US Airlines Sentiment Classification

05.12.2022

—

Suneel Kumar Thrinath
Natural Language Processing
Dr. Prof. Vahid

Git hub: Repositoryhttps://github.com/suneelkumar13/NLP_US_Airlines_Sentiment_Project_BERT_DISTILLBERT

Contents

1. Introduction

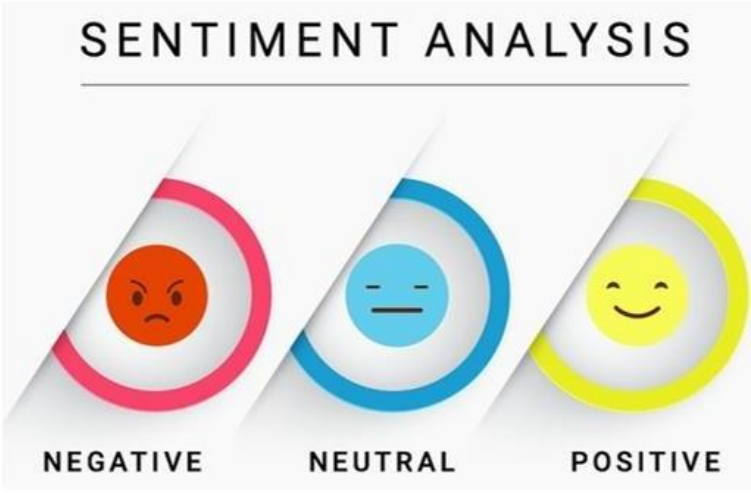


Figure 1: Sentiments or Emotions

Sentiment analysis is a text analysis technique that detects polarity (e.g., a positive or negative opinion) within text, whether a whole document, paragraph, sentence, or clause. Sentiment analysis is also known as opinion mining.

Understanding people's emotions is essential for businesses since customers express their thoughts and feelings more openly than ever before. The commonly used social media platform to express one's opinions or emotions is Twitter.

Performing analysis on customer feedback, such as opinions in survey responses and social media conversations, allows brands to listen attentively to their customers, and tailor products and services to meet their needs. However, all the opiniated data from the Twitter is in the form of text which is unstructured.

This unstructured data is hard to analyze, understand, sort through, and also time-consuming and expensive. Sentiment analysis, however, helps businesses make sense of all this unstructured text by automatically understanding, processing, and tagging it.

Objective of this project is to perform sentiment analysis on the tweets of six US Airlines. The scrapped tweets contain positive, negative, or neutral sentiments about the airline from their respective customers. The task is to analyze how travelers in February 2015 expressed their feelings on Twitter about six major US airlines.

Few of the algorithms used for sentiment analysis are Naive Bayes, SVM, Logistic Regression and LSTM. Out of them, in this project **Naïve Bayes classifier** is used to build the sentiment analysis model for the US Airline Tweets. The classifier is hard coded in Python without using any libraries with inbuilt classifiers. The project is coded and executed using Google Collaboratory.

2. Dataset Details

The dataset is borrowed from Kaggle, Twitter US Airline Sentiment.

This data originally came from Crowdflower's Data for Everyone library.

Number of Instances: 14640

Number of Class: 3 (positive, negative, neutral)

Number of Attributes: 15

The attributes in the dataset are listed below:

1. tweet_id
2. airline_sentiment
3. airline_sentiment_confidence
4. negativereason
5. negativereason_confidence
6. airline
7. airline_sentiment_gold
8. name
9. negativereason_gold
10. retweet_count
11. text
12. tweet_coord
13. tweet_created
14. tweet_location
15. user_timezone

The data is available in the csv file, 'Tweets.csv'. Most of the attributes of the dataset are irrelevant to the sentiment analysis of US Airlines. So, only the relevant attributes are considered for the analysis.

