IST 664 – NLP (Winterlude)

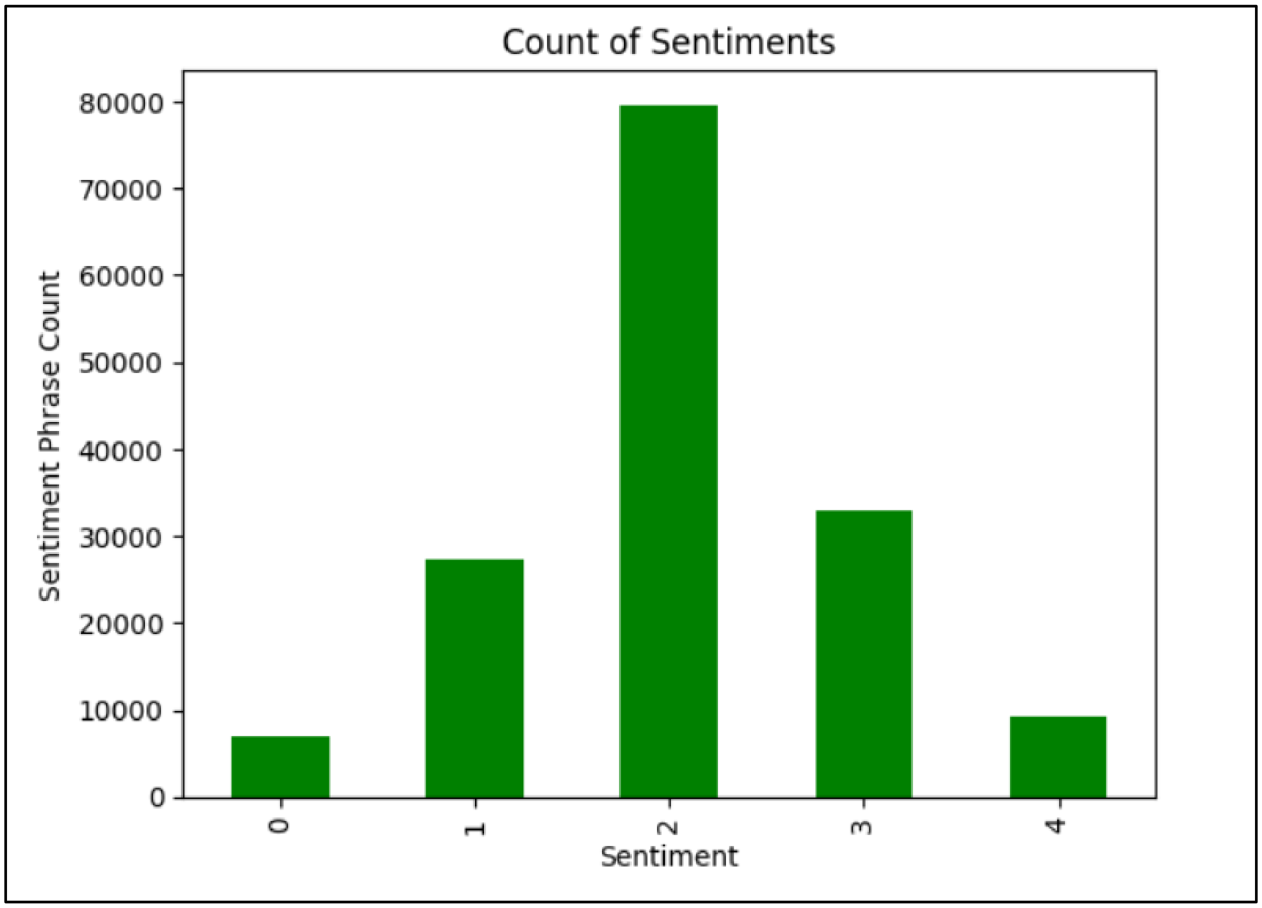
Final Project

Anish Kumar – 267542057

## Introduction

In this project we use the dataset that was produced for the Kaggle competition to build a classifier that can classify texts into different groups based on their sentiments. In an era dominated by digital communication, understanding the intricate nuances of human sentiment has become paramount. The sentiment analysis project endeavors to decode the vast realm of emotions embedded within textual data, unraveling the sentiments expressed by individuals for the movies through reviews. Through advanced natural language processing techniques, machine learning algorithms, and a keen eye for contextual analysis, this project aims to not only discern the positive, negative, or neutral tones in text but also delve deeper into the subtleties of emotions, providing valuable insights for people interested in watching a movie. By bridging the gap between human expression and computational analysis, the sentiment analysis project stands as a testament to the evolving synergy between technology and the intricacies of our shared human experience.

