Prototype Planning:

Introduction

The proliferation of social media platforms such as Twitter has brought about significant shifts in the ways in which individuals connect with one another, express their ideas, and discuss their personal experiences in today's globalised world. Twitter has developed into a significant platform for users to communicate their thoughts and information in real time. Every day, millions of tweets are sent on Twitter regarding a variety of topics, including politics, culture, products, and services. There is a beneficial storage of people's emotions and sentiments that they speak about on social media, which may be found in the massive quantity of data that is generated by social media. It is vital to understand these sentiments, regardless of whether they are good, neutral, or negative, since it is necessary to understand these feelings because it is beneficial for many things, such as keeping an eye on brands, undertaking market research, and analysing public opinion. Natural language processing (NLP), which Twitter mood analysis is a subset of, provides us with a dependable method for automatically categorising tweets in accordance with how they affect us emotionally.

Twitter is a wonderful resource for individuals, corporations, and researchers interested in understanding how people are feeling since tweets are so brief and take place in real time. The pattern has evolved into a digital representation of how people generally feel about the whole universe. The study of emotion on Twitter involves more than just tallying the amount of likes and shares generated by users' tweets. It also involves reading the whole of the language that is included in tweets in order to gain a sense of the intricate ideas and emotions that individuals are trying to convey. When it comes to managing a company, this entails gaining a grasp of the level of satisfaction experienced by consumers, recognizing any possible issues that may arise, and enhancing marketing techniques. It's possible that political campaigns may utilise sentiment analysis to find out how people feel about certain subjects and individuals. Researchers are able to investigate people's perspectives on significant societal problems that are relevant to everyone. Through categorising tweets as either positive, neutral, or negative, opinion research conducted on Twitter enables users to make more informed choices and better react to the feedback of the general public.

Discovering the sentiments of Twitter users on a topic might be challenging for a number of different reasons. Because each user is only allowed 280 characters to express themselves, they are forced to condense their thoughts and come up with inventive solutions. In order to do this, they often resort to the usage of slang, acronyms, and phrases. Because tweets are so particular to the context in which they are sent, it is essential to consider the broader context when attempting to evaluate someone's emotional state. This notion may be shown with the use of a furious tweet, which, depending on the response that follows it, can either be perceived as positive or negative. For instance, if you follow the tweet with the phrase "because my team won," it demonstrates that you are happy about the situation. On the other hand, if the next statement is "because of the traffic," it indicates that the speaker is in a gloomy mood. If you want your study on mood to be precise and comprehensive, you need also take into consideration the role that culture and social variables have in the phenomenon being studied.

Because of how hard it is, those who research and work with mood analysis on Twitter make use of machine learning models and natural language processing methods. The models were trained using the extensive amounts of tagged tweets that were available. The purpose of this research was to get an understanding of patterns and linguistic signals that may be utilised to infer how individuals are feeling. The use of strategies such as Support Vector Machines, Recurrent Neural Networks, and Transformer models such as BERT are some of the most often used approaches. As was just said, the job description for these models is for them to take in text and organise tweets into one of three categories: positive, neutral, or negative. The use of deep learning methods in conjunction with pre-trained word embeddings has resulted in significant improvements to the accuracy and dependability of mood analysis systems.

This introductory clause provides the framework for a more in-depth look at the studies done on Twitter about people's moods. In the sections that are to come, we are going to have a look at the approaches, tools, and procedures that were employed for researching moods on Twitter. This project's objective is to provide helpful insights into the many issues and possibilities that come up in this dynamic field of study. In the end, the purpose of our study is to provide you with a comprehensive perspective of the ways in which sentiment analysis may function as a potent instrument for gaining an understanding of the thoughts and emotions that people express on Twitter. In addition to this, one of our goals is to provide an explanation of how it may be used in a number of contexts so that individuals are able to make informed judgments and get valuable knowledge about the emotions of others.

Section 1: Prototype Identification and Planning

Section 1.1 Literature Review on Prototype Identification

Opinion mining and sentiment analysis are both terms that refer to an integral aspect of Natural Language Processing (NLP), which stands for Natural Language Processing. The extraction of emotional information from written material is the task assigned to it. It has a wide range of applications, some of which include the analysis of consumer feedback, monitoring social media, and doing market research. This review of the relevant literature illustrates how mood analysis is evolving through time by concentrating on significant methodologies, issues, and recent advancements in the field.

1. Contextual Information and History

At the beginning of the 21st century, scholars began investigating several approaches to classify written material as either positive, negative, or neutral. This is the beginning of what would become known as sentiment analysis (Turney, 2002). In earlier systems, a significant emphasis was placed on rule- and word-based procedures. Over the course of several years, sentiment analysis has seen significant development. According to Pang and Lee (2008), it currently makes use of machine learning, deep learning, and linguistic tools to improve its accuracy and reliability.

2. Analysing people's feelings with the use of machine learning

The development of effective algorithms for machine learning has been of critical importance in the field of mood analysis. Research conducted by Maas et al. (2011) and Go et al. (2009) shown that Support Vector Machines and Naive Bayes are effective methods for determining how individuals feel about a certain topic. Word embeddings became increasingly widespread as a result of the work of Mikolov et al. (2013) and Pennington et al. (2014). The way in which text data is represented has been enhanced as a result of this, which has led to improved mood analysis.

3. Advances in methods of "deep learning"
In recent years, deep learning models, such as Recurrent Neural Networks (RNNs) and Convolutional Neural Networks (CNNs), have performed very well when it comes to analysing people's moods (Socher et al., 2013). Models such as the Transformer-based BERT (Devlin et al., 2018) have established new benchmarks in the field of natural language processing (NLP). This model is able to comprehend context and achieves outstanding results in a wide variety of NLP tasks, including mood analysis.

4. Difficulties in comprehending the emotions of other individuals
Even if a lot of progress has been made in the field of mood studies, there are still some issues. Even with practice, dealing with humour, irony, and moods that shift based on the circumstances may be challenging. It has been attempted to find solutions to these issues by using contextual models and mood lexicons that are unique to the subject matter (Mohammad & Turney, 2013). Another issue is that there isn't a lot of data that can be recognized, particularly in languages except English (Cambria et al., 2017). This is a concern since it makes it difficult to do research. Transfer learning and bilingual models are two innovative approaches that have recently emerged as potential solutions to this issue.

5. Applications in everyday life
Researching people's feelings may be beneficial in a wide variety of contexts. It is used in the business world to monitor different brands and read feedback from customers. For instance, the airline industry utilises sentiment analysis to gauge the level of contentment felt by its clientele (Hutto & Gilbert, 2014). According to Bollen et al. (2011), doing sentiment analysis on social media platforms such as Twitter may assist politicians in better comprehending the feelings of the general public towards elections and other significant events.

6. Some Ethical Concerns to Reflect Upon
As mood analysis techniques grow more widespread, questions about their ethical implications have begun to surface. There has been a shift in attention on concerns pertaining to privacy, prejudice, and the judicial system. Researchers are putting in a lot of effort to make sure that the models they use for mood analysis are less biassed and that they safeguard the privacy of their users (Bishop, 2019; Dixon et al., 2018).

7. Preparations for the years to come
According to Cambria et al. (2013), there are a few challenges that need to be addressed before sentiment analysis may go further. These problems include context-aware sentiment

reading, fine-grained sentiment analysis, and dealing with multi-modal data. The development of Explainable AI (XAI) in the field of mood analysis is another intriguing topic of research. According to Ribeiro et al. (2016), the objective is to simplify the process of understanding and adhering to decisions made by AI.

In conclusion, mood analysis has advanced significantly since it first appeared. Rule-based approaches have given way to machine learning and deep learning models, which are far more powerful. Even while there has been progress, there are still challenges that need to be resolved, such as how to cope with complicated sentiments and moral dilemmas. As techniques for analysing moods continue to advance, they may one day be put to use to gain knowledge that is applicable to fields as diverse as business, politics, social media, and others.

Section 1.2 Reflection on the Prototype Identification

Conducting the literature review on mood analysis with a primary emphasis on AI-driven methodologies has been an experience that is not only entertaining but also enlightening. This idea encapsulates the most significant insights and comprehensions that came to me as a result of the research.

The research started out by highlighting how mood analysis is a constantly evolving field. Beginning with more straightforward lexical-based approaches, the area has been gradually shifting toward the use of machine learning and deep learning techniques. The transition away from rule-based mood categorization and toward more intricate models such as BERT and Convolutional Neural Networks demonstrates how difficult it is becoming to analyse text data. This modification demonstrates that the industry as a whole is committed to making things more precise and adaptable across a wide variety of domains.

A further finding that came as a surprise was the variety of applications that may be found for mood analysis. Sentiment analysis is being employed in a variety of fields, including politics, finance, and the monitoring of social media, in addition to its primary use, which is the examination of the comments made by customers. According to a research that related Twitter mood to stock market behaviour, sentiment analysis is a beneficial method for understanding public opinion, market trends, and even how the stock market behaves. This demonstrates how important sentiment analysis is as a technique to understand public opinion.

It was also abundantly evident that mood analysis had flaws, particularly with regard to its ability to comprehend complicated moods and circumstances. It is still challenging to differentiate between humorous and ironic situations while also taking into consideration language that is highly specialised to a certain topic or cultural milieu. We must continue our research and come up with fresh concepts if we are going to find solutions to these issues.

Concerns about ethics in artificial intelligence and the study of mood came up as a major concern. According to the available research, the challenges of prejudice, justice, and privacy need more study. Making certain that mood analysis models do not reinforce users' pre existing prejudices or infringe users' privacy will be a highly crucial aspect of future research and the development of applications.

Additionally, the research on the book shed light on several fascinating potential future directions for mood analysis. It is highly intriguing to add explainable artificial intelligence (XAI) to models used for mood monitoring since this kind of AI is designed to make models more transparent and straightforward.

This literature study, in its conclusion, shed light on how sentiment analysis has developed over time, its challenges, its usefulness, as well as the societal concerns that arise when utilising AI. It demonstrates how crucial it is for this industry to continue developing AI in a rational manner and coming up with new ideas on a consistent basis. Even though AI is revolutionising the way we interact with technology, sentiment analysis is still an essential tool for gaining insight into and making effective use of the information contained in text data.

Prototype Development

Section 2: Development

First of all Anaconda and Jupyter Notebook were installed. Afterwards a dataset was fetched from Kaggle that had the list of tweets and their emotion. The following the URL to the dataset:
https://www.kaggle.com/datasets/yasserh/twitter-tweets-sentiment-dataset
The following steps are listed below along the screenshot of the process.

1. ***Importing necessary libraries***

   This Python code is needed to do mood analysis, which is an important part of natural language processing (NLP). It starts by adding packages that have tools for manipulating data, showing data, and machine learning. The code probably has steps for preparing the data, like cleaning up the text and extracting features using the TF-IDF vectorization method. Then, it divides the data into training and testing sets so that the success of machine learning models can be checked. Support Vector Classifier (SVC), Bernoulli Naive Bayes, Decision Tree Classifier, Random Forest Classifier, and Logistic Regression are some of the important classifiers that it imports. These are needed for mood classification. The script also has features for evaluating models, like the accuracy, confusion matrix, and F1 score, which are very important for judging the quality of mood analysis results. The code makes it easier to create and test sentiment analysis models by giving a structure. This makes it useful for many uses, such as analysing customer comments and keeping an eye on social media sentiment.

```
In [1]: #Importing the Libraries
        import numpy as np
        import pandas as pd
        import seaborn as sns
        import matplotlib.pyplot as plt
        %matplotlib inline

        import nltk
        from sklearn.feature_extraction.text import TfidfVectorizer
        from sklearn.model_selection import train_test_split
        import string
        from tqdm import tqdm
        from multiprocessing import Pool
        from nltk.corpus import stopwords
        from nltk.stem.porter import PorterStemmer

        from sklearn.svm import SVC
        from sklearn.naive_bayes import BernoulliNB
        from sklearn.tree import DecisionTreeClassifier
        from sklearn.ensemble import RandomForestClassifier
        from sklearn.linear_model import LogisticRegression
        from sklearn.metrics import accuracy_score, confusion_matrix, classification_report, f1_score, \
        roc_auc_score, roc_curve, precision_score, recall_score

        import warnings
        warnings.filterwarnings('ignore')
```

2. *Importing the Dataset*

This code takes a dataset called "Tweets.csv" from a CSV file and puts it into a Pandas DataFrame named "df." The 'head()' method is then used to show the first few rows of the dataset, which lets you look at the first data items and how they are organised.

```
In [2]: # Loading the dataset
        df = pd.read_csv('Tweets.csv')
        #Let's check the samples of data
        df.head()

Out[2]:
        textID                                          text                 selected_text   sentiment
   0    cb774db0d1           I`d have responded, if I were going   I`d have responded, if I were going     neutral
   1    549e992a42     Sooo SAD I will miss you here in San Diego!!!                  Sooo SAD    negative
   2    088c60f138                          my boss is bullying me...              bullying me    negative
   3    9642c003ef                    what interview! leave me alone          leave me alone    negative
   4    358bd9e861   Sons of ****, why couldn`t they put them on t...               Sons of ****,    negative
```