3. Approach

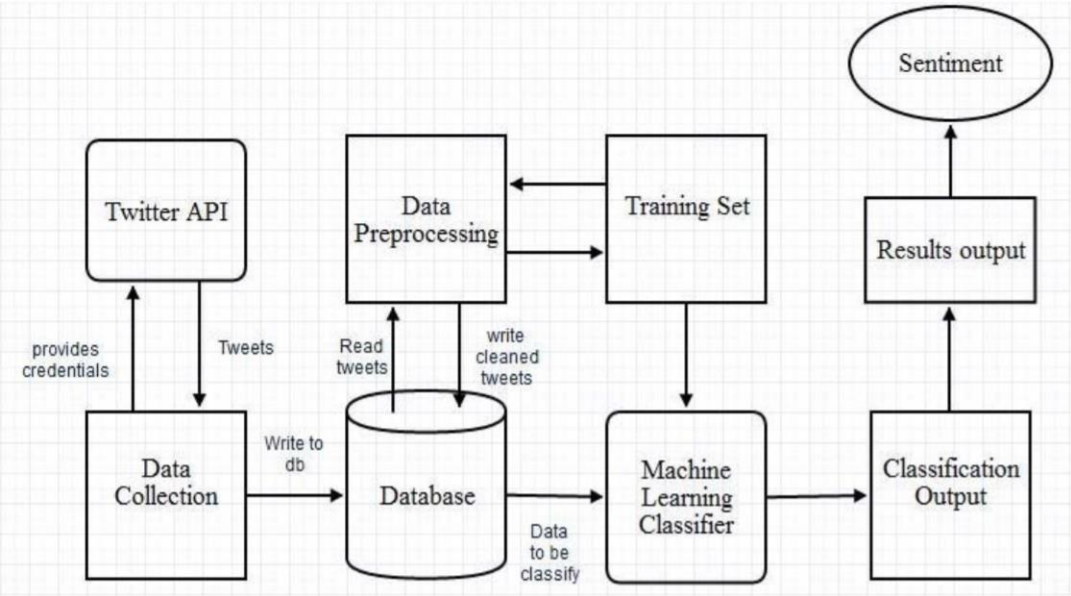


Figure 2: Sentiment Analysis Model Flow

In this project, the following steps are implemented to build a Naïve Bayes Tweet Classifier model to predict the sentiment of the tweets:

1. Importing Libraries & Loading Data
2. Data Preprocessing
3. Splitting the Data for Classification
4. Text Preprocessing
5. Building Dictionary / Vocabulary
6. Building the Naïve Bayes Tweet Classifier Model
7. Model Training & Evaluation
8. Trying to Improve the Model Performance

Working & results of each of these steps are elucidated in detail along with the required code snippets and other details.

3.1. Importing Libraries & Loading Data

Initially, all the basic necessary libraries like Pandas, Numpy, String, train_test_spit etc., are imported into Jupyter Notebook. The machine learning classifier is coded in pure Python without using any inbuilt library models. If any other libraries are required in the future, they can be imported accordingly.

The ~15k samples of data in the CSV file are loaded into Pandas dataframe using read_csv () function. This function returns the data in the CSV file as a two-dimensional data structure with labeled axes, called dataframe as shown below:

Out[3]:

	tweet_id	airline_sentiment	airline_sentiment_confidence	negativereason	negativereason_confidence	airline	airline_sentiment_gold	na
0	570306133677760513	neutral	1.0000	NaN	NaN	Virgin America	NaN	cair
1	570301130888122368	positive	0.3486	NaN	0.0000	Virgin America	NaN	jnard
2	570301083672813571	neutral	0.6837	NaN	NaN	Virgin America	NaN	yvonnal
3	570301031407624196	negative	1.0000	Bad Flight	0.7033	Virgin America	NaN	jnard
4	570300817074462722	negative	1.0000	Can't Tell	1.0000	Virgin America	NaN	jnard

Figure 3: Dataframe showing few samples of data from CSV file

The shape of the created dataframe is shown below:

```
1 data_frame.shape
```

```
(14640, 15)
```

Figure 4: Dataframe Shape

3.2. Data Preprocessing

In the data preprocessing step, the sentiment values in the data which are categorical are modified to numerical values. The categorical values in the data are 'neutral', 'positive' & 'negative'. For the easy computation, we are replacing the 'neutral', 'positive' & 'negative' with 0, 1 & 2 values, respectively.

The code for the above modification is shown below:

```
#repalcing the categorical values of 'airline_sentiment' to numeric values
data_frame['airline_sentiment'].replace(('neutral', 'positive', 'negative'), (0, 1, 2), inplace=True)
data_frame['airline_sentiment'].value_counts()
```

```
2    9178
0    3099
1    2363
Name: airline_sentiment, dtype: int64
```

Figure 5: Replacing the sentiment values with numeric

From the above execution, we can see that there are 3099 samples that are with neutral sentiment, 2363 samples with positive sentiment and 9178samples with negative sentiment. It is clear that the training data has more negative opinion or sentiments of US Airlines customers.

3.3. Splitting the Data for Classification

Before getting into data splitting to train and test data, we have to create feature (X) and target (y) variables from the dataset. The US Airline Sentiment dataset has so many features. Among them, in our project we will be working with the 'text' & 'airline_segment' features. Here, the 'text' is considered as a feature (X) and the 'airline_segment' as a target label (y) for the classifier. The code snippet of this task is as follows:

```
#forming the feature & label variables
data = data_frame['text'].values.tolist()
labels = data_frame['airline_sentiment'].values.tolist()
```

```
#First five samples text
data[:5]
```

```
['@VirginAmerica What @dhepburn said.',
 "@VirginAmerica plus you've added commercials to the experience... tacky.",
 "@VirginAmerica I didn't today... Must mean I need to take another trip!",
 '@VirginAmerica it\'s really aggressive to blast obnoxious "entertainment" in your guests\' faces &amp;amp; they have little recourse',
 "@VirginAmerica and it's a really big bad thing about it"]
```

```
#first 5 samples label
labels[:5]
```

```
['neutral', 'positive', 'neutral', 'negative', 'negative']
```

Figure 6: Forming features and label variables

After forming the feature (X) and target (y) variables, the entire dataset needs to be split into train and test datasets. We train our model on the train data and test the accuracy of built model prediction or classification on the test data. By splitting the dataset, we are not doing any changes to the test data which gives unaltered results of our model efficiency.