I started by reading the data from the corpus provided to us. I used train.tsv file to create dataframe in python. Then I did some Exploratory Data Analysis to check if there are any null values present in our dataset. I also created a visualization to check the distribution of different sentiments in our dataset. Below is the produced visualization: -



The sentiment labels are as follows: -

0 – Negative

1 – Somewhat negative

2 – Neutral

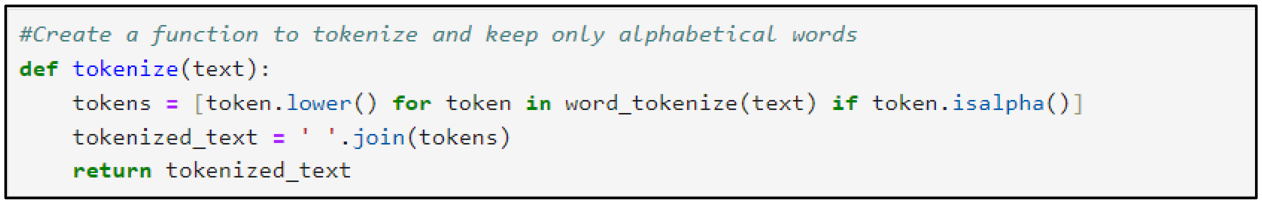
3 – Somewhat Positive

4 – Positive

Since the majority of the reviews are neutral, this dataset is highly imbalanced. So I am predicting that the models we will create will not perform really well because of this imbalance.

## Data Pre-Processing

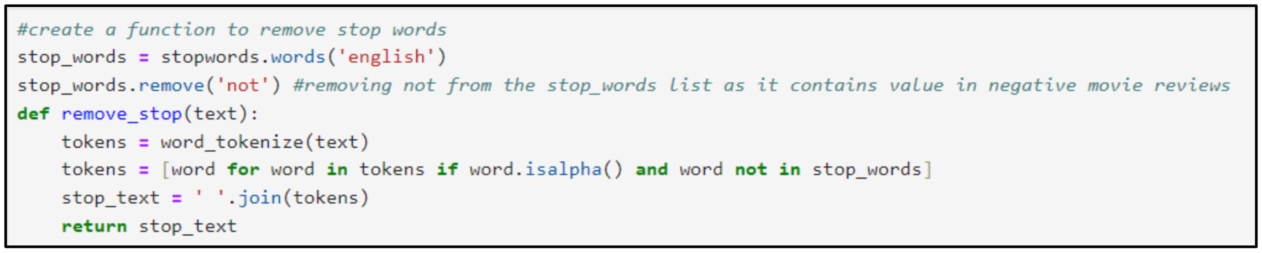
I started with pre-processing of data. I created a function called ‘tokenize’ to create tokens from the phrases in the dataset.



```python
#Create a function to tokenize and keep only alphabetical words
def tokenize(text):
    tokens = [token.lower() for token in word_tokenize(text) if token.isalpha()]
    tokenized_text = ' '.join(tokens)
    return tokenized_text
```

The NLTK has in-built functions for text tokenization, one of which is word_tokenize() which is being used here. The lower() function converts the tokens to lowercase and in the end the different tokens are joined together with space in between them to create a sentence of tokens.

Next, I created a function called ‘remove_stop’ to remove stop words from the tokens and to only contain alphabetical tokens.



```python
#create a function to remove stop words
stop_words = stopwords.words('english')
stop_words.remove('not') #removing not from the stop_words list as it contains value in negative movie reviews
def remove_stop(text):
    tokens = word_tokenize(text)
    tokens = [word for word in tokens if word.isalpha() and word not in stop_words]
    stop_text = ' '.join(tokens)
    return stop_text
```

NLTK has some in-built english stop words which are accessed by using stopwords.words(‘english’). I removed the ‘not’ stop word from the list of stop words as it contains value in negative movie reviews. The is.alpha() function checks if the word contains alphabets only. If it finds any character other than alphabets, it discards that word from the list of tokenized words. At last, after removing stop words all tokenized words are joined together with a space in between to create a sentence.

Next, I create a function called ‘wordnet_lemmatization’ to lemmatize the tokens.

Lemmatization and stemming are both techniques used in natural language processing to reduce words to their base or root form, aiding in text analysis and information retrieval. However, they differ in their approaches and outcomes.

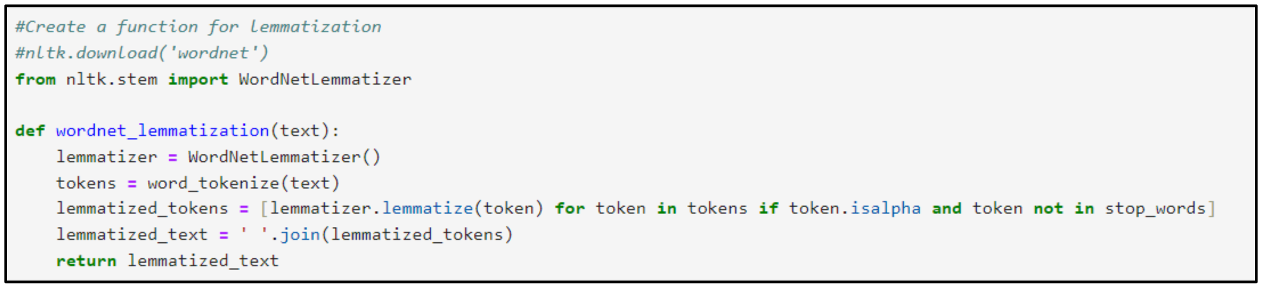
Stemming involves chopping off prefixes or suffixes from words to obtain the root form. This process might result in words that are not actual words but share a common root.

On the other hand, lemmatization focuses on reducing words to their base or dictionary form (lemma). This involves considering the context and part of speech of the word to ensure the transformed word is a valid one.

Stemming: If we apply stemming to both "better" and "best," we might end up with "bet" as the common root, which is not a valid English word.

Lemmatization: If we apply lemmatization, the words "better" and "best" would both be lemmatized to their base form, "good," which is a valid English word and reflects their common root in meaning.