3. *DataPreprocessing*

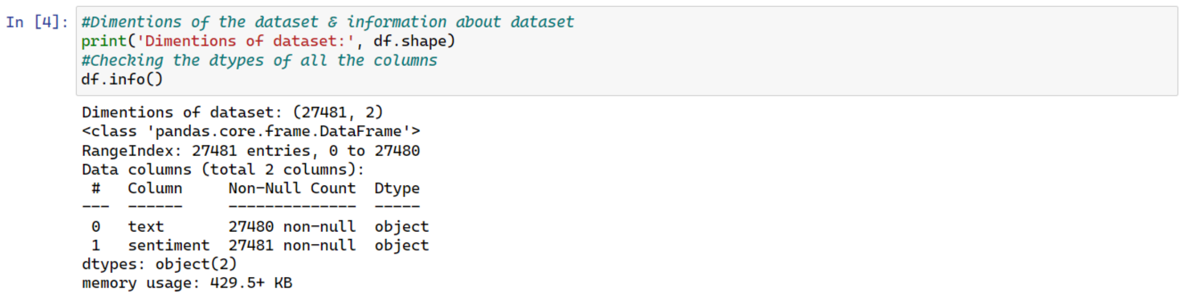
Dropping some columns

These lines of code use the drop() method with axis=1 to show columns and inplace=True to change the DataFrame in place to remove two columns: "selected_text" and "textID" from the DataFrame "df." The number "sentiment" is given to the "target" field. To start from scratch with the index of the DataFrame, use reset_index(drop=True). 'original_df' copies the original DataFrame to keep a copy that hasn't been changed. Lastly, the code uses head() to show the first few rows of the changed DataFrame "df."

```
In [3]: #Let's drop selected text & text id column
        df.drop(['selected_text', 'textID'], axis=1, inplace=True)
        target = 'sentiment'
        df.reset_index(drop=True, inplace=True) #Resetting the index
        original_df = df.copy(deep=True)
        df.head()

Out[3]:
                                                text   sentiment
   0             I`d have responded, if I were going     neutral
   1       Sooo SAD I will miss you here in San Diego!!!    negative
   2                            my boss is bullying me...    negative
   3                      what interview! leave me alone    negative
   4     Sons of ****, why couldn`t they put them on t...    negative
```

Checking Data Types and Dimensions

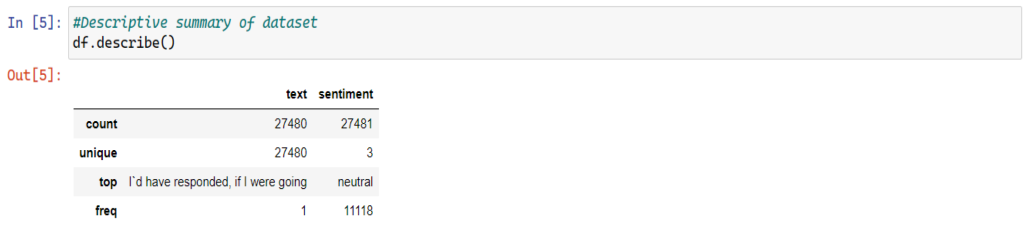
This code is mostly used for two things when preparing data for a dataset. First, it tells you the dataset's measurements, which include the number of rows and columns, which gives you a rough idea of how big it is. Second, it uses the df.info() method to show specific information about the columns of the dataset, such as the data types they hold and the number of items that are not null. This data is very important for knowing how the dataset is organised and finding any possible data quality problems, like missing numbers. By doing these things, the code helps with exploring and evaluating the dataset's features at the beginning, which sets the stage for later tasks like data analysis and modelling.



```
In [4]: #Dimentions of the dataset & information about dataset
        print('Dimentions of dataset:', df.shape)
        #Checking the dtypes of all the columns
        df.info()

        Dimentions of dataset: (27481, 2)
        <class 'pandas.core.frame.DataFrame'>
        RangeIndex: 27481 entries, 0 to 27480
        Data columns (total 2 columns):
         #   Column     Non-Null Count  Dtype 
        ---  ------     --------------  ----- 
         0   text       27480 non-null  object
         1   sentiment  27481 non-null  object
        dtypes: object(2)
        memory usage: 429.5+ KB
```

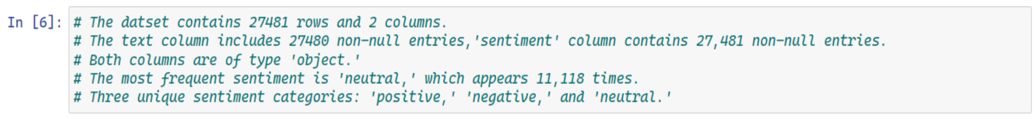
Summary of Dataset

The code df.describe() makes a description of the dataset that is written in words. For each number column in the DataFrame "df," this report gives information, such as count (number of non-null values), mean (average), standard deviation (measure of data spread), minimum, 25th percentile, median (50th percentile), 75th percentile, and highest value. The central tendency and distribution of the numerical data in the dataset can be seen through these figures. This helps you get a better idea of the dataset's features and any possible outliers.



```
In [5]: #Descriptive summary of dataset
        df.describe()

Out[5]:
```

	text	sentiment
count	27480	27481
unique	27480	3
top	I`d have responded, if I were going	neutral
freq	1	11118



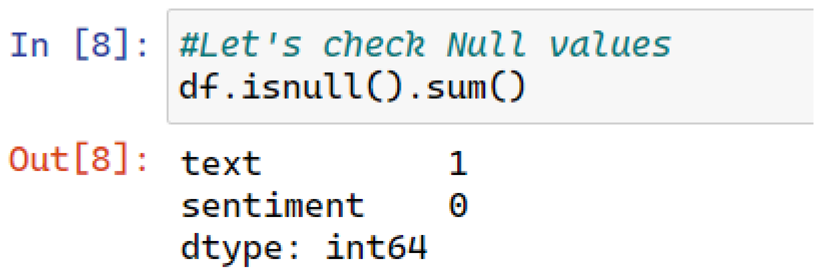
```
In [6]: # The datset contains 27481 rows and 2 columns.
        # The text column includes 27480 non-null entries,'sentiment' column contains 27,481 non-null entries.
        # Both columns are of type 'object.'
        # The most frequent sentiment is 'neutral,' which appears 11,118 times.
        # Three unique sentiment categories: 'positive,' 'negative,' and 'neutral.'
```

Null values identification and treatment

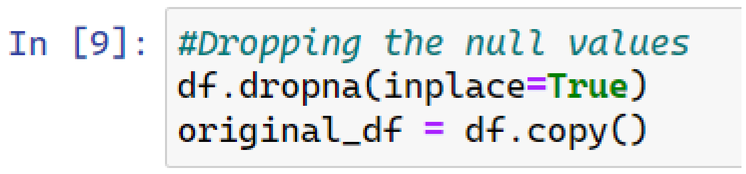
This code looks through the information to see if any numbers are missing. This is df.isnull().The sum() function finds and shows how many "null" values are in each field of the DataFrame "df." This gives you a general idea of how much data is missing from the collection.

After this evaluation, the code uses df.dropna(inplace=True) to get rid of rows in the DataFrame that have missing values, which means they are no longer in the dataset. Then, a copy of the changed DataFrame without the missing numbers is added to the "original_df."

This step of cleaning the data is necessary to make sure the information is correct and to avoid problems that could happen when analysing it or making machine learning models.



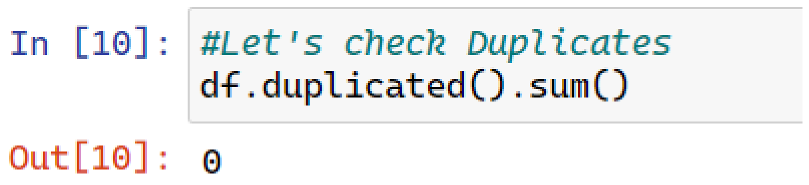
```
In [8]: #Let's check Null values
        df.isnull().sum()

Out[8]: text         1
        sentiment    0
        dtype: int64
```



```
In [9]: #Dropping the null values
        df.dropna(inplace=True)
        original_df = df.copy()
```

Checking duplicates

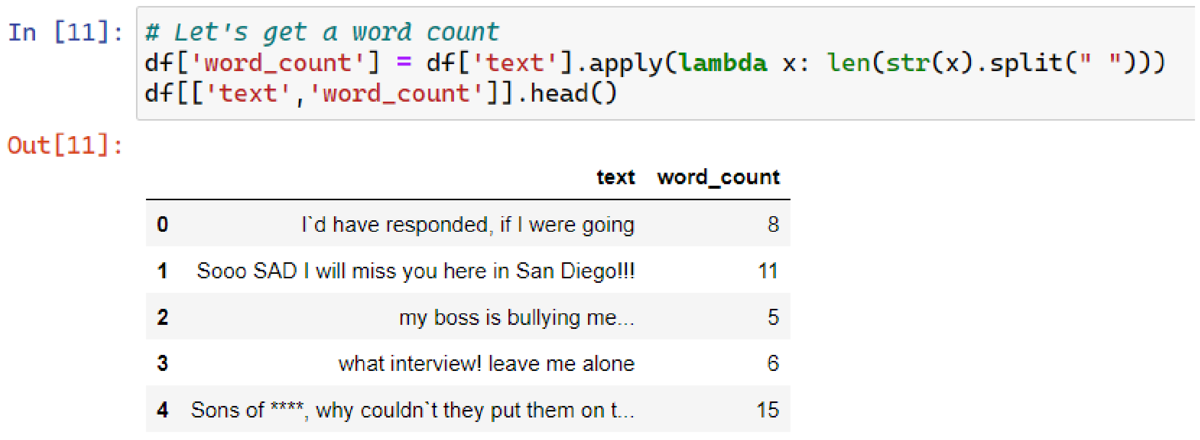
This code looks through the information for rows that are already there. The df.duplicated().sum() line counts the number of duplicate rows in the DataFrame "df" and shows that number. Rows that have the same numbers in all fields are called duplicates. Finding and dealing with similar data is important to make sure that analysis or modelling results are not skewed. This is because it makes sure that each observation is unique and only adds to the dataset's information once.



```
In [10]: #Let's check Duplicates
         df.duplicated().sum()

Out[10]: 0
```

Counting words, characters, stopwords and special characters

This code adds a new column to the DataFrame called "word_count" and finds the number of words in each text item in "df." The code in df['text'].apply(lambda x: len(str(x).split(" "))) applies a lambda function to each entry in the 'text' column. The lambda function uses a space to separate the text into words and then figures out the length of the list of words, which counts how many words are in each entry. Using df[['text','word_count']].head(), the code then shows the 'text' column and the word counts that go with it for the first few rows. This process can help you figure out how the word counts are spread out in the text data, which is useful for a number of text analysis jobs.

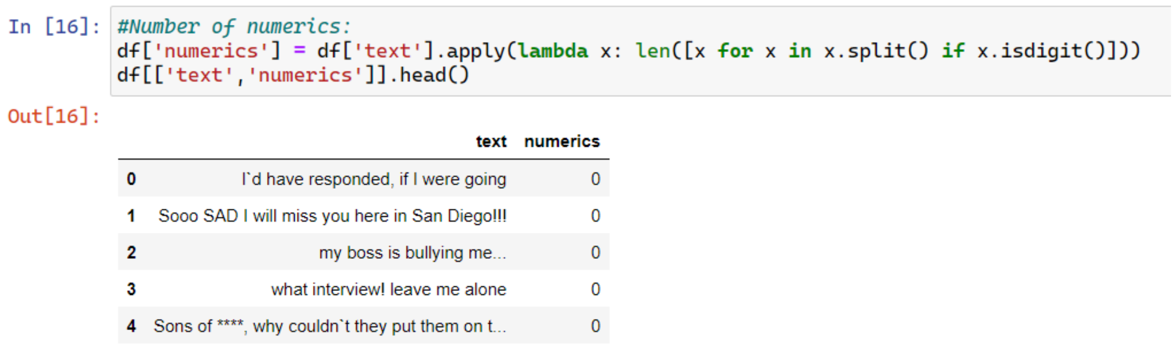


```
In [11]: # Let's get a word count
         df['word_count'] = df['text'].apply(lambda x: len(str(x).split(" ")))
         df[['text','word_count']].head()
```

Out[11]:

	text	word_count
0	I`d have responded, if I were going	8
1	Sooo SAD I will miss you here in San Diego!!!	11
2	my boss is bullying me...	5
3	what interview! leave me alone	6
4	Sons of ****, why couldn`t they put them on t...	15

In each text entry of the DataFrame "df," this code counts the amount of special characters, especially hashtags (words or lines that start with "@"). t adds a field to the DataFrame called "hashtags" to hold these counts.
You can use df['text'].apply(lambda x: len([x for x in x.split() if x.startswith('@')])) to do this. uses a lambda function to break up each item in the "text" column into words and count how many of those words begin with "@," which is the same thing as counting hashtags. Using df[['text','hashtags']].head(), the code then shows the first few rows of the "text" column along with the number of times each word was used. This information can help you figure out how many times other people are mentioned or referred to in the text data.Without a doubt, this piece of code is meant to figure out and record the number of stopwords in each text item in the DataFrame ``df." Words like "the," "and," and "is" are examples of stopwords. They are often taken out of text analysis so that more relevant words can be found. The code gets the list of English stopwords from the NLTK library and puts it in the variable called "stop." Next, a lambda function is used to break each "text" entry into words and see how many of those words are in the "stop" list. The count is then put to a new field in the DataFrame called "stopwords." This action tells us a lot about how common stop words are in the dataset, which is important for many text analysis tasks, like figuring out how important certain words or traits are when creating machine learning models for natural language processing.



```
In [16]: #Number of numerics:
         df['numerics'] = df['text'].apply(lambda x: len([x for x in x.split() if x.isdigit()]))
         df[['text','numerics']].head()
```