The dataset splitting is done by using predefined 'train_test_split ()' function from scikit-learn as shown below. And the dataset is split into 80% of train data and 20% of test data, basically an 80-20 split.

```
#splitting the data into 80 and 20 split
train_X, test_X, y_train, y_test = train_test_split(data, labels, test_size=0.2,
                                                    random_state=42, shuffle=True)

print(f'Number of training examples: {len(train_X)}')
print(f'Number of testing examples: {len(test_X)}')
```

```
Number of training examples: 11712
Number of testing examples: 2928
```

Figure 7: Splitting the dataset

The input and target variables of train data are denoted by train_X & train_y and for test data, test_X & test_y, respectively. After splitting, the training dataset has 11712 samples and the test dataset has 2928 samples. In the next step is text processing.

3.4. Text Preprocessing

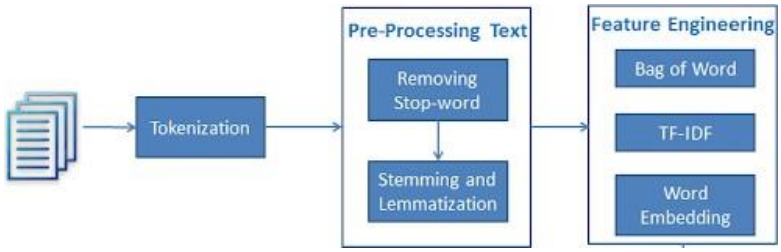


Figure 8: Text Processing Steps

A process of transforming text into something an algorithm can digest is text processing. This includes:

- tokenizing the data
- removing the punctuation
- removing the stopwords
- stemming
- lemmatization

As of now, we are only going to tokenize the data and work without removing the punctuation or stop words and apply any other text processing methods. Tokenization is a way of separating a piece of text into smaller units called tokens. Here, tokens can be either words, characters, or subwords.

```
#funtion for tokenizing the data
""" Tokenize sentence with specific pattern
Arguments: text {str} -- sentence to be tokenized, such as "I love NLP"
Keyword Arguments: pattern {str} -- reg-expression pattern for tokenizer (default: {default_pattern})
Returns: list -- list of tokenized words, such as ['I', 'love', 'nlp'] """
def tokenize(text, pattern = default_pattern):

  text = text.lower()
  return regexp_tokenize(text, pattern)
```

The nltk package has built in tokenization function. But here we are not using any of the libraries. So, for the tokenization function, we are using, a default predefined pattern. The code snippet for the tokenization is shown below:

Figure 9: Tokenization function

Few of the tokenized samples are shown below:

```
['@', 'united', 'you', 'are', 'offering', 'us', '8', 'rooms', 'for', '32', 'people', 'fail']
['@', 'united', 'pushing', '2', 'hours', 'on', 'hold', '.', 'priceless', '.', 'http', ':', 't', '.', 'co', 'ths10ldy2a']
['@', 'southwestair', "you're", 'my', 'early', 'frontrunner', 'for', 'best', 'airline', 'oscars2016']
['@', 'americanair', 'flight', '3056', 'still', 'sitting', 'at', 'dfw', 'waiting', 'for', 'baggage', 'to', 'be', 'loaded']
```

Figure 10: Tokenized Samples

3.5. Building Dictionary / Vocabulary

In this step, we are building a vocabulary based on training corpus. The dictionary omits grammar and word order but has the number of occurrences of words within the text of training data. The code snippet for creating dictionary is as follows:

```
#building dictionary
def createDictionary(data):
  """ Function: To create a dictionary of tokens from the data
  Arguments: data in the type - list
  Returns: Sorted dictionary of the tokens and their count in the data """

  dictionary = dict()
  for sample in  data:
    for token in sample:
      dictionary[token] = dictionary.get(token, 0) + 1
  #sorting the dictionary based on the values
  sorted_dict = sorted(dictionary.items(), key=lambda x: x[1], reverse=True)
  return dict(sorted_dict)
```

Figure 11: Function to create Dictionary

And the top 10 tokens of the dictionary are shown below:

```
Top 10 tokens in the training dictionary:

[('@', 13290),
 ('.', 12534),
 ('to', 6858),
 ('the', 4856),
 ('i', 4385),
 ('?', 3729),
 ('a', 3619),
 (',', 3354),
 ('united', 3338),
 ('you', 3284)]
```

Figure 12: Top 10 tokens of the Vocabulary

3.6. Building the Naïve Bayes Tweet Classifier Model

Naïve Bayes classification model is based on Bayes rule. It relies on a very simple representation of the document (called the bag of words representation). It is a Bayesian classifier that makes a simplifying (naive) assumption about how the features interact.