This example illustrates how lemmatization retains valid words and considers the context and part of speech, while stemming might result in a root that is not linguistically correct or meaningful in all cases.

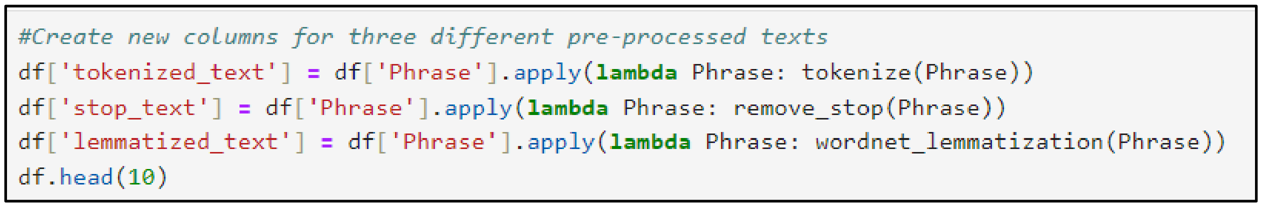


```python
#Create a function for lemmatization
#nltk.download('wordnet')
from nltk.stem import WordNetLemmatizer

def wordnet_lemmatization(text):
    lemmatizer = WordNetLemmatizer()
    tokens = word_tokenize(text)
    lemmatized_tokens = [lemmatizer.lemmatize(token) for token in tokens if token.isalpha and token not in stop_words]
    lemmatized_text = ' '.join(lemmatized_tokens)
    return lemmatized_text
```

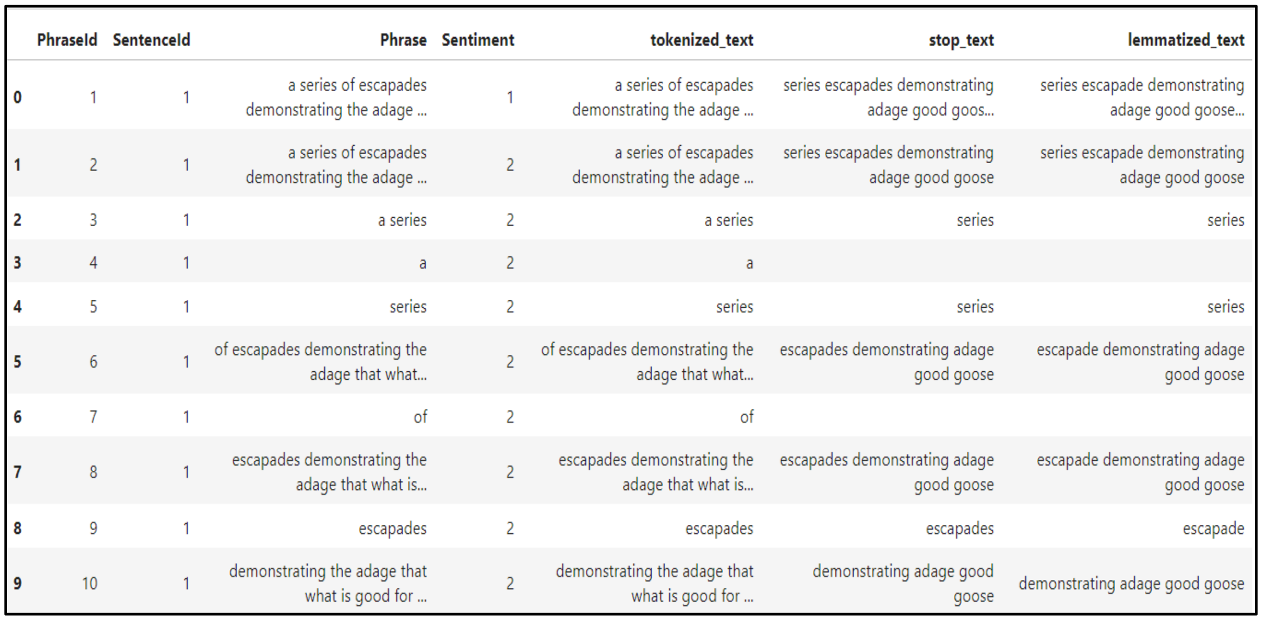
I used the NLTK in-built WordNetLemmatizer for this purpose.

Next, I created 3 new columns in my dataframe – tokenized_text, stop_text, and lemmatized_text – which contained tokens in a sentence form produced from the three different methodologies explained above.



```python
#Create new columns for three different pre-processed texts
df['tokenized_text'] = df['Phrase'].apply(lambda Phrase: tokenize(Phrase))
df['stop_text'] = df['Phrase'].apply(lambda Phrase: remove_stop(Phrase))
df['lemmatized_text'] = df['Phrase'].apply(lambda Phrase: wordnet_lemmatization(Phrase))
df.head(10)
```

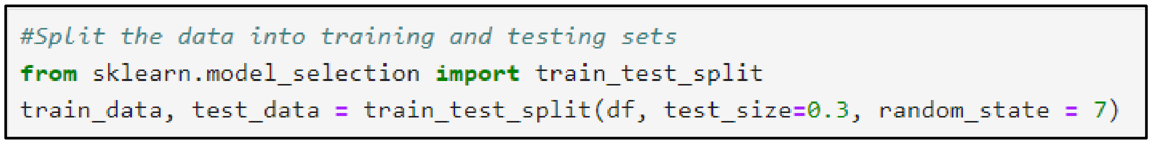
Now the dataset looks something like this.