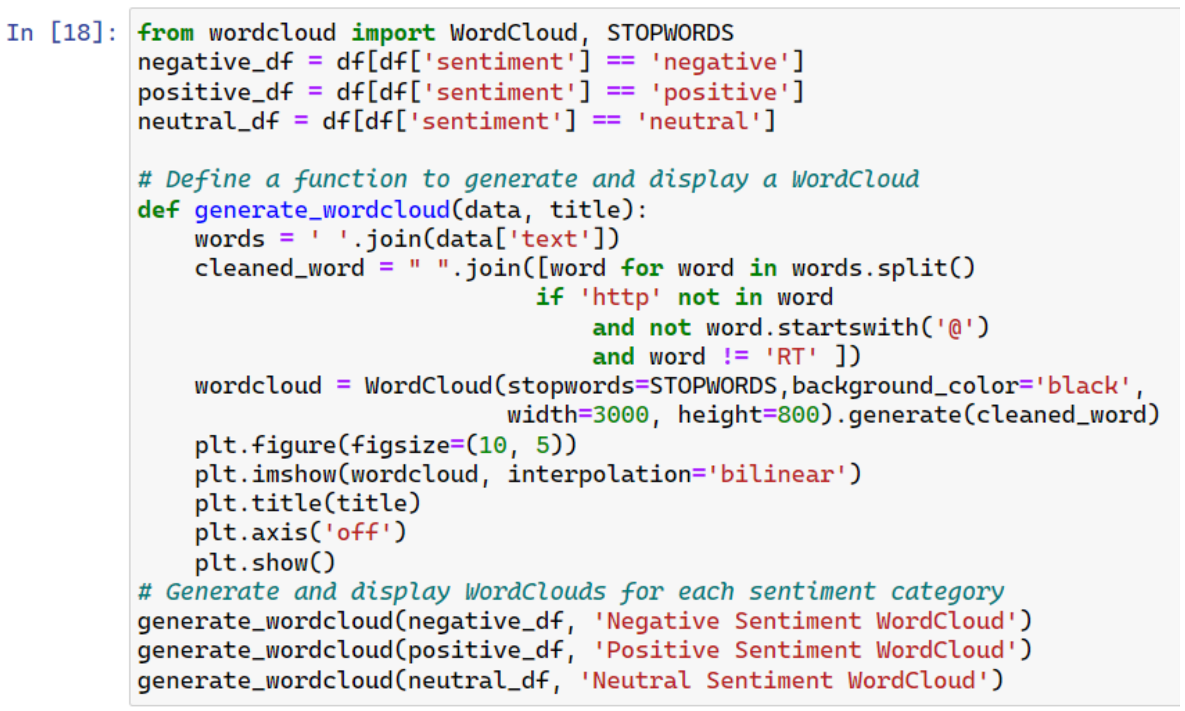
Out[16]:

	text	numerics
0	I`d have responded, if I were going	0
1	Sooo SAD I will miss you here in San Diego!!!	0
2	my boss is bullying me...	0
3	what interview! leave me alone	0
4	Sons of ****, why couldn`t they put them on t...	0

4. *Data visualisation before cleaning*

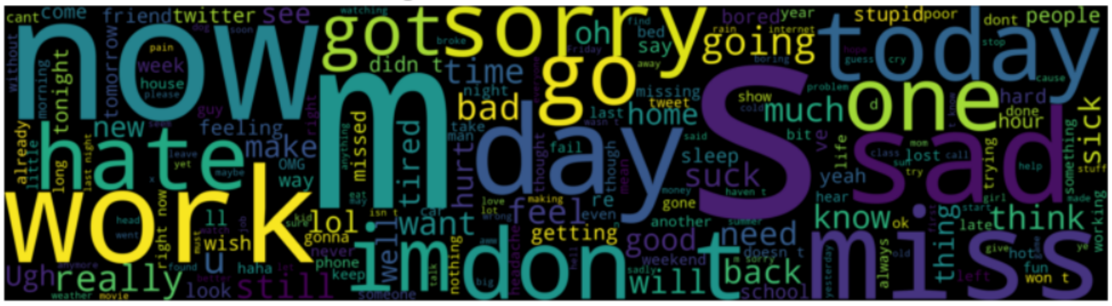
This piece of code makes use of the library known as 'word cloud' in order to build and display word clouds for a variety of emotion classifications included inside the DataFrame known as 'df.' The dataset is broken up into three different subsets depending on the respondents' feelings: 'negative_df' for responses expressing negative sentiment, 'positive_df'

for responses expressing positive sentiment, and 'neutral_df' for responses expressing neutral feelings.
'generate_wordcloud' is a function that is defined in the code, and it accepts two parameters: 'data,' which stands for the DataFrame that contains text data for a certain sentiment category; and 'title,' which is a title for the word cloud that is formed. The text data is preprocessed inside the function to eliminate common components such as URLs, Twitter usernames (beginning with '@'), and "RT" (presumably indicating retweets).
The 'WordCloud' class from the 'wordcloud' library is then used to build word clouds. To modify the look of the word cloud, properties such as stop words,' 'background_color,' 'width,' and 'height' may be specified. Finally, the 'plt.imshow' command is used to display a word cloud for each emotion category, and the 'plt.title' command is used to display the title that corresponds to the cloud. The axis labels are removed by the statement 'plt.axis('off'),' and the word cloud is shown by the statement 'plt.show()'.
This code provides a straightforward method for recognizing important keywords that are connected to a variety of emotions included in the dataset, since it graphically shows the words that appear the most often in each category of mood. It may be useful for getting insights into the vocabulary and linguistic patterns that are related with particular emotions in textual data.

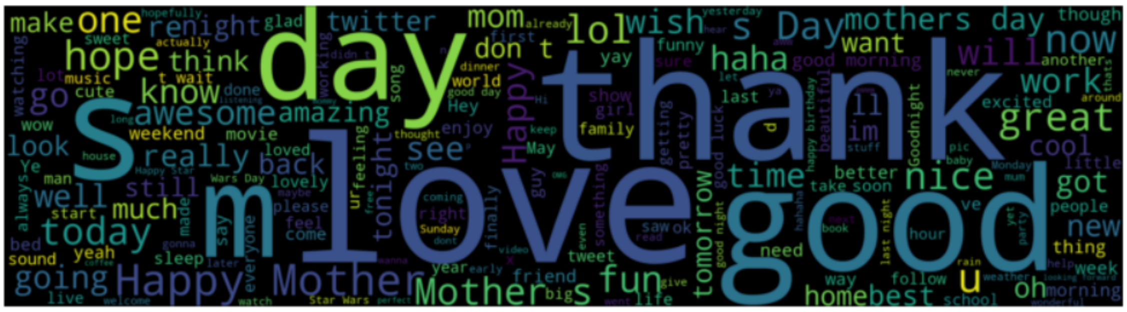


```
In [18]: from wordcloud import WordCloud, STOPWORDS
         negative_df = df[df['sentiment'] == 'negative']
         positive_df = df[df['sentiment'] == 'positive']
         neutral_df = df[df['sentiment'] == 'neutral']

         # Define a function to generate and display a WordCloud
         def generate_wordcloud(data, title):
             words = ' '.join(data['text'])
             cleaned_word = " ".join([word for word in words.split()
                                 if 'http' not in word
                                     and not word.startswith('@')
                                     and word != 'RT' ])
             wordcloud = WordCloud(stopwords=STOPWORDS,background_color='black',
                               width=3000, height=800).generate(cleaned_word)
             plt.figure(figsize=(10, 5))
             plt.imshow(wordcloud, interpolation='bilinear')
             plt.title(title)
             plt.axis('off')
             plt.show()
         # Generate and display WordClouds for each sentiment category
         generate_wordcloud(negative_df, 'Negative Sentiment WordCloud')
         generate_wordcloud(positive_df, 'Positive Sentiment WordCloud')
         generate_wordcloud(neutral_df, 'Neutral Sentiment WordCloud')
```

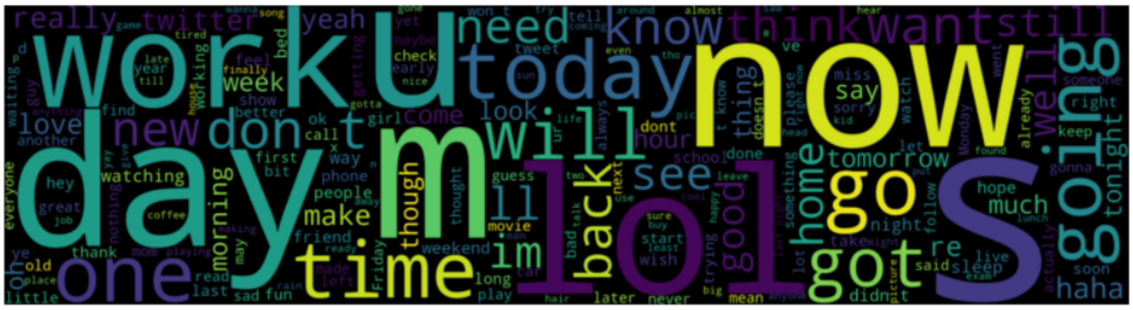
Negative Sentiment WordCloud



Positive Sentiment WordCloud



Neutral Sentiment WordCloud



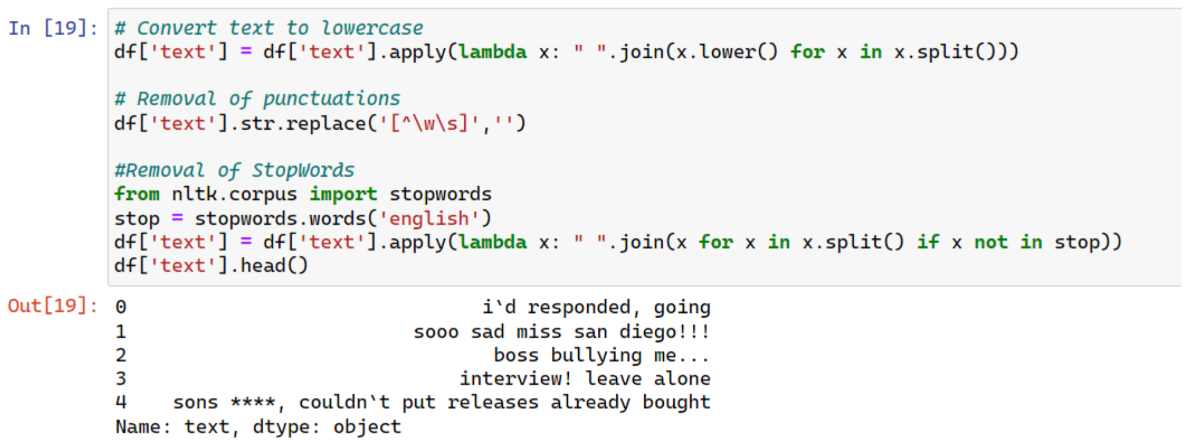
5. *Data Cleaning*

This section of code is devoted to the preparation of text data included inside the DataFrame known as 'df' in order to make it more amenable to operations involving natural language processing. It does so by carrying out a number of crucial steps:

First, it changes all of the text in the 'text' column to lowercase, which guarantees that the dataset will have consistent letter casing throughout. It is essential that this be done in order to avoid future text analysis being impacted by changes in capitalization.

The second thing that it does is use a regular expression to try to get rid of any punctuation marks that are in the text. However, in order for the change to have a long-lasting impact on the DataFrame, it has to save the result back into the 'text' column.

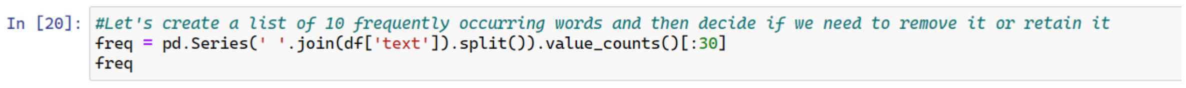
Last but not least, the program uses NLTK's stopwords list to get rid of common English stop words like "the" and "and," which have less of an impact on the overall meaning of the text and may be ignored throughout the analysis process. The outcome is then saved in the 'text' column, which results in the production of a text data column that has been thoroughly cleaned and preprocessed and is thus prepared for more complex text analysis activities. In order to improve the overall quality of the text data used in a variety of NLP applications, these preprocessing processes are very necessary.



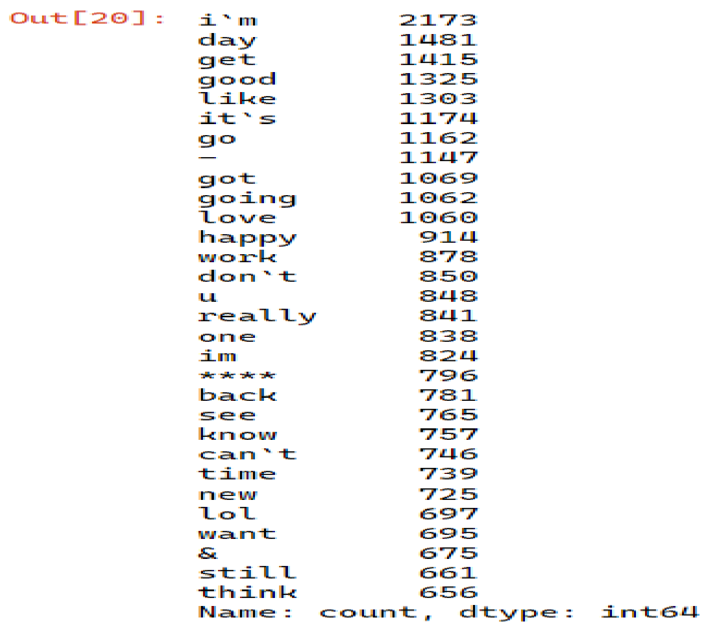
```
In [19]: # Convert text to lowercase
         df['text'] = df['text'].apply(lambda x: " ".join(x.lower() for x in x.split()))

         # Removal of punctuations
         df['text'].str.replace('[^\w\s]','')

         #Removal of StopWords
         from nltk.corpus import stopwords
         stop = stopwords.words('english')
         df['text'] = df['text'].apply(lambda x: " ".join(x for x in x.split() if x not in stop))
         df['text'].head()
```

```
Out[19]: 0                               i`d responded, going
         1                         sooo sad miss san diego!!!
         2                                  boss bullying me...
         3                               interview! leave alone
         4    sons ****, couldn`t put releases already bought
         Name: text, dtype: object
```

Despite the fact that the comment refers to a list of 10 commonly occurring words, the code produces a list of the 30 words that appear in the 'text' column of the DataFrame 'df' the most frequently. These words and their counts are stored in the 'freq' variable, which is named after the word that appears most often. These frequently used words are able to provide light on the most frequent keywords included in the text data, which may be essential for future investigation. Depending on the particular objectives of the study, you have the option of deciding whether the text processing and modelling activities you are working on should keep these terms, get rid of them, or handle them in a different way.