Naive Bayes is a probabilistic classifier, meaning that for a document d, out of all classes c ∈ C the classifier returns the class ˆc which has the maximum posterior ˆ probability given the document. The hat notation ˆ in the below formula means "our estimate of the correct class".

$$\hat{c} = \underset{c \in C}{\operatorname{argmax}} P(c|d)$$

Here, we are considering maximum likelihood estimation. For the document prior P(c) we ask what percentage of the documents in our training set are in each class c. Let N_c be the number of documents in our training data with class c and N_{doc} be the total number of documents. Then, prior can be computed as:

$$\hat{P}(c) = \frac{N_c}{N_{doc}}$$

To learn the probability P (w_i |c), which can be computed as the fraction of times the word w_i appears among all words in all documents of topic c. We first concatenate all documents with category c into one big "category c" text. Then we use the frequency of w_i in this concatenated document to give a maximum likelihood estimate of the probability:

$$\hat{P}(w_i|c) = \frac{count(w_i, c)}{\sum_{w \in V} count(w, c)}$$

Here the vocabulary V consists of the union of all the word types in all classes, not just the words in one class c.

There is a problem, however, with maximum likelihood training. If a word, w_i does not exists in class c of training corpus then the count of it will be 0. This results in P(w_i|c) = 0. But since naive Bayes naively multiplies all the feature likelihoods together, zero probabilities in the likelihood term for any class will cause the probability of the class to be zero.

The simplest solution is the add-one (Laplace) smoothing. While Laplace smoothing is usually replaced by more sophisticated smoothing algorithms in language modeling, it is commonly

used in naive Bayes text categorization:

$$\hat{P}(w_i|c) = \frac{count(w_i,c)+1}{\sum_{w\in V}(count(w,c)+1)} = \frac{count(w_i,c)+1}{\left(\sum_{w\in V} count(w,c)\right)+|V|}$$

Note once again that it is crucial that the vocabulary V consists of the union of all the word types in all classes, not just the words in one class c.

If a word occurs in our test data but is not in our vocabulary at all because they did not occur in any training document in any class then the solution for such **unknown words** is to ignore them—remove them from the test document and not include any probability for them at all.

Now, we defined a text classifier class called **NBClassifier**, which comprises of four functions:

- createDictionary ()
- fit ()
- predict ()
- score ()

createDictionary (): This function takes in the tokenized text data and gives out the dictionary or the bag of words of the data.

fit (): This function has all the word counts required to calculate the Naïve Bayes Classifier probabilities and then fits the classifier on our training data.

predict (): The test data is inputted to this function which determines the sentiment label based of each tweet by using the word counts computed during the training process (from fit function). In this step, Laplace smoothing is applied while computing Naïve Bayes probabilities for the test data. And the unknown words are ignored while computing probabilities.

score (): Determine how many tweets are classified correctly and measures the performance of the model in terms of accuracy. This function returns number of correct predictions and accuracy of the model.

3.7. Model Training & Evaluation

The Naïve Bayes Classifier, NBClassifier class we coded in the previous step takes three arguments which are listed below:

- X_train: Features of training dataset
- y_train: Labels of training dataset
- size: Size of vocabulary to be used in the model.

All three arguments are needed for the model to work.

Before jumping into the model training and evaluating, I created a function to store the evaluation results of each trained model to a list. This step is done to make the comparison of the models on variants of data easy. The code for this is as follows:

```
# Creating holders to store the model performance results
attributes = []
corr = []
acc = []

#function to call for storing the results
def storeResults(attr, cor,ac):
  attributes.append(attr)
  corr.append(round(cor, 3))
  acc.append(round(ac, 3))
```

Figure 13: To Store the Model Results

After training the model and calculating the performance it, the above function is called to store the test data accuracy & number of correct predictions. Later this stored data is used to compare the model trained on different data variations.

Now, the tokenized training data is passed to the model and a fit () function is called to find the required counts of words in the corpus based on the class. Then the predict method is called with the tokenized test data which calculates the probabilities of each sample based on the class and predicts the class it belongs to. The code for all these method calls are shown below:

```
#training the classifier
nb = NBClassifier(X_train, y_train, 'full')
nb.fit()

#predicting the labels for test samples
y_pred = nb.predict(X_test)

#Checking
print("NBClassifier Model miss any prediction???", len(X_test) != len(y_pred))
```

```
Model Start Time: 14:35:04
Model End Time: 14:35:58
NBClassifier Model miss any prediction??? False
```

Figure 14: NBClassifier Model Training

Now, we have predicted labels for the test data. So, to determine the model performance, score method is called with predicted and actual labels. This gives count of correct predictions by the model and accuracy of the model. The code snippet for this is shown below:

```
#Performance of the classifier
cor1, acc1 = nb.score(y_pred, y_test)
print("Count of Correct Predictions:", cor1)
print("Accuracy of the model: %i / %i = %.4f " %(cor1, len(y_pred), acc1))
```

```
Count of Correct Predictions: 2291
Accuracy of the model: 2291 / 2928 = 0.7824
```

Figure 15: NBClassifier Model Evaluation / Performance

Analysis: The Naïve Bayer's Classifier that we trained on the tokenized data predicts 78.24% of samples correctly i.e., 2291 samples out of 2928 samples of test data. This is the baseline performance that needs improvement.