| | PhraseId | SentenceId | Phrase | Sentiment | tokenized_text | stop_text | lemmatized_text |
|---|---|---|---|---|---|---|---|
| 0 | 1 | 1 | a series of escapades demonstrating the adage ... | 1 | a series of escapades demonstrating the adage ... | series escapades demonstrating adage good goos... | series escapade demonstrating adage good goose... |
| 1 | 2 | 1 | a series of escapades demonstrating the adage ... | 2 | a series of escapades demonstrating the adage ... | series escapades demonstrating adage good goose | series escapade demonstrating adage good goose |
| 2 | 3 | 1 | a series | 2 | a series | series | series |
| 3 | 4 | 1 | a | 2 | a | | |
| 4 | 5 | 1 | series | 2 | series | series | series |
| 5 | 6 | 1 | of escapades demonstrating the adage that what... | 2 | of escapades demonstrating the adage that what... | escapades demonstrating adage good goose | escapade demonstrating adage good goose |
| 6 | 7 | 1 | of | 2 | of | | |
| 7 | 8 | 1 | escapades demonstrating the adage that what is... | 2 | escapades demonstrating the adage that what is... | escapades demonstrating adage good goose | escapade demonstrating adage good goose |
| 8 | 9 | 1 | escapades | 2 | escapades | escapades | escapade |
| 9 | 10 | 1 | demonstrating the adage that what is good for ... | 2 | demonstrating the adage that what is good for ... | demonstrating adage good goose | demonstrating adage good goose |

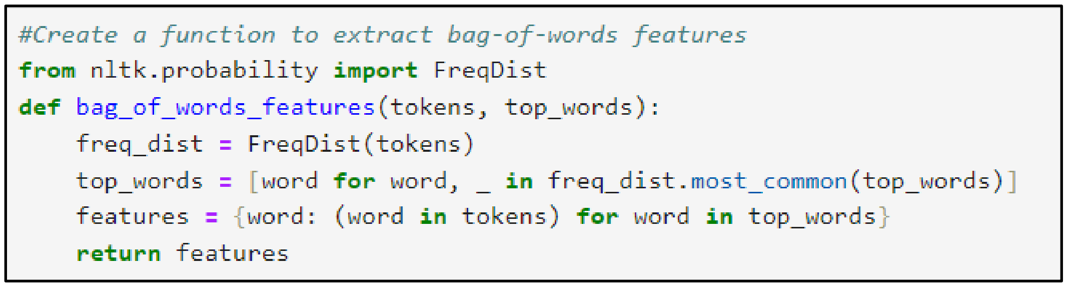
## Model Generation

I decided to build three different Naïve Bayes Classifier models from NLTK using the 3 newly created columns as independent variable and the Sentiment column as my target variable. I divided the dataset into training and testing sets in 70%-30% ratio. I will evaluate the models based on their Precision, Recall and F1 scores.



```
#Split the data into training and testing sets
from sklearn.model_selection import train_test_split
train_data, test_data = train_test_split(df, test_size=0.3, random_state = 7)
```

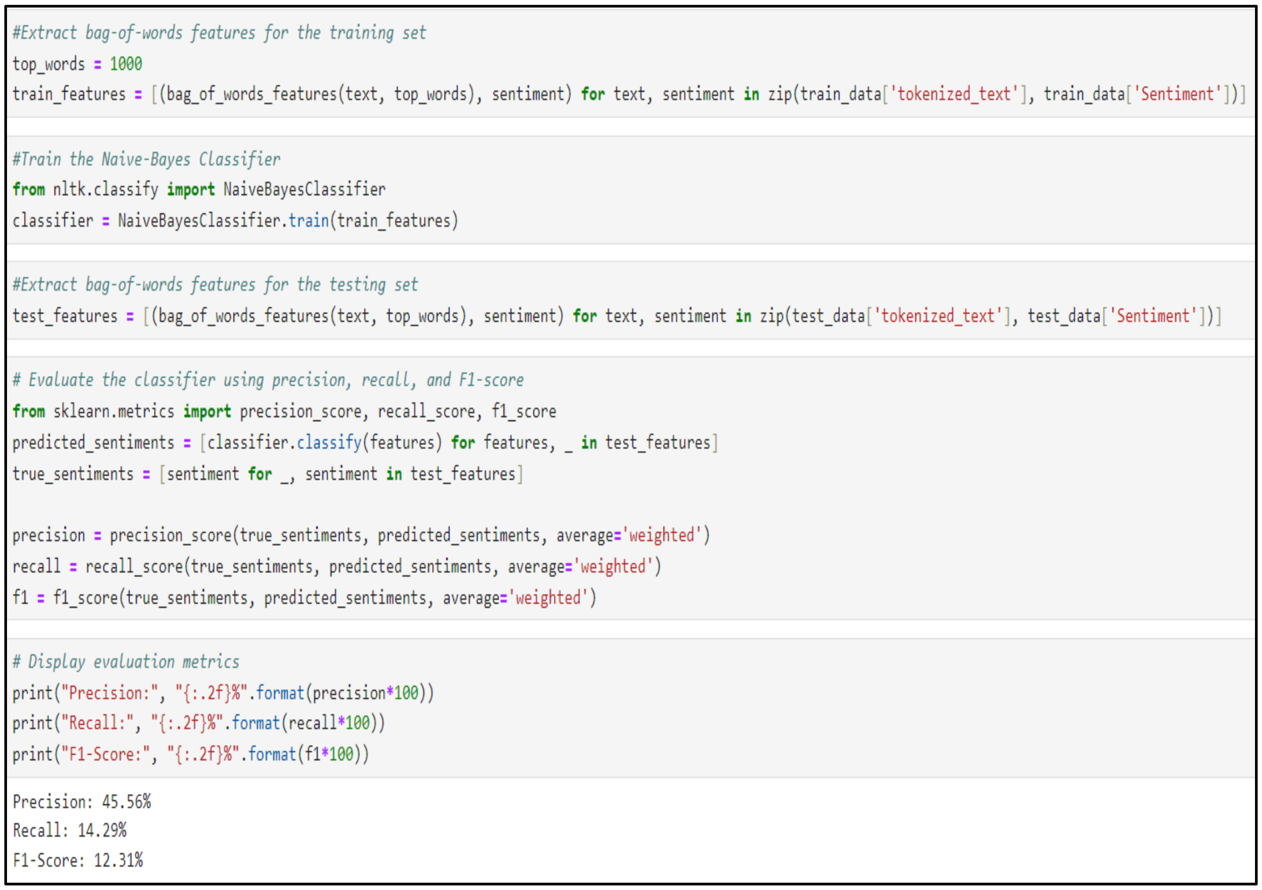
I created a function called 'bag_of_words_features' to extract bag-of-words features from the pre-processed texts.



```
#Create a function to extract bag-of-words features
from nltk.probability import FreqDist
def bag_of_words_features(tokens, top_words):
    freq_dist = FreqDist(tokens)
    top_words = [word for word, _ in freq_dist.most_common(top_words)]
    features = {word: (word in tokens) for word in top_words}
    return features
```