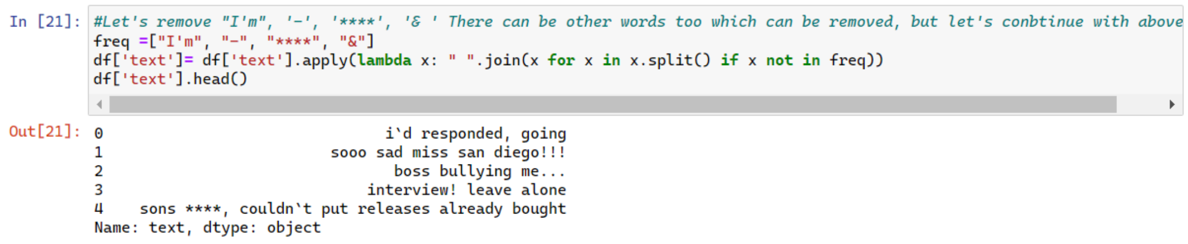


```
In [20]: #Let's create a list of 10 frequently occurring words and then decide if we need to remove it or retain it
         freq = pd.Series(' '.join(df['text']).split()).value_counts()[:30]
         freq
```



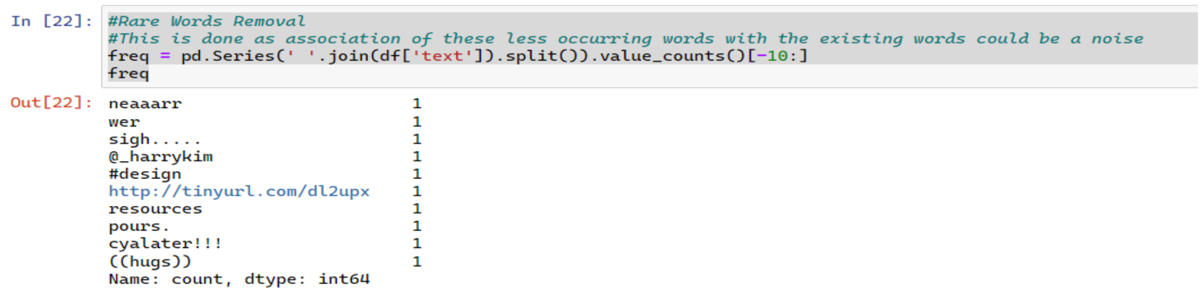
```
Out[20]: i`m          2173
         day          1481
         get          1415
         good         1325
         like         1303
         it`s         1174
         go           1162
         -            1147
         got          1069
         going        1062
         love         1060
         happy         914
         work          878
         don`t         850
         u             848
         really        841
         one           838
         im            824
         ****          796
         back          781
         see           765
         know          757
         can`t         746
         time          739
         new           725
         lol           697
         want          695
         &             675
         still         661
         think         656
         Name: count, dtype: int64
```

This piece of code eliminates certain terms from the 'text' column of the DataFrame referred to as 'df.' According to the 'freq' list, the following words will need to be eliminated from the document: "I'm," "-," "****," and "&." The program uses a lambda function to break up each text input into individual words, and it only retains those words that are not included in the 'freq' list. The result is saved in the 'text' column that was previously used. This stage is helpful for removing certain words or characters from the text analysis that are regarded to be unnecessary or harmful.



```
In [21]: #Let's remove "I'm", '-', '****', '& ' There can be other words too which can be removed, but let's conbtinue with above
         freq =["I'm", "-", "****", "&"]
         df['text']= df['text'].apply(lambda x: " ".join(x for x in x.split() if x not in freq))
         df['text'].head()

Out[21]: 0                          i'd responded, going
         1                     sooo sad miss san diego!!!
         2                              boss bullying me...
         3                           interview! leave alone
         4    sons ****, couldn't put releases already bought
         Name: text, dtype: object
```

This piece of code finds and shows the 10 words that appear in the 'text' column of the DataFrame referred to as 'df' the fewest number of times. Due to the fact that they appear so seldom in the text data, the presence of these words has earned them the status of being uncommon. The removal of seldom occurring words is a typical step in the preparation of text. This is done for the reason that these words may generate noise and do not contribute substantially to the analysis. The data may be simplified by excluding these uncommon terms, and, depending on the particular text analysis job at hand, the quality of the analysis findings might possibly be improved as a consequence.



```
In [22]: #Rare Words Removal
         #This is done as association of these less occurring words with the existing words could be a noise
         freq = pd.Series(' '.join(df['text']).split()).value_counts()[-10:]
         freq

Out[22]: neaaarr                        1
         wer                            1
         sigh.....                      1
         @_harrykim                     1
         #design                        1
         http://tinyurl.com/dl2upx      1
         resources                      1
         pours.                         1
         cyalater!!!                    1
         ((hugs))                       1
         Name: count, dtype: int64
```

The purpose of the code snippet that has been supplied is, without a doubt, to apply stemming, a method for normalising text, to the first five rows of the 'text' column that is included inside the DataFrame 'df.' The process of simplifying words by deleting suffixes, such as "ing" or "ly," in order to reduce them to their base or root forms is referred to as stemming.

To begin, the program makes use of the NLTK library to bring in the Porter Stemmer algorithm, which is a method that is often used for stemming words.

Following that, it creates an instance of the Porter Stemmer with the name ist' so that it may be used in stemming operations.

The lambda function is what is used to carry out the primary operation. This function will cycle over the text entries in the first five rows, break each entry into individual words, apply stemming to each word via the Porter Stemmer, and then reunite the stemmed words into a sentence at the end of the process.

This piece of code demonstrates how stemming may improve the quality of the text data by reducing complex words to their most fundamental forms. Stemming is useful in natural language processing jobs because it limits the number of different words that may be used to mean the same thing. This enables more effective and accurate text analysis, such as text categorization or the retrieval of information.

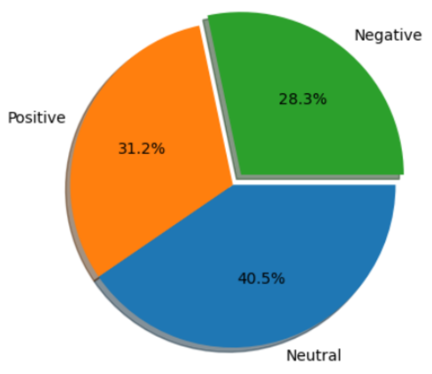
```
In [23]: #Stemming -refers to the removal of suffices, like “ing”, “ly”, “s”, etc. by a simple rule-based approach¶
         from nltk.stem import PorterStemmer
         st = PorterStemmer()
         df['text'][:5].apply(lambda x: " ".join([st.stem(word) for word in x.split()]))

Out[23]: 0                              i`d responded, go
         1                       sooo sad miss san diego!!!
         2                                   boss bulli me...
         3                             interview! leav alon
         4      son ****, couldn`t put releas alreadi bought
         Name: text, dtype: object
```

6. *Data visualisation after cleaning*

The portion of code that has been supplied is intended to create a graphical representation of the distribution of sentiment categories included inside the dataset that is being held in the DataFrame with the name 'df.' It does so by using the plt.pie() method included in the matplotlib.pyplot package to produce a pie chart. The piece of code calculates the count of each sentiment category ('Neutral,' 'Positive,' and 'Negative') that is included inside the sentiment' column of the DataFrame, and then presents them as segments in the pie chart. The chart has been developed with a few different aesthetic embellishments, such as a shadow effect, an expanded part that is just barely separated, and percentage labels for each component of the chart. This visualisation provides a comprehensive summary of the manner in which sentiments are dispersed across the dataset, making it possible to make a speedy evaluation of the sentiment composition of the dataset. The graphic presents the proportion of each sentiment category as a percentage of the total, which makes it simple to comprehend the distribution of sentiments in their whole.

```
In [24]: #Let's look at the overall distribution of positive, negative and neutral sentiments
         print('\033[1mSentiment Variable Distribution'.center(55))
         plt.pie(df['sentiment'].value_counts(), labels=['Neutral','Positive','Negative'], counterclock=False, shadow=True,
                 explode=[0,0,0.08], autopct='%1.1f%%', radius=1, startangle=0)
         plt.show()
```

Sentiment Variable Distribution

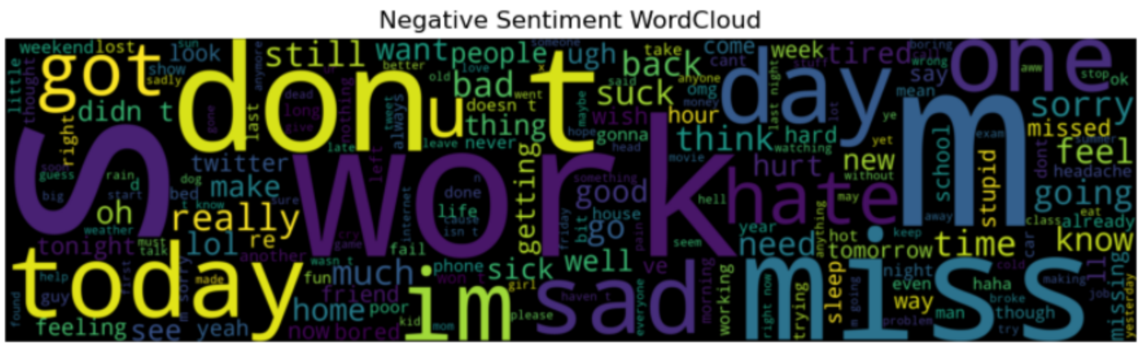


This section of code is responsible for creating and presenting word clouds that depict the most frequently occurring words in distinct emotion categories inside a dataset that is contained in the DataFrame named 'df.' It starts by separating the data into three different subsets, which are labelled as follows: 'negative_df' for negative sentiment, 'positive_df' for positive sentiment, and 'neutral_df' for neutral sentiment.

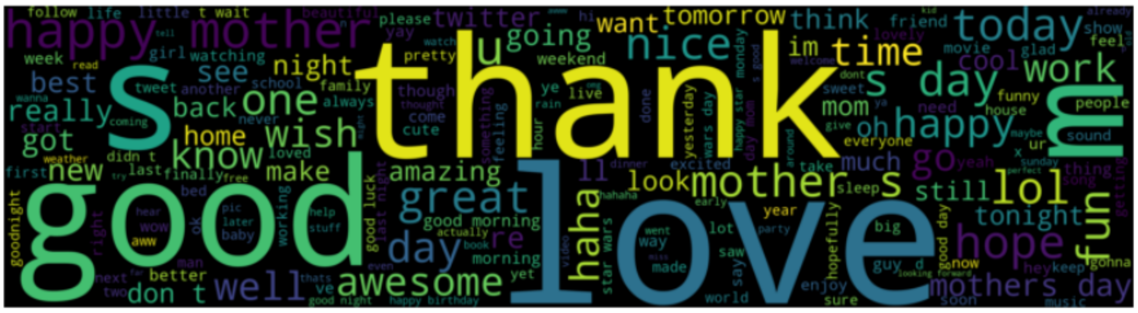
Next, the code establishes a bespoke operation known as 'generate_wordcloud' in order to produce the word clouds. This method builds a word cloud by concatenating all of the text data that falls under a given emotion category, removing specific components like URLs, Twitter handles, and "RT" (which indicates retweets), and then removing those specific pieces from the text itself.

For this purpose, the 'WordCloud' class included in the 'wordcloud' library is utilised. This class has options for configuring the stopwords, backdrop colour, and size of the word cloud. The word clouds that were generated as a consequence are presented in a unique manner for each of the distinct emotion categories. This results in a visual depiction of the words that are most often connected with each feeling. Through the use of this visualisation, insights into the prominent language patterns and important phrases within each sentiment category of the dataset may be gained.