After this, the following function call is executed to store the model results:

```
#storing the results. The below mentioned order of parameter passing is important.
#Caution: Execute only once to avoid duplications.
storeResults('Unprocessed Data', cor1, acc1)
```

Figure 16: Storing Model Evaluation / Performance

Now, to improve this number few more text processing methods are applied on the training data and then the classifier is trained on this modified data to predict the sentiment of the test samples.

3.8. Trying to Improve the Model Performance

So far, we have trained our model with the tokenized data and the model gave a performance of 78.24%. To improve the performance of the NBClassifier, we are going to do the following modification to our training & testing datasets:

1. apply other text processing methods
2. reduce the size of dictionary

3.8.1. Further Processing Text Data

In this step, we are going to apply two other text processing methods on the previously tokenized data. They are as follows:

- remove the punctuation
- remove stop words

3.8.1.1. Remove Punctuation

In this step, from the previously tokenized data we are removing any punctuation from the tokens and also punctuation tokens. Python provides a constant called string.punctuation that provides a great list of punctuation characters. The string constant is as follows:

```
1 #string of punctiations
2 string.punctuation
```

```
'!"#$%&\'()*+,-./:;<=>?@[\\]^_`{|}~'
```

Figure 17: String of Punctuation

With the help of this string constant, we will check each character of the tokens are remove the ones that are in the constant. And the function that performs this task is shown below:

```
1 #Removing the punctuation
2 '''Function: Removes the punctuation from the tokens
3    Arguments: list of text data samples
4    Returns: list of tokens of each sample without punctuation '''
5 def removePunctuation(data):
6     update = []
7     for sample in data:
8         #removing punctuation from the tokens
9         re_punct = [''.join(char for char in word if char not in string.punctuation) for word in sample]
10        #removes the empty strings
11        re_punct = [word for word in re_punct if word]
12
13        update.append(re_punct)
14    return update
```

Figure 18: Function that removes Punctuation from tokens

Now that we have the function that removes the punctuation from tokens, tokenized training & test data sets are passed into the function. The code for this task and the ending result of this is as follows:

```
#Removing punctuation from training data text tokens
X_train_P = removePunctuation(X_train)

#Removing punctuation from testing data text tokens
X_test_P = removePunctuation(X_test)

#train & test data after removing punctuation
print(X_train_P[0])
print(X_test_P[0])
```

```
['united', 'you', 'are', 'offering', 'us', '8', 'rooms', 'for', '32', 'people', 'fail']
['southwestair', 'youre', 'my', 'early', 'frontrunner', 'for', 'best', 'airline', 'oscars2016']
```

Figure 19: Removing Punctuation from datasets

Here we can see that the tokenized data displayed has no punctuation marks when compared to the tokens shown in Figure 10.

The modified data is then trained on the NBClassifier model to compute the model performance. The results of this execution are shown below:

```
#Performance of the classifier
cor2, acc2 = nb_punct.score(y_pred_P, y_test)
print("Count of Correct Predictions:", cor2)
print("Accuracy of the model: %i / %i = %.4f " %(cor2, len(y_pred_P), acc2))
```

```
Count of Correct Predictions: 2285
Accuracy of the model: 2285 / 2928 = 0.7804
```

Figure 20: NBClassifier Model Evaluation / Performance after Removing Punctuation from datasets

Analysis: The Naïve Bayer's Classifier that we trained on the data after removing the punctuation predicts 78.04% of samples correctly i.e., 2285 samples out of 2928 samples of test data. It is clear that the model performance of only tokenized data is better than the performance after removing the punctuation from tokens.

There are few tokens in the training corpus that are only punctuation marks like "@", ". ", ", "etc., In fact, it is evident from Figure 12 that top 2 tokens are punctuation tokens. One of the reasoning for the high count of '@' is that it is most commonly used in the twitter texts.

3.8.1.2. Remove Specified StopWords

A stop word is a commonly used word (such as "the", "a", "an", "in") that a search engine has been programmed to ignore, both when indexing entries for searching and when retrieving them as the result of a search query. They are the English words which does not add much meaning to a sentence. They can safely be ignored without sacrificing the meaning of the sentence.

We would not want these words to take up space in our database or taking up valuable processing time. Few of the stopwords that I chose to remove from the training corpus are ['the', 'at','i', 'of', 'us', 'have', 'a', 'you', 'ours', 'themselves', 'that', 'this', 'be', 'is', 'for']. I picked few of the stopwords from the top k tokens and few from the last k tokens of the training corpus.