The FreqDist function from NLTK is used to compute the frequency distribution of a list of texts. It creates a frequency distribution object, which is essentially a dictionary where keys are the words and values are their corresponding frequencies in the given list.

## Model for Tokenized Texts



```python
#Extract bag-of-words features for the training set
top_words = 1000
train_features = [(bag_of_words_features(text, top_words), sentiment) for text, sentiment in zip(train_data['tokenized_text'], train_data['Sentiment'])]
```

```python
#Train the Naive-Bayes Classifier
from nltk.classify import NaiveBayesClassifier
classifier = NaiveBayesClassifier.train(train_features)
```

```python
#Extract bag-of-words features for the testing set
test_features = [(bag_of_words_features(text, top_words), sentiment) for text, sentiment in zip(test_data['tokenized_text'], test_data['Sentiment'])]
```

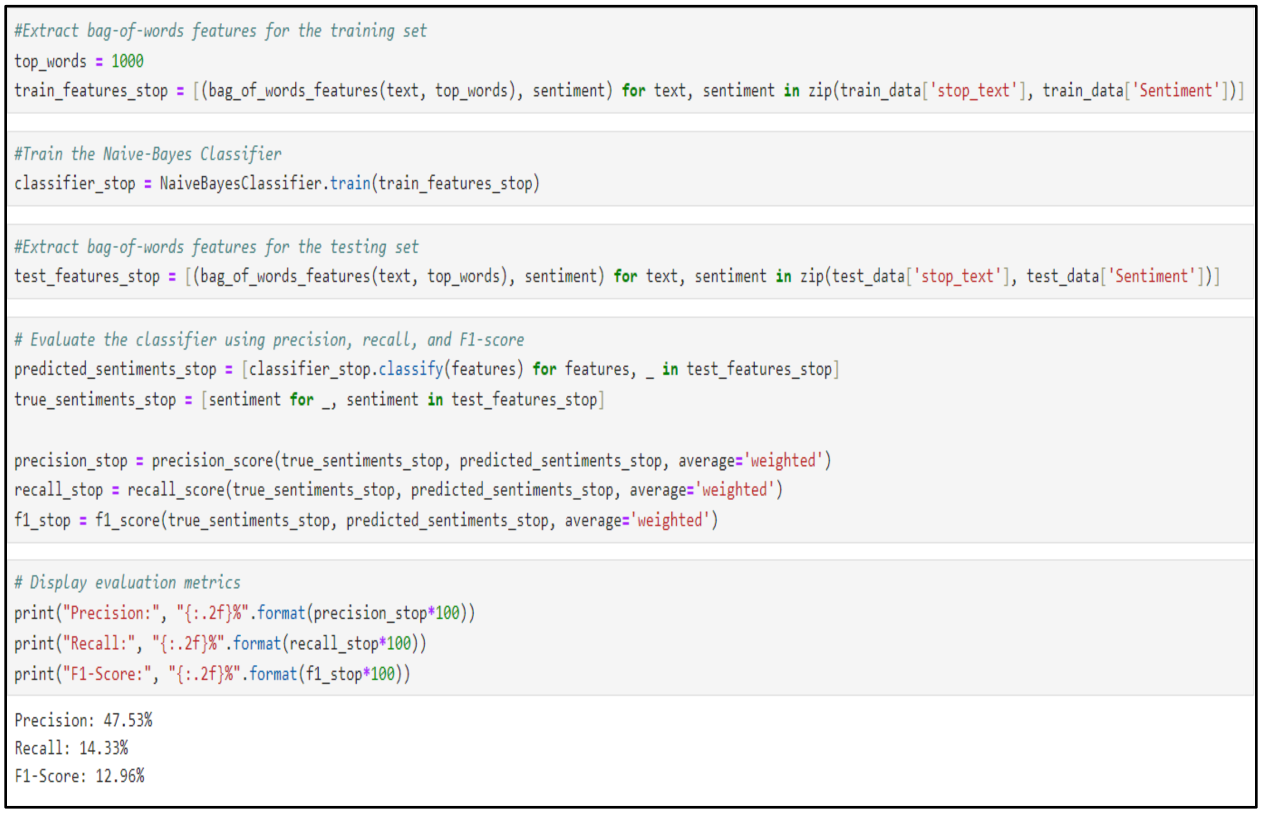
```python
# Evaluate the classifier using precision, recall, and F1-score
from sklearn.metrics import precision_score, recall_score, f1_score
predicted_sentiments = [classifier.classify(features) for features, _ in test_features]
true_sentiments = [sentiment for _, sentiment in test_features]

precision = precision_score(true_sentiments, predicted_sentiments, average='weighted')
recall = recall_score(true_sentiments, predicted_sentiments, average='weighted')
f1 = f1_score(true_sentiments, predicted_sentiments, average='weighted')
```

```python
# Display evaluation metrics
print("Precision:", "{:.2f}%".format(precision*100))
print("Recall:", "{:.2f}%".format(recall*100))
print("F1-Score:", "{:.2f}%".format(f1*100))
```

```
Precision: 45.56%
Recall: 14.29%
F1-Score: 12.31%
```