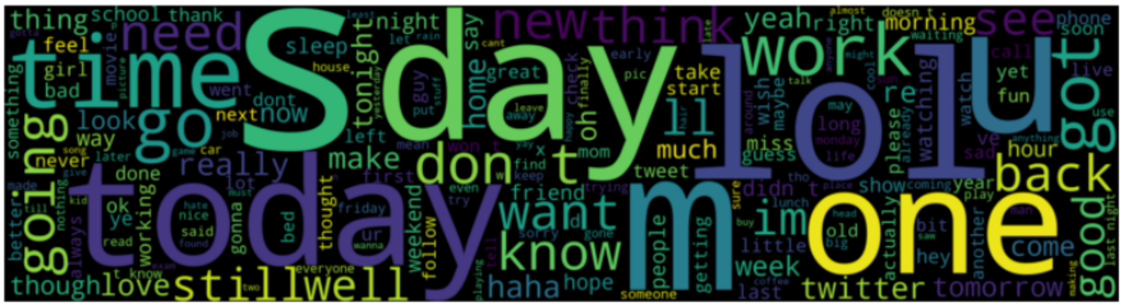
```
In [26]: from wordcloud import WordCloud, STOPWORDS
         negative_df = df[df['sentiment'] == 'negative']
         positive_df = df[df['sentiment'] == 'positive']
         neutral_df = df[df['sentiment'] == 'neutral']

         # Define a function to generate and display a WordCloud
         def generate_wordcloud(data, title):
             words = ' '.join(data['text'])
             cleaned_word = " ".join([word for word in words.split()
                                     if 'http' not in word
                                         and not word.startswith('@')
                                         and word != 'RT' ])
             wordcloud = WordCloud(stopwords=STOPWORDS,background_color='black',
                                   width=3000, height=800).generate(cleaned_word)
             plt.figure(figsize=(10, 5))
             plt.imshow(wordcloud, interpolation='bilinear')
             plt.title(title)
             plt.axis('off')
             plt.show()
         # Generate and display WordClouds for each sentiment category
         generate_wordcloud(negative_df, 'Negative Sentiment WordCloud')
         generate_wordcloud(positive_df, 'Positive Sentiment WordCloud')
         generate_wordcloud(neutral_df, 'Neutral Sentiment WordCloud')
```



Positive Sentiment WordCloud



Neutral Sentiment WordCloud



7. *Evaluation*

Predictive Modelling

These lines of code are absolutely necessary in order to properly prepare the data for analysis using machine learning, in particular for the categorization of sentiments. The first line of the program, which reads 'X = df['text']', takes the column named 'text' from the DataFrame named 'df' and assigns it to the variable referred to as 'X.' In the field of machine learning, the letter 'X' often stands for the feature or the input data. In this particular instance, the letter 'X' holds the text data that was taken from the dataset and will be utilised as input for sentiment analysis.

The second line, which reads 'y = df['sentiment'], takes the column named 'sentiment' from the DataFrame referred to as 'df' and assigns it to the variable referred to as 'y.' In the field of machine learning, the letter 'y' stands for the target variable or labels that you want the machine learning model to be able to predict. Here, the column labelled 'y' stores the sentiment labels that are connected with each text item in the column labelled 'X,' indicating whether the sentiment is positive, negative, or neutral.

You are laying the framework for supervised machine learning when you divide the dataset into 'X' (features) and 'y' (labels). In this kind of machine learning, a model is trained to predict sentiment based on the textual content, therefore it is important to divide the dataset into features and labels. During the steps of developing a sentiment analysis model, the variables 'X' and 'y' will be employed in training the model, evaluating the model, and testing the model.

```
In [28]: X = df['text']
         y = df['sentiment']
```

This piece of code utilises the 'TfidfVectorizer' function included in the scikit-learn package in order to transform the textual input into numerical features that are compatible with machine learning. The 'TfidfVectorizer' was developed expressly for this objective. It makes use of the TF-IDF (Term Frequency-Inverse Document Frequency) weighting method, which provides numerical values to words in accordance with the significance they have within the text corpus.

The 'TfidfVectorizer' class from the scikit-learn library is imported into the code first. Then, it starts a new instance of the vectorizer with the name vectorizer.' The 'X' variable, which stores the text data, will be subjected to the vectorization procedure in the next phase. During this stage, the vectorizer is applied to the text data, and the information is then transformed into a TF-IDF matrix.

The resultant TF-IDF matrix is then converted into a format that is compatible with dense arrays by using the '.toarray()' technique. The data that was previously represented in 'X' has been converted into a numerical feature matrix, making it appropriate for use in machine learning models. Because of these properties, machine learning algorithms are able to successfully deal with the text data, since they capture the value of individual words within the text. In order to train and evaluate models for sentiment analysis, as well as perform a variety of other text-based machine learning tasks, this preprocessing phase is essential.

```
In [29]: from sklearn.feature_extraction.text import TfidfVectorizer

         vectorizer = TfidfVectorizer ()
         X = vectorizer.fit_transform(X).toarray()
         vectorizer

Out[29]: ▾ TfidfVectorizer
         TfidfVectorizer()
```

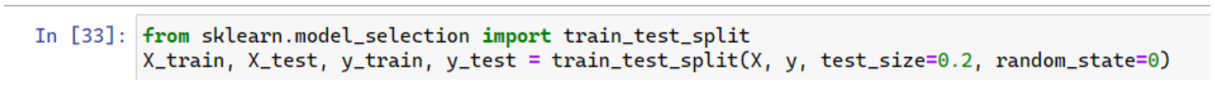
Splitting data

You are now carrying out the critical step of segmenting your dataset into training and testing subsets for the purpose of developing and evaluating a machine learning model in this piece of code. You have simplified the procedure by using the 'train_test_split' method that scikit-learn provides. This allowed you to do what you set out to do.

The code uses the feature data that you have saved in the variable 'X' (which represents the preprocessed text data) and the target labels that you have saved in the variable 'y' (which indicates sentiment categories). You may assign 20% of the data for testing by supplying the value 'test_size=0.2,' with the remaining 80% of the data being used for training the model.

The 'random_state=0' argument guarantees that the procedure of random division may be repeated accurately. You can assure that you will always obtain the same split by running this code several times and setting a random seed ('random_state') to zero each time. This is important for maintaining consistency while testing and analysing various models.

This procedure of splitting creates four subsets as a result: 'X_train' (training features), 'X_test' (testing features), 'y_train' (training labels), and 'y_test' (testing labels). 'X_train' refers to the training features, and 'X_test' refers to the testing features. During the training phase of the machine learning model, the variables 'X_train' and 'y_train' will be utilised, and during the testing phase, the variables 'X_test' and 'y_test' will be used to assess how well the model performs on data that it has not previously seen. This division is critical for determining whether or not the model is able to generalise and make correct predictions on text material that it has not previously encountered.



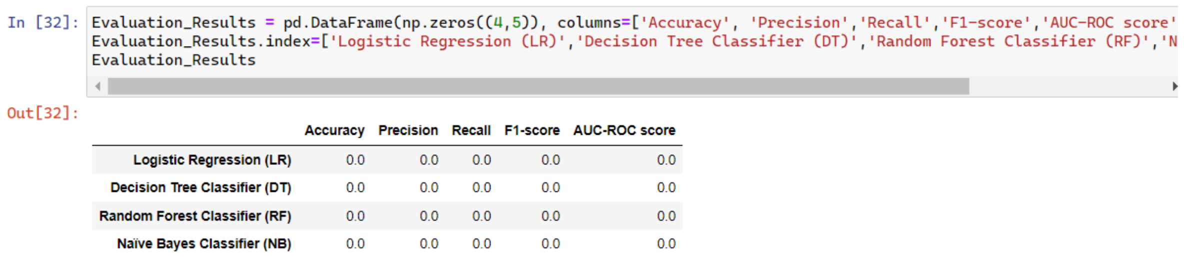
```
In [33]: from sklearn.model_selection import train_test_split
         X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=0)
```

Creating evaluation table

In this section of code, you are creating a structured DataFrame with the name 'Evaluation_Results' for the purpose of storing the evaluation metrics of several machine learning classifiers in a methodical manner. The DataFrame is intended to comprise four rows, each of which will represent a different machine learning classifier, and five columns, each of which would have a different measure for assessment.

You fill the DataFrame with zeros by using the command 'pd.DataFrame(np.zeros((4,5)), columns=['Accuracy', 'Precision','Recall','F1-score','AUC-ROC score'])'. This allows you to initialise the DataFrame. Logistic Regression (LR), Decision Tree Classifier (DT), Random Forest Classifier (RF), and Naive Bayes Classifier (NB) are the four different classifiers that are represented by the different rows in this table.

In addition, you may make sure that each row belongs to the appropriate classifier by using the 'Evaluation_Results.index=' syntax to provide the names of the classifiers as row indices. The performance metrics of these classifiers may be recorded and compared inside this organised DataFrame, which will make it much simpler to evaluate and present the findings in a complete manner. It is usual practice to employ measures such as accuracy, precision, recall, F1-score, and AUC-ROC score when assessing the efficacy of classification models. These metrics are included in the evaluation.



```
In [32]: Evaluation_Results = pd.DataFrame(np.zeros((4,5)), columns=['Accuracy', 'Precision','Recall','F1-score','AUC-ROC score'
         Evaluation_Results.index=['Logistic Regression (LR)','Decision Tree Classifier (DT)','Random Forest Classifier (RF)','N
         Evaluation_Results
```

Out[32]:

	Accuracy	Precision	Recall	F1-score	AUC-ROC score
Logistic Regression (LR)	0.0	0.0	0.0	0.0	0.0
Decision Tree Classifier (DT)	0.0	0.0	0.0	0.0	0.0
Random Forest Classifier (RF)	0.0	0.0	0.0	0.0	0.0
Naïve Bayes Classifier (NB)	0.0	0.0	0.0	0.0	0.0

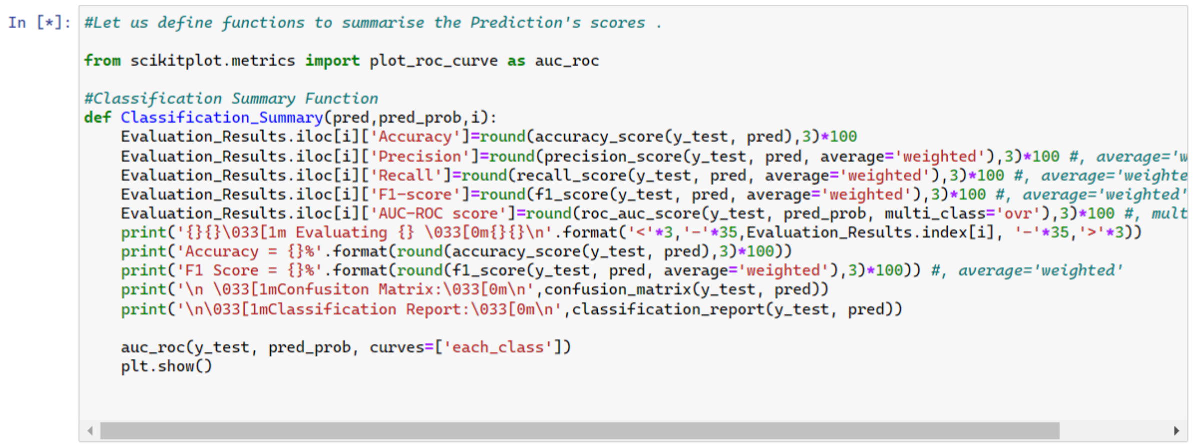
Creating summary function

A Python function with the name "Classification_Summary" is defined by the given code snippet. This function's purpose is to summarize and show evaluation metrics for machine learning classifiers. This function requires three parameters: 'pred' for predicted labels, 'pred_prob' for predicted probabilities, and 'i' as an index identifying the classifier that is being evaluated in the function.

A number of important classification measures, including as accuracy, precision (weighted average), recall (weighted average), F1-score (weighted average), and AUC-ROC score, are produced inside the function. Not only are these metrics generated, but they are also saved in the 'Evaluation_Results' DataFrame, which was created earlier, so that they may be referred to and compared at a later time.

Following this, the function will output a well-structured summary for the classifier that is being evaluated. This summary will contain the accuracy score, the F1 score, a confusion matrix, and a classification report. In addition, by using the 'auc_roc' function, it provides a graphical representation of the Receiver Operating Characteristic (ROC) curves, which is helpful for evaluating the effectiveness of classifiers.

In general, the 'Classification_Summary' function offers a method that is both efficient and instructive for evaluating and contrasting the overall performance of several machine learning classifiers. It makes the process of measuring model accuracy and efficacy more straightforward, which in turn makes it simpler to choose the classifier that is best suited to a certain endeavor.



```
In [*]: #Let us define functions to summarise the Prediction's scores .

        from scikitplot.metrics import plot_roc_curve as auc_roc

        #Classification Summary Function
        def Classification_Summary(pred,pred_prob,i):
            Evaluation_Results.iloc[i]['Accuracy']=round(accuracy_score(y_test, pred),3)*100
            Evaluation_Results.iloc[i]['Precision']=round(precision_score(y_test, pred, average='weighted'),3)*100 #, average='w
            Evaluation_Results.iloc[i]['Recall']=round(recall_score(y_test, pred, average='weighted'),3)*100 #, average='weighte
            Evaluation_Results.iloc[i]['F1-score']=round(f1_score(y_test, pred, average='weighted'),3)*100 #, average='weighted'
            Evaluation_Results.iloc[i]['AUC-ROC score']=round(roc_auc_score(y_test, pred_prob, multi_class='ovr'),3)*100 #, mult
            print('{}{}\033[1m Evaluating {} \033[0m{}{}\n'.format('<'*3,'-'*35,Evaluation_Results.index[i], '-'*35,'>'*3))
            print('Accuracy = {}%'.format(round(accuracy_score(y_test, pred),3)*100))
            print('F1 Score = {}%'.format(round(f1_score(y_test, pred, average='weighted'),3)*100)) #, average='weighted'
            print('\n \033[1mConfusiton Matrix:\033[0m\n',confusion_matrix(y_test, pred))
            print('\n\033[1mClassification Report:\033[0m\n',classification_report(y_test, pred))

            auc_roc(y_test, pred_prob, curves=['each_class'])
            plt.show()
```

Visualising the summary

This piece of code introduces a Python function named 'AUC_ROC_plot' that can be used to display and analyze the performance of a machine learning classifier by making use of ROC (Receiver Operating Characteristic) curves and the related AUC-ROC (Area Under the ROC Curve) score. The following is an in-depth description of what this code really achieves:

1. **Function goal**: The 'AUC_ROC_plot' function has the goal of computing the AUC-ROC score for a binary classification issue as well as creating ROC curves. The purpose of this test is to evaluate how well a machine learning classifier differentiates between positive and negative classifications.

2. The "Input Parameters" are as follows: The function is looking for two arguments to be sent in:
 - "y_test": This parameter is the ground truth for the evaluation since it reflects the real labels that were taken from the test dataset.
 This parameter, which is denoted by the letter 'pred,' refers to the predicted probabilities that are produced by the machine learning classifier. These probabilities represent the possibility that each data point is a member of the positive class.

