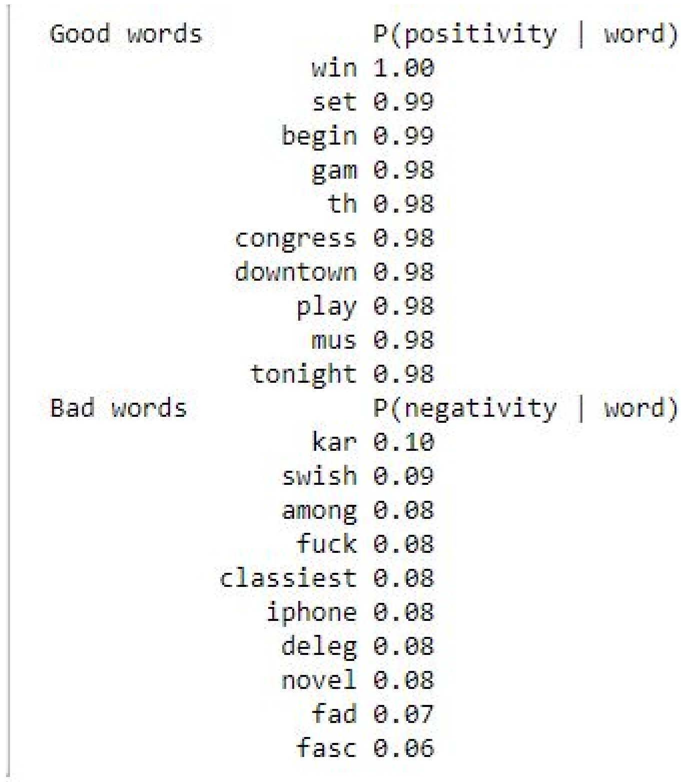
It looks like using word stems to train this MultinomialNB classifier performs the best on test data.

Please recall that at the very beginning, I split the original data set into two parts based on the two different targeted brands-- Google and Apple.And my goal is to build an as generalized as possible model such that can be used to predict tweets about many brands. By far, I have been using only the data about Apple products. Now I will compare the accuracy scores of using my four models to predict the tweets on Google products to make the final decision on which model is going to be my final model. The difference among those models are the data set used to train them. Here is a summary of the scores of the four models predicting on different data set:

	Accuracy Score on		
	Training Data	Test Data	Google Data
Words	0.969	0.825	0.754
Word Stems	0.956	0.870	0.761
Verb Lemmas	0.960	0.842	0.745
Noun Lemmas	0.966	0.870	0.735

It looks like the best model is the one trained with word stems as it has the best score on test data and Google data in multiple runs.

Let’s take a look at the top 10 strongest and weakest predictive words of this model:



```
Good words           P(positivity | word)
                 win 1.00
                 set 0.99
               begin 0.99
                 gam 0.98
                  th 0.98
            congress 0.98
            downtown 0.98
                play 0.98
                 mus 0.98
             tonight 0.98
Bad words            P(negativity | word)
                 kar 0.10
               swish 0.09
               among 0.08
                fuck 0.08
           classiest 0.08
              iphone 0.08
               deleg 0.08
               novel 0.08
                 fad 0.07
                fasc 0.06
```

Please note that there are some word stems in the above list do not seem like proper English words. That is what one may get when using the LancasterStemmer. LancasterStemmer tends to be heavy stemming, which leads stems to be non-linguistic. But it does give the best classification accuracy among my four models.

Conclusion and Thoughts

The classification model built with word stems data has a good accuracy score on test set and it can be considered as a generalized model since the accuracy score on a completely new data set, the tweets about Google products, is 0.787. My first goal of this project is achieved. However, I am not really sure whether this model is able to provide brands with readable feedback from Twitter users.

In other words, brands and companies do not only care about whether customers have good feeling at their products or they have bad experience with the products, but also, if not more, care about how to improve the products or what features they would like to keep. My intention is to build a model that predicts the tweets' sentiment and is able to provide meaningful key words by extracting the strongest and weakest predictive features. However, this model does not accomplish the second goal. Part of the reason is using the LancasterStemmer algorithm, which tends to overstem and returns non-linguistic 'words'. Another struggle is that I cannot figure out why the best accuracy score on test data is almost always as good as the score on training set while doing the grid search cross validation on the pipeline. On the other hand, the score on test data is often about 0.1 less than the score on the training data when I redo the words converting process and classification with the best hyper-parameters. However, like I mentioned earlier, despite of this inconsistency, the accuracy scores of test data and Google data are still good and I will say my model accomplished one of my goals -- to predict the sentiment of tweets on products for brands and companies.