## Model after Removing Stop Words



```
#Extract bag-of-words features for the training set
top_words = 1000
train_features_stop = [(bag_of_words_features(text, top_words), sentiment) for text, sentiment in zip(train_data['stop_text'], train_data['Sentiment'])]
```

```
#Train the Naive-Bayes Classifier
classifier_stop = NaiveBayesClassifier.train(train_features_stop)
```

```
#Extract bag-of-words features for the testing set
test_features_stop = [(bag_of_words_features(text, top_words), sentiment) for text, sentiment in zip(test_data['stop_text'], test_data['Sentiment'])]
```

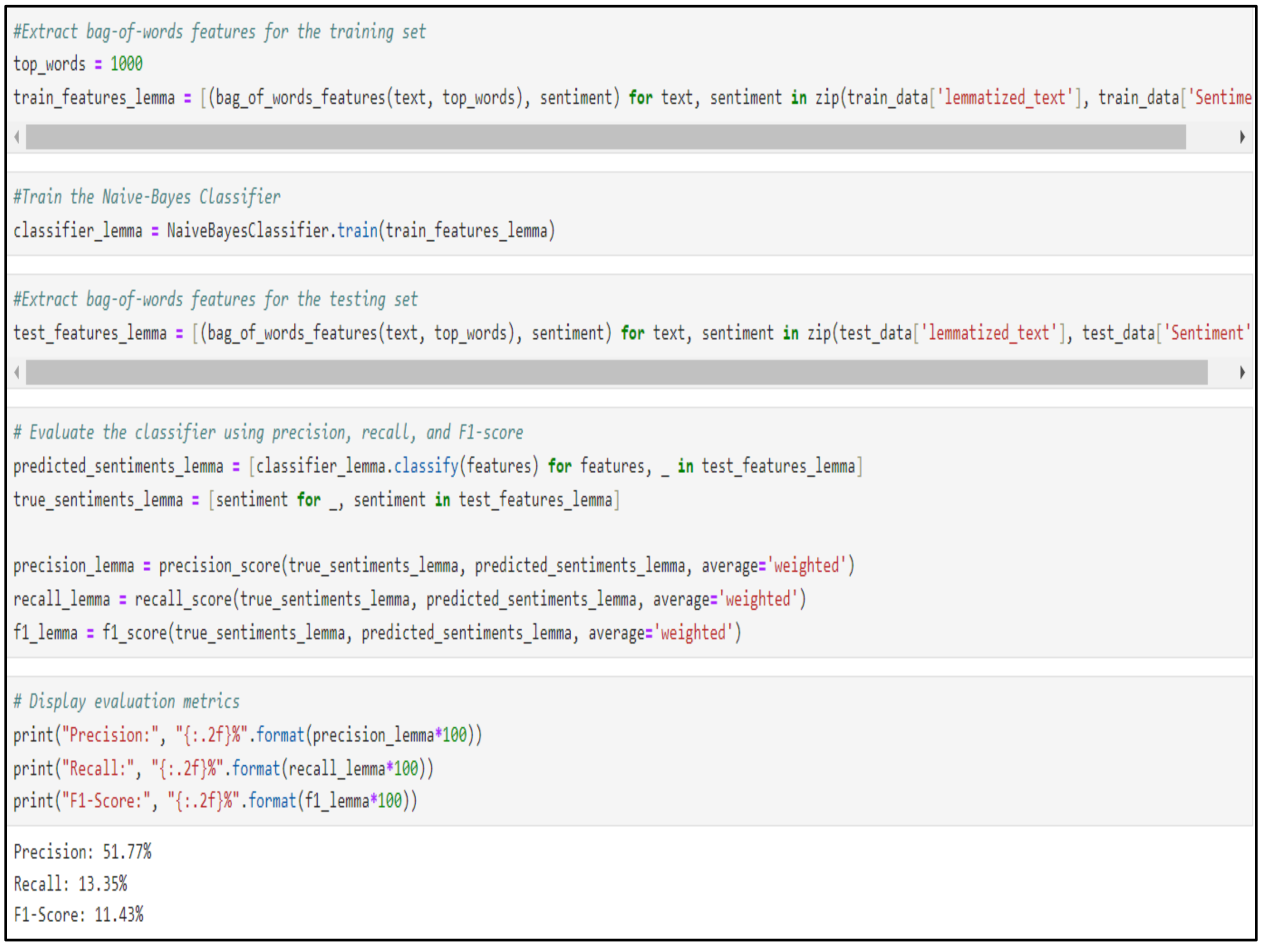
```
# Evaluate the classifier using precision, recall, and F1-score
predicted_sentiments_stop = [classifier_stop.classify(features) for features, _ in test_features_stop]
true_sentiments_stop = [sentiment for _, sentiment in test_features_stop]

precision_stop = precision_score(true_sentiments_stop, predicted_sentiments_stop, average='weighted')
recall_stop = recall_score(true_sentiments_stop, predicted_sentiments_stop, average='weighted')
f1_stop = f1_score(true_sentiments_stop, predicted_sentiments_stop, average='weighted')
```

```
# Display evaluation metrics
print("Precision:", "{:.2f}%".format(precision_stop*100))
print("Recall:", "{:.2f}%".format(recall_stop*100))
print("F1-Score:", "{:.2f}%".format(f1_stop*100))
```

```
Precision: 47.53%
Recall: 14.33%
F1-Score: 12.96%
```