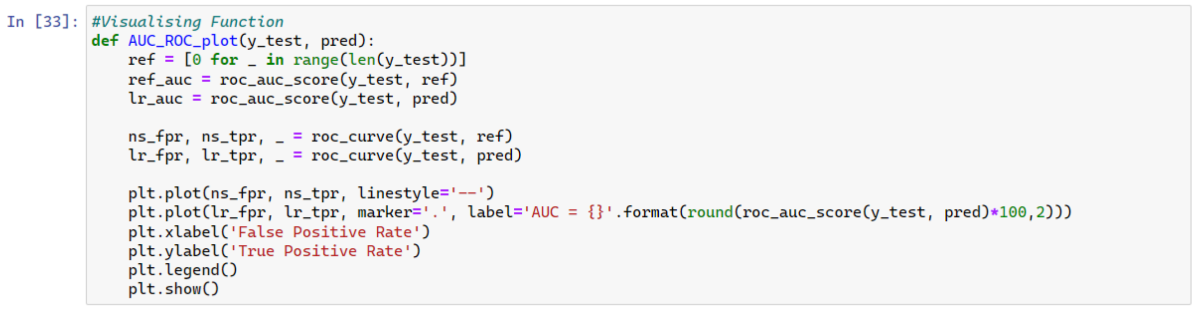
3. Calculation of the AUC and ROC: The function is responsible for computing two different AUC-ROC scores:
 This score provides the AUC-ROC value for a reference or baseline model, which is often a random or "no skill" classifier. The term "ref_auc" refers to the value represented by this score. It offers a point of reference against which the performance of the classifier may be evaluated.
 - The 'lr_auc' score is the AUC-ROC value that is being calculated for the classifier that is being evaluated. It provides a quantitative measure of the classifier's ability to differentiate between the positive and the negative classes.

4. **Data from the ROC Curve**: The 'roc_curve' function is used in the code to determine the False Positive Rate (FPR) and the True Positive Rate (TPR) for both the reference model

and the evaluated classifier. These rates are shown as a percentage. These data points are absolutely necessary in order to draw the ROC curves correctly.

Matplotlib is used to generate graphical representations of the ROC curves, which is step number five in the plotting process. The ROC curve for the reference model is shown using a dashed line in the code, whereas the ROC curve for the evaluated classifier is plotted using markers. The AUC-ROC score for the classifier that is being evaluated may be found shown in the plot's legend.

In general, this function offers a helpful way to evaluate and evaluate and compare the discriminatory strength and performance of various binary classifiers. The AUC-ROC score provides a quantitative representation of the overall performance of the classifier, while the ROC curves provide a visual representation of the trade-off between the true positive and false positive rates at varied decision thresholds. This information is critical for deciding which categorization models to use and for fine-tuning those models.



```
In [33]: #Visualising Function
         def AUC_ROC_plot(y_test, pred):
             ref = [0 for _ in range(len(y_test))]
             ref_auc = roc_auc_score(y_test, ref)
             lr_auc = roc_auc_score(y_test, pred)

             ns_fpr, ns_tpr, _ = roc_curve(y_test, ref)
             lr_fpr, lr_tpr, _ = roc_curve(y_test, pred)

             plt.plot(ns_fpr, ns_tpr, linestyle='--')
             plt.plot(lr_fpr, lr_tpr, marker='.', label='AUC = {}'.format(round(roc_auc_score(y_test, pred)*100,2)))
             plt.xlabel('False Positive Rate')
             plt.ylabel('True Positive Rate')
             plt.legend()
             plt.show()
```

Logistic Regression Model

You are constructing a Logistic Regression classifier in these lines of code, after which you are training it on the training data, making predictions on the test data, and finally producing a classification summary. The following is a report of the following actions:

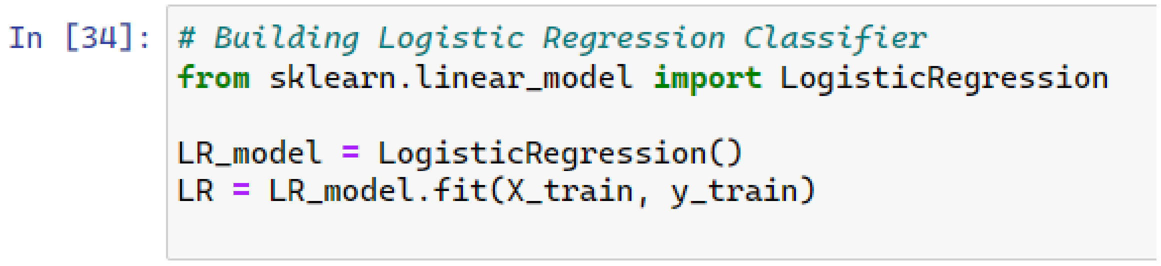
1. The **Initialization of the Logistic Regression Classifier**: 'from sklearn.linear_model import LogisticRegression' is the command you will use to bring the Logistic Regression classifier across from scikit-learn. Next, an instance of the Logistic Regression model is created, and you assign it to the variable 'LR_model' by writing 'LR_model = LogisticRegression()' in the appropriate location.

2. "Model Training": You use the training data to "fit" (train) the Logistic Regression model by entering "LR = LR_model.fit(X_train, y_train)" into your computer. Adjusting the model's parameters so that it can recognize patterns and correlations in the training data is the task at hand for this stage.

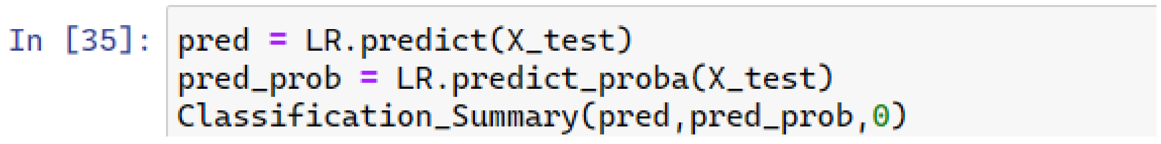
3. Utilising the Trained Model to generate Predictions: Once the training is complete, you will use the trained model to generate predictions based on the test data. 'pred = LR.predict(X_test)' is responsible for storing the predicted sentiment labels for the test data, while 'pred_prob = LR.predict_proba(X_test)' is responsible for capturing the predicted probabilities associated with each sentiment category.

4. **Classification Summary**: When you are finished, you need to make a call to the function named 'Classification_Summary' in order to obtain a detailed summary of classification metrics and assessment outcomes for the Logistic Regression classifier. This function is responsible for calculating and displaying a variety of metric values, including accuracy, precision, recall, F1-score, confusion matrix, and a classification report. In addition to this, it offers the area under the ROC curve (AUC-ROC) score so that you may evaluate the classifier's effectiveness.

The whole process of training and assessing a Logistic Regression classifier for sentiment analysis is encapsulated inside these lines of code, which together make up the full pipeline. You will be able to determine how effectively the classifier works by analysing the classification summary. This will help you to differentiate between positive, negative, and neutral feelings in the test data.

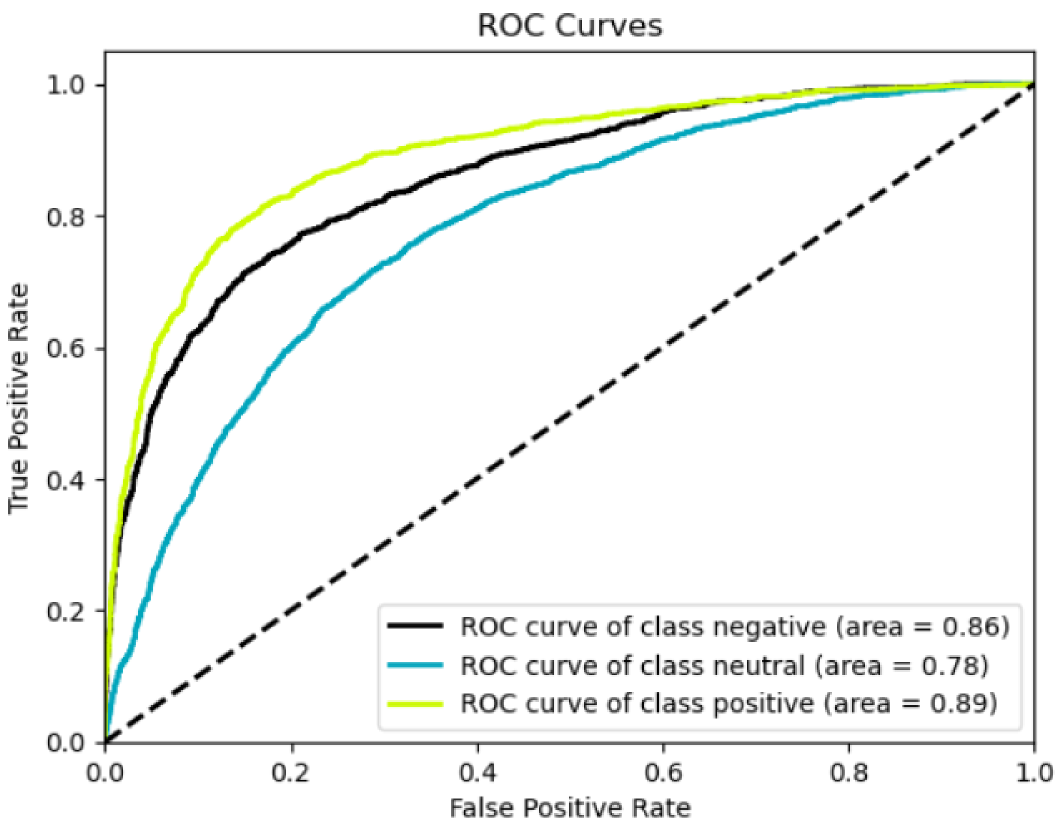


```
In [34]: # Building Logistic Regression Classifier
         from sklearn.linear_model import LogisticRegression

         LR_model = LogisticRegression()
         LR = LR_model.fit(X_train, y_train)
```



```
In [35]: pred = LR.predict(X_test)
         pred_prob = LR.predict_proba(X_test)
         Classification_Summary(pred,pred_prob,0)
```

```
<<<----------------------------------- Evaluating Logistic Regression (LR) ----------------------------------->>>

Accuracy = 68.5%
F1 Score = 68.4%

 Confusiton Matrix:
 [[ 848  589   86]
 [ 257 1775  243]
 [  51  505 1142]]

Classification Report:
               precision    recall  f1-score   support

    negative       0.73      0.56      0.63      1523
     neutral       0.62      0.78      0.69      2275
    positive       0.78      0.67      0.72      1698

    accuracy                           0.69      5496
   macro avg       0.71      0.67      0.68      5496
weighted avg       0.70      0.69      0.68      5496
```



Decision Tree Classifier

In this code segment, a Decision Tree classifier is being built, trained on the training data, predictions are being made on the test data, and a classification summary is being generated. Here's an overview of the actions that were taken.

1. **Initialization of the Decision Tree Classifier**: A Decision Tree classifier is created by importing it from scikit-learn using `from sklearn.tree import DecisionTreeClassifier`. Then, the Decision Tree model is instantiated and assigned to the variable `DT_model` with the line `DT_model = DecisionTreeClassifier()`.

2. **Model Training**: The Decision Tree model was fit (trained) using the training data with `DT = DT_model.fit(X_train, y_train)`. During this process, a decision tree structure is created by the model based on the features and labels in the training data.

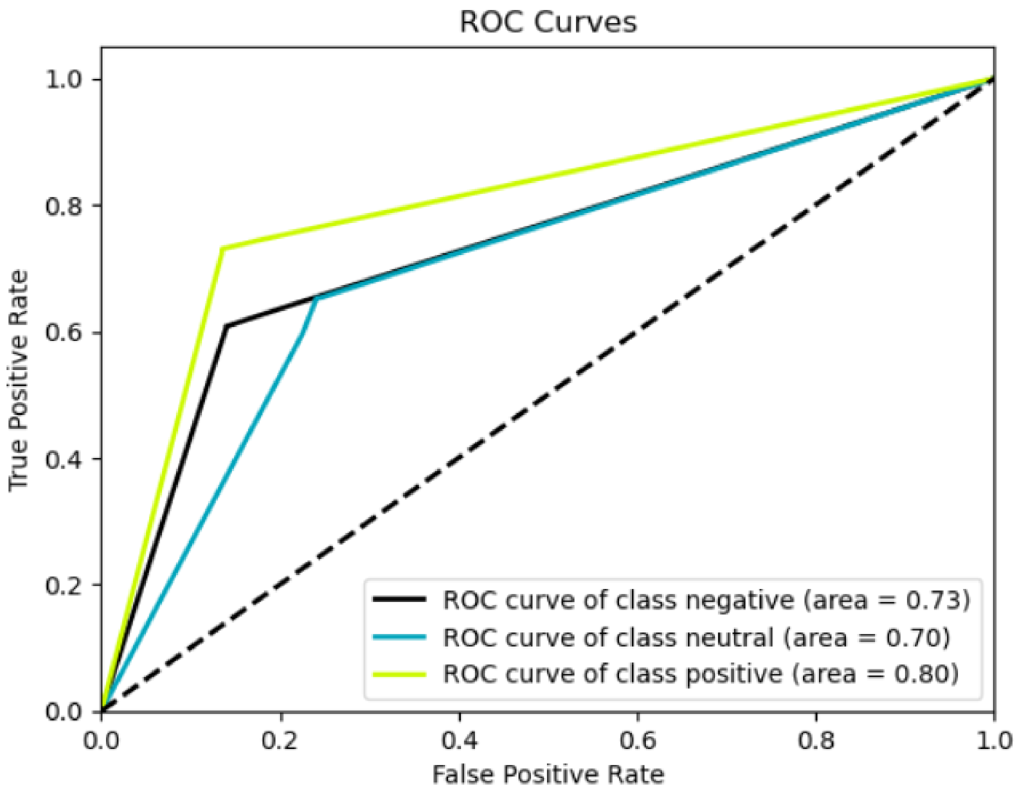
3. **Predictions Made**: After being trained, the trained Decision Tree model is used to make predictions on the test data. The predicted sentiment labels for the test data were stored in `pred = DT.predict(X_test)`, and the predicted probabilities associated with each sentiment category were captured in `pred_prob = DT.predict_proba(X_test)`.