The Python code to remove the stop words from the training corpus tokens is as follows:

```
'''Function: Removes the stopwords from the tokens
   Arguments: list of text data samples
   Returns: list of tokens of each sample without punctuation '''
def removeStopWords(data):
    update = []
    stopwords = ['the', 'at','i', 'of', 'us', 'have', 'a', 'you','ours', 'themselves',
                 'that', 'this', 'be', 'is', 'for']
    for sample in data:
        #removing stopwords from tokenized data
        re_stop = [word for word in sample if word not in stopwords]

        update.append(re_stop)
    return update
```

Figure 21: Function that removes stopwords from tokens

Now that we have the function that removes the specified stopwords from tokens, tokenized training & test data sets are passed into the function. The code for this task and the ending result of this is as follows:

```
#Removing stopwords from training data text tokens
X_train_S = removeStopWords(X_train)

#Removing stopwords from testing data text tokens
X_test_S = removeStopWords(X_test)

#train & test data after removing stopwords
print(X_train_S[0])
print(X_test_S[0])
```

```
['@', 'united', 'are', 'offering', '8', 'rooms', '32', 'people', 'fail']
['@', 'southwestair', "you're", 'my', 'early', 'frontrunner', 'best', 'airline', 'oscars2016']
```

Figure 22: Removing Specific stopwords from datasets

Here we can see that the tokenized data displayed has no specified stop words when compared to the tokens shown in Figure 10.

The modified data is then trained on the NBClassifier model to compute the model performance. The results of this execution are shown below:

```
#Performance of the classifier
cor3, acc3 = nb_stop.score(y_pred_S, y_test)
print("Count of Correct Predictions:", cor3)
print("Accuracy of the model: %i / %i = %.4f " %(cor3, len(y_pred_S), acc3))
```

```
Count of Correct Predictions: 2300
Accuracy of the model: 2300 / 2928 = 0.7855
```

Figure 23: NBClassifier Model Evaluation / Performance after Removing Punctuation from datasets

Analysis: The Naïve Bayer's Classifier that we trained on the data after removing the punctuation predicts 78.55% of samples correctly i.e., 2300 samples out of 2928 samples of test data. It is clear that the model performance of data after removing specified stopwords is better than the performance of only tokenized data.

Various combinations of stopwords are specified in the function and the accuracy of the model is improved from 73% to 78.55%. On further trying the various combinations of stopwords, one can improve the model performance much higher.

3.8.1.3. Remove Both Punctuation & Specified StopWords

In this data modification, we are taking the data without punctuation and passing it into the remove stopwords function shown in Figure 21. The end result of this task is that the modified data has no punctuation marks and specified stopwords also. The code snippet for this is as follows:

```
#Removing stopwords from training data text tokens
X_train_PS = removeStopWords(X_train_P)

#Removing stopwords from testing data text tokens
X_test_PS = removeStopWords(X_test_P)

#train & test data after removing stopwords
print(X_train_PS[0])
print(X_test_PS[0])
```

```
['united', 'are', 'offering', '8', 'rooms', '32', 'people', 'fail']
['southwestair', 'youre', 'my', 'early', 'frontrunner', 'best', 'airline', 'oscars2016']
```

Figure 24: Removing Punctuation & Specific stopwords from datasets

Here we can see that the tokenized data displayed has no punctuation and specified stop words when compared to the tokens shown in Figure 10.

The modified data is then trained on the NBClassifier model to compute the model performance. The results of this execution are shown below:

```
#Performance of the classifier
cor4, acc4 = nb_PS.score(y_pred_PS, y_test)
print("Count of Correct Predictions:", cor4)
print("Accuracy of the model: %i / %i = %.4f " %(cor4, len(y_pred_PS), acc4))
```

```
Count of Correct Predictions: 2283
Accuracy of the model: 2283 / 2928 = 0.7797
```

Figure 25: NBClassifier Model Evaluation / Performance after Removing Punctuation & Stopwords from datasets

Analysis: The Naïve Bayer's Classifier that we trained on the data after removing the punctuation predicts 77.97% of samples correctly i.e., 2283 samples out of 2928 samples of test data. It is clear that the model performance of this modified data is less that previously computed model performances.

Even after trying various combinations of stopwords that are specified in the function, the accuracy of the model reduced to 72%. Removing the punctuation marks from the tokens is affecting the model performance.

3.8.2. Reducing the Dictionary Size

In this step to improve the model performance, we are reducing the size of training dictionary further by taking only top-k frequent word types that appear in it. Here, we vary the value of k and compare the model performance.

The total tokens in the training dictionary are as follows:

```
#total tokens in training dictionary
print('Total tokens in the dictionary:', len(bog))
```

```
Total tokens in the dictionary: 13606
```

Figure 26: Total tokens in the dictionary

Now, we are considering to the check the model performance by taking top **5000 tokens** and **10,000 tokens**. Every data modification done in 3.8.1 are also applied while training on these two k sized dictionary.