## Model after Lemmatization



```
#Extract bag-of-words features for the training set
top_words = 1000
train_features_lemma = [(bag_of_words_features(text, top_words), sentiment) for text, sentiment in zip(train_data['lemmatized_text'], train_data['Sentime
```

```
#Train the Naive-Bayes Classifier
classifier_lemma = NaiveBayesClassifier.train(train_features_lemma)
```

```
#Extract bag-of-words features for the testing set
test_features_lemma = [(bag_of_words_features(text, top_words), sentiment) for text, sentiment in zip(test_data['lemmatized_text'], test_data['Sentiment'
```

```
# Evaluate the classifier using precision, recall, and F1-score
predicted_sentiments_lemma = [classifier_lemma.classify(features) for features, _ in test_features_lemma]
true_sentiments_lemma = [sentiment for _, sentiment in test_features_lemma]

precision_lemma = precision_score(true_sentiments_lemma, predicted_sentiments_lemma, average='weighted')
recall_lemma = recall_score(true_sentiments_lemma, predicted_sentiments_lemma, average='weighted')
f1_lemma = f1_score(true_sentiments_lemma, predicted_sentiments_lemma, average='weighted')
```

```
# Display evaluation metrics
print("Precision:", "{:.2f}%".format(precision_lemma*100))
print("Recall:", "{:.2f}%".format(recall_lemma*100))
print("F1-Score:", "{:.2f}%".format(f1_lemma*100))
```

```
Precision: 51.77%
Recall: 13.35%
F1-Score: 11.43%
```

After creating these three models, it was clear from the results that the second model i.e the model created using ‘stop_text’ as independent variable was the best out of the three as it had the highest F1 score (12.96%).

But still the F1 score is pretty low, and I would like to do some additional task to check if I can increase the model’s performance.

So, I decided to create a model using not just unigrams (bag-of-words) features but also using bigrams, POS tags and LIWC sentiment lexicon features all combined together. And I will use ‘stop_text’ as the independent column for the model because of the better performance than the other two.

## Model using LIWC, Unigram, Bigram and POS Tags Features

I created a function called 'liwc_sentiment_feature' to extract the LIWC sentiment lexicon features.

```python
# Function to count positive and negative words using LIWC lexicon
#nltk.download('vader_lexicon')
def liwc_sentiment_feature(text):
    sia = SentimentIntensityAnalyzer()
    sentiment_scores = sia.polarity_scores(text)
    return {'pos_words': sentiment_scores['pos'], 'neg_words': sentiment_scores['neg']}
```

I used the in-built SentimentIntensityAnalyzer function for this purpose.

Next, I created a function called 'combined_features' which extracted unigram, bigram and POS tag features and combined them all together.