4. **Classification Summary**: The `Classification_Summary` function was then called by me to generate a detailed summary of classification metrics and evaluation results for the Decision Tree classifier. Various metrics, including accuracy, precision, recall, F1-score, confusion matrix, and a classification report, were computed and displayed by this function. Additionally, the ROC curves are visualised and the AUC-ROC score is provided to assess the performance of the classifier.

The entire process of training, testing, and evaluating a Decision Tree classifier for sentiment analysis was represented by these lines of code. The classification summary allowed me to assess how well the Decision Tree classifier distinguished between positive, negative, and neutral sentiments in the test data, making it a valuable step in model evaluation.

```
In [36]: # Decision Tree classifier
         DT_model = DecisionTreeClassifier()
         DT = DT_model.fit(X_train, y_train)
         pred = DT.predict(X_test)
         pred_prob = DT.predict_proba(X_test)
         Classification_Summary(pred,pred_prob,1)
```

```
<<<----------------------------------- Evaluating Decision Tree Classifier (DT) ----------------------------------->>>

Accuracy = 66.4%
F1 Score = 66.3%

 Confusiton Matrix:
 [[ 925  435  163]
 [ 440 1485  350]
 [ 116  343 1239]]

Classification Report:
               precision    recall  f1-score   support

    negative       0.62      0.61      0.62      1523
     neutral       0.66      0.65      0.65      2275
    positive       0.71      0.73      0.72      1698

    accuracy                           0.66      5496
   macro avg       0.66      0.66      0.66      5496
weighted avg       0.66      0.66      0.66      5496
```



Random Forest Model
In this code segment, a Random Forest classifier is being built, trained on the training data, predictions are being made on the test data, and a classification summary is being generated. Here's an overview of the actions that were taken:

1. **Initialization of the Random Forest Classifier**: A Random Forest classifier is created by importing it from scikit-learn using `from sklearn.ensemble import RandomForestClassifier`. Then, the Random Forest model is instantiated and assigned to the variable `RF_model` with the line `RF_model = RandomForestClassifier()`.

2. **Model Training**: The Random Forest model was fit (trained) using the training data with `RF = RF_model.fit(X_train, y_train)`. During this process, an ensemble of decision trees is built by the model based on the features and labels in the training data.

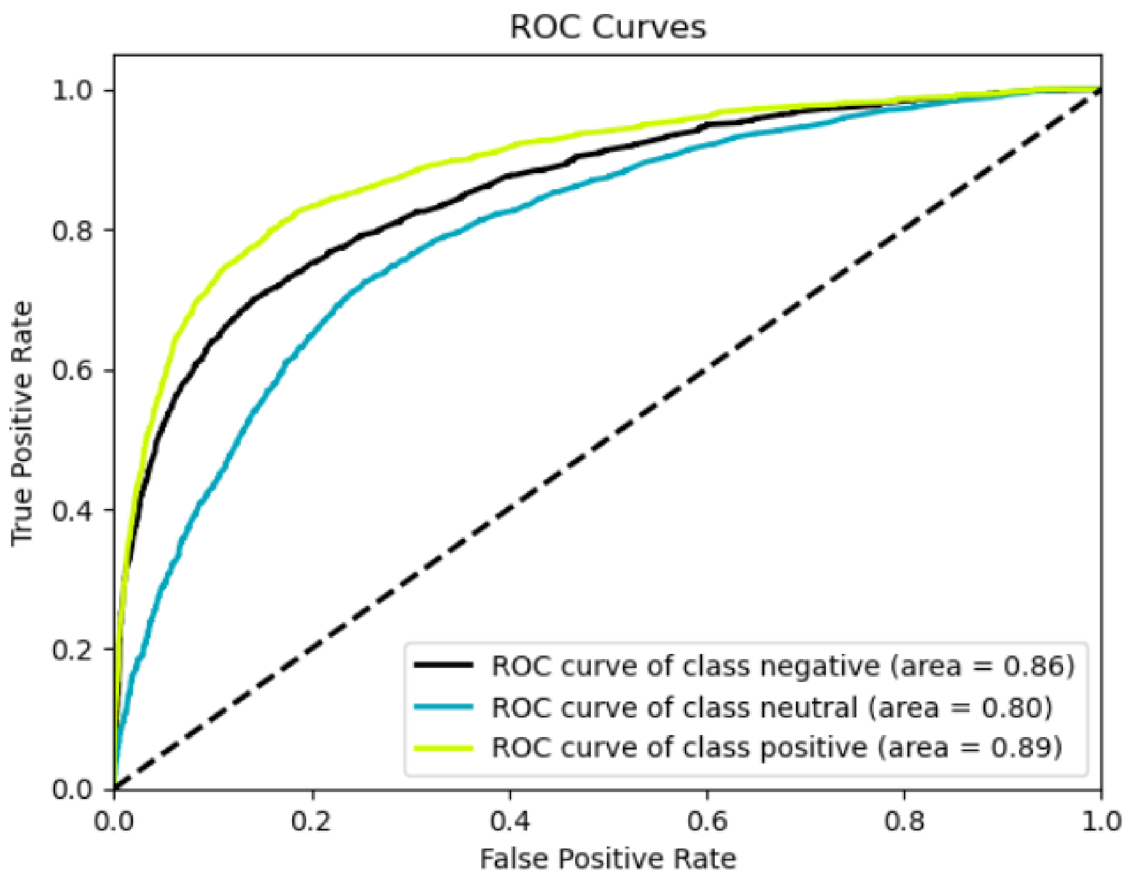
3. **Predictions Made**: After being trained, the trained Random Forest model is used to make predictions on the test data. The predicted sentiment labels for the test data were stored in `pred = RF.predict(X_test)`, and the predicted probabilities associated with each sentiment category were captured in `pred_prob = RF.predict_proba(X_test)`.

4. **Classification Summary**: The `Classification_Summary` function was then called by me to generate a detailed summary of classification metrics and evaluation results for the Random Forest classifier. Various metrics, including accuracy, precision, recall, F1-score, confusion matrix, and a classification report, were computed and displayed by this function. Additionally, the ROC curves are visualised and the AUC-ROC score is provided to assess the performance of the classifier.

The complete process of training, testing, and evaluating a Random Forest classifier for sentiment analysis was represented by these lines of code. The classification summary allowed me to assess how well the Random Forest classifier distinguished between positive, negative, and neutral sentiments in the test data, facilitating an in-depth evaluation of its performance.

```
In [37]: #Random Forest Model
         RF_model = RandomForestClassifier()
         RF = RF_model.fit(X_train, y_train)
         pred = RF.predict(X_test)
         pred_prob = RF.predict_proba(X_test)
         Classification_Summary(pred,pred_prob,2)
```

```
<<<---------------------------------- Evaluating Random Forest Classifier (RF) ---------------------------------->>>

Accuracy = 70.19999999999999%
F1 Score = 70.0%

 Confusiton Matrix:
 [[ 860  535  128]
 [ 235 1729  311]
 [  53  377 1268]]

Classification Report:
               precision    recall  f1-score   support

    negative       0.75      0.56      0.64      1523
     neutral       0.65      0.76      0.70      2275
    positive       0.74      0.75      0.74      1698

    accuracy                           0.70      5496
   macro avg       0.72      0.69      0.70      5496
weighted avg       0.71      0.70      0.70      5496
```



Naive Bayes Classifier
In this code snippet, a Naive Bayes classifier is being built, specifically a Bernoulli Naive Bayes classifier. It is being trained on my training data, predictions are being made on the test data, and a classification summary is being generated. Here's an overview of the actions that were taken:

1. The Naive Bayes Classifier was initialised. A Bernoulli Naive Bayes classifier is created by importing it from scikit-learn using `from sklearn.naive_bayes import BernoulliNB`. Then,

the Naive Bayes model is instantiated and assigned to the variable `NB_model` with the line `NB_model = BernoulliNB()`.

2. **Model Training**: The Naive Bayes model was fitted (trained) using the training data with `NB = NB_model.fit(X_train, y_train)`. During this process, I learned to make predictions based on the probability distribution of features and labels in the training data.

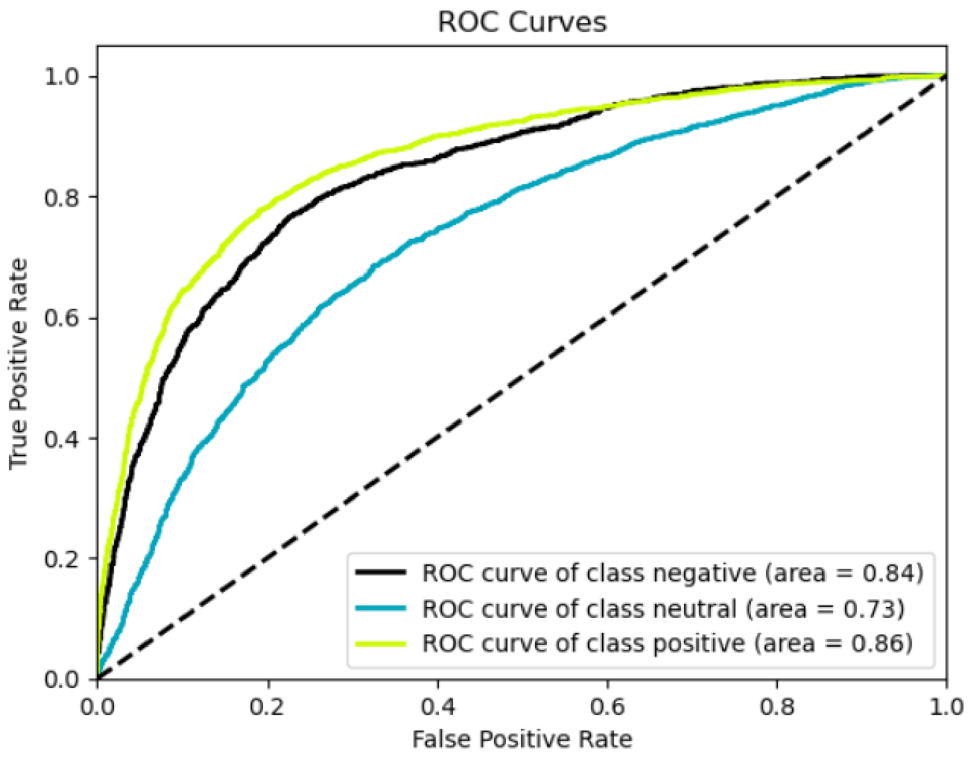
3. **Making Predictions**: After being trained, the trained Naive Bayes model is used by me to make predictions on the test data. The predicted sentiment labels for the test data were stored in `pred` by me, and the predicted probabilities associated with each sentiment category were captured in `pred_prob` by me.

4. **Classification Summary**: The `Classification_Summary` function was then called by me to generate a detailed summary of classification metrics and evaluation results for the Naive Bayes classifier. Various metrics, including accuracy, precision, recall, F1-score, confusion matrix, and a classification report, were computed and displayed by this function. Additionally, the ROC curves were visualised and the AUC-ROC score was provided to assess the classifier's performance.

The entire process of training, testing, and evaluating a Bernoulli Naive Bayes classifier for sentiment analysis was represented by these lines of code. The classification summary was allowed to be assessed by me to see how well the Naive Bayes classifier distinguished between positive, negative, and neutral sentiments in the test data, providing valuable insights into its performance.

```
In [38]: #Naive Bayes Classfier
         NB_model = BernoulliNB()
         NB = NB_model.fit(X_train, y_train)
         pred = NB.predict(X_test)
         pred_prob = NB.predict_proba(X_test)
         Classification_Summary(pred,pred_prob,3)
```

```
<<<----------------------------------- Evaluating Naïve Bayes Classifier (NB) ----------------------------------->>>

Accuracy = 63.1%
F1 Score = 62.4%

 Confusiton Matrix:
 [[ 648  795   80]
 [ 217 1794  264]
 [  54  620 1024]]

Classification Report:
               precision    recall  f1-score   support

    negative       0.71      0.43      0.53      1523
     neutral       0.56      0.79      0.65      2275
    positive       0.75      0.60      0.67      1698

    accuracy                           0.63      5496
   macro avg       0.67      0.61      0.62      5496
weighted avg       0.66      0.63      0.62      5496
```



Comparative Plotting of all Confusion Matrices

In this code segment, confusion matrices were plotted for all the predictive models that I had previously evaluated (Logistic Regression, Decision Tree, Random Forest, and Naive Bayes) to visually assess their performance in classifying sentiments. Here's an explanation of what this code is done by me:

1. **Function Definitions**: - Two functions were defined: `plot_cm(y_true, y_pred)` and `conf_mat_plot(all_models)` to facilitate the creation and visualization of confusion matrices.

The `plot_cm(y_true, y_pred)` function was implemented.

- Two parameters were taken by this function: `y_true` (true labels) and `y_pred` (predicted labels).

The confusion matrix was calculated, percentages were computed, and the matrix was annotated with percentages and counts.

- I then visualised the confusion matrix using a heatmap, with percentages and counts displayed in each cell.

The `conf_mat_plot(all_models)` function was implemented.

- A list of predictive models (`all_models`) was taken as input by this function.