All together 4 models with data modifications are executed with considering top k tokens of training dictionary. They are:

1. Model with Unprocessed data (only tokenized)
2. Model with no punctuation data
3. Model with data without few specific stopwords
4. Model with no punctuation and specific stopwords data

The model performance of these combinations is tabulated and sorted based on the accuracy as shown below:

Data Modification	Correct Predictions	Model Accuracy
5k Tokens of Voab - Unprocessed Data	2332	0.796
10k Tokens of Voab - Unprocessed Data	2332	0.796
5k Tokens of Voab - Removed few Stopwords	2321	0.793
10k Tokens of Voab - Removed few Stopwords	2321	0.793
5k Tokens of Voab - No Punctuation Data	2309	0.789
5k Tokens of Voab - Removed both Punctuation &...	2296	0.784
10k Tokens of Voab - No Punctuation Data	2287	0.781

Figure 27: NBClassifier Model Evaluation / Performance

Analysis:

- On reducing the size of dictionary and with the unprocessed data, model performance increased from 78.2% to 79.6%.
- On comparing other accuracy to its respective model perform with modified data, it is clear that reducing the size of dictionary increased the model performance.

4. Model Performance Comparison

As we already stored the count of correct prediction, data modification and accuracy in the list, we can create a dataframe from these three lists for better visualization. The code snippet for this is as follows:

```
#creating dataframe
results = pd.DataFrame({ 'Model Attributes': attributes,
    'Correct Predictions': corr,
    'Model Accuracy': acc})
```

Figure 28: Creating dataframe

Now, that the dataframe is formed with the results, we are going to sort the content in the dataframe on accuracy of the model. And the resulting dataframe is as follows:

	Data Modification	Correct Predictions	Model Accuracy
4	5k Tokens of Voab - Unprocessed Data	2332	0.796
8	10k Tokens of Voab - Unprocessed Data	2332	0.796
6	5k Tokens of Voab - Removed few Stopwords	2321	0.793
10	10k Tokens of Voab - Removed few Stopwords	2321	0.793
5	5k Tokens of Voab - No Punctuation Data	2309	0.789
2	Removed few Stopwords	2300	0.786
7	5k Tokens of Voab - Removed both Punctuation &...	2296	0.784
11	10k Tokens of Voab - Removed both Punctuation ...	2293	0.783
0	Unprocessed Data	2291	0.782
9	10k Tokens of Voab - No Punctuation Data	2287	0.781
1	No Punctuation Data	2285	0.780
3	Removed both Punctuation & Few Stopwords	2283	0.780

Figure 29: DataFrame of model performance and data modification

Implementations along with this project

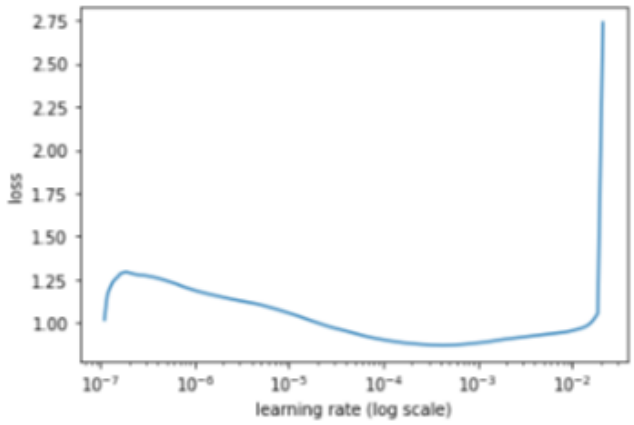
BERT :

BERT is designed to pre-train deep bidirectional representations from unlabeled text by jointly conditioning on both left and right context in all layers. As a result, the pre-trained BERT model can be fine-tuned with just one additional output layer to create state-of-the-art models for a wide range of tasks, such as question answering and language inference, without substantial task-specific architecture modifications.

```
learner.lr_find()
learner.lr_plot()
```

```
simulating training for different learning rates... this may take a few moments...
Epoch 1/1024
1221/1221 [==============================] - 1025s 820ms/step - loss: 1.2354 - accuracy: 0.5815
Epoch 2/1024
1221/1221 [==============================] - 15s 11ms/step - loss: 10.9507 - accuracy: 0.4444


done.
Please invoke the Learner.lr_plot() method to visually inspect the loss plot to help identify the
ciated with falling loss.
```



As opposed to directional models, which read the text input sequentially (left-to-right or right-to-left), the Transformer encoder reads the entire sequence of words at once. Therefore it is considered bidirectional, though it would be more accurate to say that it's non-directional. This characteristic allows the model to learn the context of a word based on all of its surroundings (left and right of the word).

```
learner.autofit(2e-5, 5)
```

```
begin training using triangular learning rate policy with max lr of 2e-05...
Epoch 1/5
1221/1221 [==============================] - 1028s 825ms/step - loss: 0.5765 - accuracy: 0.7671
Epoch 2/5
1221/1221 [==============================] - 1008s 826ms/step - loss: 0.3632 - accuracy: 0.8624
Epoch 3/5
1221/1221 [==============================] - 1009s 826ms/step - loss: 0.2324 - accuracy: 0.9215
Epoch 4/5
1221/1221 [==============================] - 1009s 826ms/step - loss: 0.1432 - accuracy: 0.9525
Epoch 5/5
1221/1221 [==============================] - 1011s 828ms/step - loss: 0.0911 - accuracy: 0.9712
```

```
<keras.callbacks.History at 0x7f9d98f62d90>
```