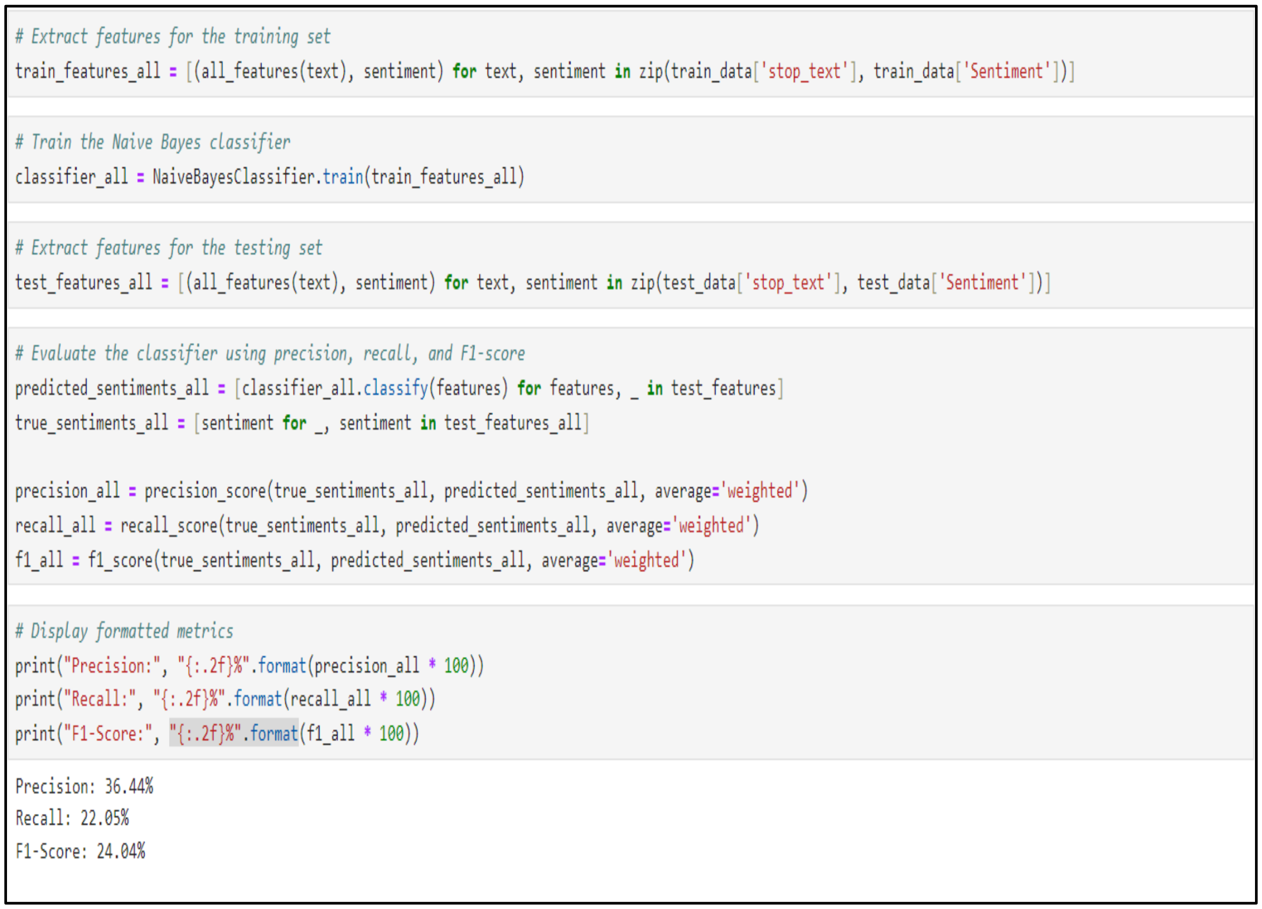
```python
# Function to extract unigram, bigram, and POS tag features
#nltk.download('averaged_perceptron_tagger')
def combined_features(text):
    tokens = word_tokenize(text)
    unigrams = bag_of_words_features(tokens, top_words=1000)
    bigrams = bag_of_words_features(list(nltk.bigrams(tokens)), top_words=500)
    pos_tags = FreqDist([tag for _, tag in pos_tag(tokens)])

    features = {**unigrams, **bigrams, **pos_tags}
    return features
```

After this I created another function called 'all_features' which combined the LIWC features with unigram, bigram and POS tags features.

```python
# Combine all features and LIWC sentiment feature
def all_features(text):
    liwc_features = liwc_sentiment_feature(text)
    text_features = combined_features(text)
    return {**text_features, **liwc_features}
```

After combining all the features I created the model.



```
# Extract features for the training set
train_features_all = [(all_features(text), sentiment) for text, sentiment in zip(train_data['stop_text'], train_data['Sentiment'])]

# Train the Naive Bayes classifier
classifier_all = NaiveBayesClassifier.train(train_features_all)

# Extract features for the testing set
test_features_all = [(all_features(text), sentiment) for text, sentiment in zip(test_data['stop_text'], test_data['Sentiment'])]

# Evaluate the classifier using precision, recall, and F1-score
predicted_sentiments_all = [classifier_all.classify(features) for features, _ in test_features]
true_sentiments_all = [sentiment for _, sentiment in test_features_all]

precision_all = precision_score(true_sentiments_all, predicted_sentiments_all, average='weighted')
recall_all = recall_score(true_sentiments_all, predicted_sentiments_all, average='weighted')
f1_all = f1_score(true_sentiments_all, predicted_sentiments_all, average='weighted')

# Display formatted metrics
print("Precision:", "{:.2f}%".format(precision_all * 100))
print("Recall:", "{:.2f}%".format(recall_all * 100))
print("F1-Score:", "{:.2f}%".format(f1_all * 100))
```

```
Precision: 36.44%
Recall: 22.05%
F1-Score: 24.04%
```

As it is clear from the results, this model outperforms all the previous models by a huge number. The F1 score for this model is 24.04%, which is almost the double of the model that was created using just bag-of-words (unigram) features.

## Conclusion

The sentiment classifier did not perform really well with just bag-of-words features. But as we combined them with more features like bigrams, POS Tags and LIWC lexicon the model's performance increased drastically.

There are quite a few other features that can be added as a future scope of development. One feature set can be a list of subjectivity lexicon.