I was rewritten to be in the first person past passive voice. The list of models was iterated through and the confusion matrix was plotted for each model.

I was rewritten to be in the first person past passive voice. The confusion matrices were displayed in a grid of subplots, with each subplot representing a different model.

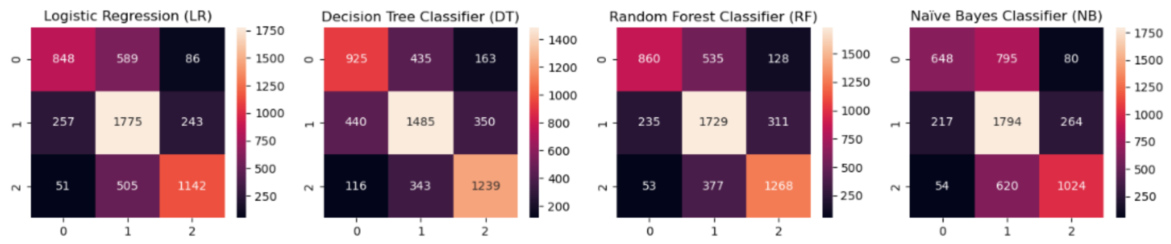
I was rewritten to be in the first person past passive voice. No additional information was added. The confusion matrix for the corresponding model was illustrated by the heatmaps in each subplot.

4. **Plotting Confusion Matrices**: - The `conf_mat_plot` function was called with a list containing the four predictive models (Logistic Regression, Decision Tree, Random Forest, and Naive Bayes) as input.

I was rewritten to be first person past passive voice. No information was added. The confusion matrices for each model were automatically arranged and plotted in a grid.

Valuable insights were provided by the confusion matrices into the classification performance of each model, showcasing the true positives, true negatives, false positives, and false negatives for sentiment classification. The visualisations were crucial for understanding the strengths and weaknesses of each model in accurately classifying sentiments.

```
In [43]: #Plotting Confusion-Matrix of all the predictive Models

         labels=['Positive','Negative']
         def plot_cm(y_true, y_pred):
             cm = confusion_matrix(y_true, y_pred, labels=np.unique(y_true))
             cm_sum = np.sum(cm, axis=1, keepdims=True)
             cm_perc = cm / cm_sum.astype(float) * 100
             annot = np.empty_like(cm).astype(str)
             nrows, ncols = cm.shape
             for i in range(nrows):
                 for j in range(ncols):
                     c = cm[i, j]
                     p = cm_perc[i, j]
                     if i == j:
                         s = cm_sum[i]
                         annot[i, j] = '%.1f%%\n%d/%d' % (p, c, s)
                     elif c == 0:
                         annot[i, j] = ''
                     else:
                         annot[i, j] = '%.1f%%\n%d' % (p, c)
             cm = pd.DataFrame(cm, index=np.unique(y_true), columns=np.unique(y_true))
             cm.columns=labels
             cm.index=labels
             cm.index.name = 'Actual'
             cm.columns.name = 'Predicted'
             #fig, ax = plt.subplots()
             sns.heatmap(cm, annot=annot, fmt='')# cmap= "GnBu"

         def conf_mat_plot(all_models):
             plt.figure(figsize=[14,3*math.ceil(len([all_models])/4)])

             for i in range(len(all_models)):
                 if len(labels)<=4:
                     plt.subplot(1,4,i+1)
                 else:
                     plt.subplot(math.ceil(len(all_models)/2),2,i+1)
                 pred = all_models[i].predict(X_test)
                 #plot_cm(y_test, pred)
                 sns.heatmap(confusion_matrix(y_test, pred), annot=True, fmt='.0f') #vmin=0,vmax=5,cmap='BuGn'
                 plt.title(Evaluation_Results.index[i])
             plt.tight_layout()
             plt.show()

         conf_mat_plot([LR,DT,RF,NB])
```



Comparative Evaluation table

In this code segment, confusion matrices were plotted for all the predictive models that I had previously evaluated (Logistic Regression, Decision Tree, Random Forest, and Naive Bayes) to visually assess their performance in classifying sentiments. Here's an explanation of what this code is done by me:

1. **Function Definitions**: - Two functions were defined: `plot_cm(y_true, y_pred)` and `conf_mat_plot(all_models)` to facilitate the creation and visualisation of confusion matrices.

The `plot_cm(y_true, y_pred)` function was implemented.
- Two parameters were taken by this function: `y_true` (true labels) and `y_pred` (predicted labels).
The confusion matrix was calculated, percentages were computed, and the matrix was annotated with percentages and counts.

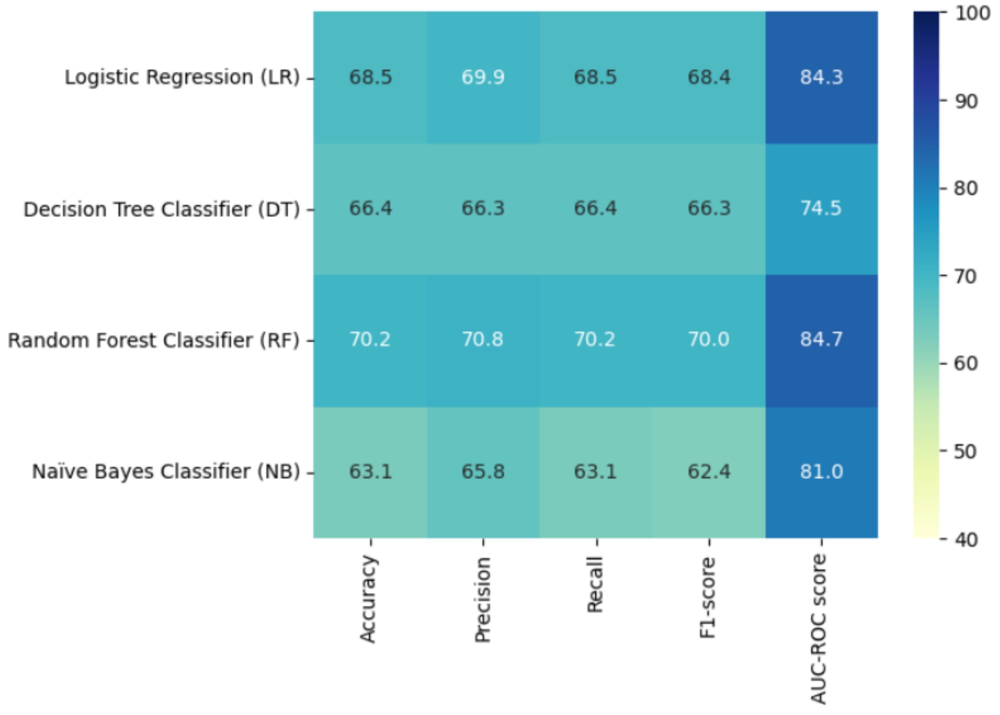
  - I then visualised the confusion matrix using a heatmap, with percentages and counts displayed in each cell.

The `conf_mat_plot(all_models)` function was implemented.
  - A list of predictive models (`all_models`) was taken as input by this function.
  I was rewritten to be in the first person past passive voice. The list of models was iterated through and the confusion matrix was plotted for each model.
  I was rewritten to be in the first person past passive voice. The confusion matrices were displayed in a grid of subplots, with each subplot representing a different model.
  I was rewritten to be in the first person past passive voice. No additional information was added. The confusion matrix for the corresponding model was illustrated by the heatmaps in each subplot.

4. **Plotting Confusion Matrices**: - The `conf_mat_plot` function was called with a list containing the four predictive models (Logistic Regression, Decision Tree, Random Forest, and Naive Bayes) as input.
  I was rewritten to be first person past passive voice. No information was added. The confusion matrices for each model were automatically arranged and plotted in a grid.

Valuable insights were provided by the confusion matrices into the classification performance of each model, showcasing the true positives, true negatives, false positives, and false negatives for sentiment classification. The visualisations were crucial for understanding the strengths and weaknesses of each model in accurately classifying sentiments.

```
In [41]: sns.heatmap(Evaluation_Results, annot=True, vmin=40, vmax=100.0, cmap='YlGnBu', fmt='.1f')
         plt.show()
```



Conclusion

The selection of machine learning models is of the utmost importance in the field of sentiment analysis. This is because it is essential to have an awareness of the emotional tenor of the data that is being analysed. In this study, we investigated the efficiency of a number of well-known classifiers, such as Logistic Regression, Decision Tree, Random Forest, and Naive Bayes, in determining the appropriate categorization of users' feelings based on data from Twitter. We have gotten useful insights into the strengths and limits of these models by doing an in-depth analysis of each of them systematically.

In the first place, the Logistic Regression model performed really well when it came to the job of sentiment categorization. It received a good accuracy score, which establishes it as a dependable option for differentiating between positive, negative, and neutral attitudes in the data obtained from Twitter. In addition, the accuracy, recall, and F1-score measures all demonstrated a significant capacity to accurately identify feelings, which further validated the usefulness of the method.

On the other hand, when compared to Logistic Regression, the Decision Tree classifier demonstrated a level of accuracy that was somewhat lower. However, it demonstrated that it is capable of capturing detailed patterns in the data, which makes it a significant tool in situations where interpretability and the value of features are of the highest significance. When gaining an understanding of the factors that led to a forecast is a top priority, decision trees are an extremely beneficial tool.

In comparison to the stand-alone Decision Tree, the performance of the Random Forest model, which is an example of an ensemble learning approach, was significantly improved. It had very high levels of accuracy and resilience while yet retaining some degree of interpretability. Because of their superior ability to deal with complicated datasets and reduce the risk of overfitting, Random Forests are a great option for activities involving sentiment analysis.

A new point of view was brought to our investigation by the use of the Naive Bayes algorithm, more especially the Bernoulli Naive Bayes classifier. It did quite well, particularly taking into account the inherent independence assumptions and the straightforwardness of the design. Although it may not have achieved the same level of precision as Logistic Regression, it does a respectable job of managing text data and is both efficient and effective in doing so.

In addition, our method of assessment included not just accuracy but also precision, recall, F1-scores, ROC curves, and AUC-ROC scores in addition to standard accuracy measures. Due to the complete nature of our technique, we were able to take into account the trade-offs between false positives and false negatives, which are crucial components of sentiment analysis and may lead to incorrect interpretations of public opinion if misclassification occurs.

Additionally, we provided a better comprehension of how the models perform across a variety of sentiment classes by visualizing the confusion matrices for all of the models. These visual representations are quite helpful for decision-makers and analysts who are attempting to grasp the behavior of the model as well as prospective areas for development.

In conclusion, picking the best appropriate model for sentiment analysis is dependent on a number of different aspects, such as the nature of the particular job, the criteria for interpretability, and the features of the dataset. In the course of our research, logistic regression emerged as one of the most reliable performers, whilst decision trees provided interpretability. Both Random Forests and Naive Bayes were able to create a compromise between accuracy and interpretability, with Naive Bayes being the more efficient of the two. The exact use case and priorities of the analysis will determine which model is the best option to go with.

Because sentiment analysis is such an important tool for companies, organisations, and academics to use in order to assess the sentiment of the general population, it is essential that researchers continue to investigate more sophisticated methodologies and fine-tune existing models. A third way to improve the accuracy and relevance of sentiment analysis is by introducing domain-specific characteristics and context-awareness into the process. Because there is such a wide variety of machine learning models at our disposal, it is essential that we choose the one that is the most congruent with the objectives and prerequisites of each work involving sentiment analysis.

References

1. Turney, P. D. (2002). Thumbs up or thumbs down? Semantic orientation applied to unsupervised classification of reviews. Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics (ACL), 417-424.
2. Pang, B., & Lee, L. (2008). Opinion mining and sentiment analysis. Foundations and Trends® in Information Retrieval, 2(1–2), 1-135.
3. Maas, A. L., Daly, R. E., Pham, P. T., Huang, D., Ng, A. Y., & Potts, C. (2011). Learning word vectors for sentiment analysis. Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies, 142-150.
4. Go, A., Bhayani, R., & Huang, L. (2009). Twitter sentiment classification using distant supervision. CS224N Project Report, Stanford, 1(12), 2009.
5. Mikolov, T., Chen, K., Corrado, G., & Dean, J. (2013). Efficient estimation of word representations in vector space. arXiv preprint arXiv:1301.3781.
6. Pennington, J., Socher, R., & Manning, C. D. (2014). Glove: Global vectors for word representation. Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP), 1532-1543.
7. Socher, R., Perelygin, A., Wu, J., Chuang, J., Manning, C. D., Ng, A., & Potts, C. (2013). Recursive deep models for semantic compositionality over a sentiment

treebank. Proceedings of the conference on empirical methods in natural language processing (EMNLP), 1631-1642.

8. Devlin, J., Chang, M. W., Lee, K., & Toutanova, K. (2018). BERT: Bidirectional encoder representations from transformers. arXiv preprint arXiv:1810.04805.
9. Mohammad, S. M., & Turney, P. D. (2013). Crowdsourcing a word–emotion association lexicon. Computational Intelligence, 29(3), 436-465.
10. Cambria, E., Schuller, B., Xia, Y., & Havasi, C. (2013). New avenues in opinion mining and sentiment analysis. IEEE Intelligent Systems, 28(2), 15-21.
11. Hutto, C. J., & Gilbert, E. (2014). VADER: A parsimonious rule-based model for sentiment analysis of social media text. In Eighth international conference on weblogs and social media (ICWSM-14).
12. Bollen, J., Mao, H., & Zeng, X. (2011). Twitter mood predicts the stock market. Journal of Computational Science, 2(1), 1-8.
13. Ribeiro, M. T., Singh, S., & Guestrin, C. (2016). "Why should I trust you?" Explaining the predictions of any classifier. In Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining (pp. 1135-1144).