DistillBert:

As Transfer Learning from large-scale pre-trained models becomes more prevalent in Natural Language Processing (NLP), operating these large models in on-the-edge and/or under constrained computational training or inference budgets remains challenging. In this work, we propose a method to pre-train a smaller general-purpose language representation model, called DistilBERT, which can then be fine-tuned with good performances on a wide range of tasks like its larger counterparts.

```
trn_d, val_d, preproc_d = text.texts_from_array(x_train=X_train,
                                                y_train=y_train,
                                                x_test=X_test,
                                                y_test=y_test,
                                                class_names=['positive','negative','neutral'],
                                                preprocess_mode='distilbert',
                                                maxlen=350)
```

```
Downloading:   0%|          | 0.00/483 [00:00<?, ?B/s]

preprocessing train...
language: en
train sequence lengths:
        mean : 17
        95percentile : 27
        99percentile : 29
```

While most prior work investigated the use of distillation for building task-specific models, we leverage knowledge distillation during the pre-training phase and show that it is possible to reduce the size of a BERT model by 40%, while retaining 97% of its language understanding capabilities and being 60% faster.

```
learner.validate(val_data=val)
```

	precision	recall	f1-score	support
0	0.89	0.91	0.90	1026
1	0.75	0.71	0.73	455
2	0.81	0.81	0.81	350
accuracy			0.84	1831
macro avg	0.82	0.81	0.81	1831
weighted avg	0.84	0.84	0.84	1831

```
array([[936,  67,  23],
       [ 88, 323,  44],
       [ 29,  39, 282]])
```

To leverage the inductive biases learned by larger models during pre-training, we introduce a triple loss combining language modeling, distillation and cosine-distance losses.

```
learner_d.fit_onecycle(3e-5, 4)
```

```
begin training using onecycle policy with max lr of 3e-05...
Epoch 1/4
1221/1221 [==============================] - 329s 257ms/step - loss: 0.6410 - accuracy: 0.7350 - val_loss: 0.6085 - val_accu
racy: 0.7553
Epoch 2/4
1221/1221 [==============================] - 314s 256ms/step - loss: 0.4338 - accuracy: 0.8383 - val_loss: 0.4639 - val_accu
racy: 0.8247
Epoch 3/4
1221/1221 [==============================] - 313s 255ms/step - loss: 0.2670 - accuracy: 0.9033 - val_loss: 0.5061 - val_accu
racy: 0.8269
Epoch 4/4
1221/1221 [==============================] - 314s 256ms/step - loss: 0.1050 - accuracy: 0.9696 - val_loss: 0.5730 - val_accu
racy: 0.8378

<keras.callbacks.History at 0x7f9a44274090>
```

```
learner_d.validate(val_data = val_d)
```

	precision	recall	f1-score	support
0	0.89	0.91	0.90	1026
1	0.74	0.71	0.72	455
2	0.80	0.80	0.80	350
accuracy			0.84	1831
macro avg	0.81	0.80	0.81	1831
weighted avg	0.84	0.84	0.84	1831

```
array([[933,  70,  23],
       [ 89, 321,  45],
       [ 28,  42, 280]])
```

5. Model Performance Analysis

So far with all the data modifications, we executed the NBClassifier model for 12 times. So, we have 12 accuracies & count of correct predictions to compare and analyze. Below are few of the analysis points:

- NBClassifier model performance increased on reducing the size of training vocabulary irrespective of the applied data modifications to training and test datasets.

- Model execution with complete training vocabulary:
  - ✓ Model with no specific stopwords data has higher performance (78.6%) than the other three models with data variations.
  - ✓ This model is followed by the model with unprocessed data with performance of 78.2%
- Model execution with 5k training vocabulary tokens:
  - ✓ Model with unprocessed data has higher performance accuracy (79.6%).
  - ✓ This model is followed by model with no specific stopwords data with performance of 79.3%
- Model execution with 10k training vocabulary tokens:
  - ✓ Model with unprocessed data has higher performance accuracy (79.6%).
  - ✓ This model is followed by model with no specific stopwords data with performance of 79.3%
- Removing punctuation from the data tokens effects the model performance. This is observed in models with different size of training vocabulary.
- Removing specific stopwords increased the model performance when complete vocabulary is considered.
- All the model performance of 12 model executions are in the range of 78% to 80%.

6. Conclusion

The final take away from this project is to understand the computations in the Naïve Bayes Classifier. Code the text processing methods and the model in Python without using any libraries. Analyze the effect of text processing methods on the model performance. Understand the effect of reduced vocabulary size on the model performance.

7. References

[1] https://web.stanford.edu/~jurafsky/slp3/4.pdf
[2] https://monkeylearn.com/sentiment-analysis/
[3] https://machinelearningmastery.com/clean-text-machine-learning-python/
[4] https://www.tutorialspoint.com/python_text_processing/python_remove_stopwords.htm
[5] https://www.shanelynn.ie/select-pandas-dataframe-rows-and-columns-using-iloc-loc-and-ix/