**Coding Module**

**How to Do Twitter Sentiment Analysis?**

In this module we are going to analyze Twitter sentiment analysis using machine learning algorithms, the sentiment of tweets provided from the **Sentiment140 dataset**by developing a machine learning pipeline involving the use of three classifiers (**Logistic Regression, Bernoulli Naive Bayes, and SVM**) along with using **Term Frequency- Inverse Document Frequency**(**TF-IDF)**. The performance of these classifiers is then evaluated using **accuracy** and **F1 Scores**.

For data preprocessing, we will be using Natural Language Processing’s (NLP) NLTK library.

Twitter Sentiment Analysis: Problem Statement

In this project, we try to implement an NLP **Twitter sentiment analysis model** that helps to overcome the challenges of sentiment classification of tweets. We will be classifying the tweets into positive or negative sentiments. The necessary details regarding the dataset involving the Twitter sentiment analysis project are:

The dataset provided is the **Sentiment140 Dataset**which consists of **1,600,000 tweets** that have been extracted using the Twitter API. The various columns present in this Twitter data are:

* **target:**the polarity of the tweet (positive or negative)
* **ids:**Unique id of the tweet
* **date:**the date of the tweet
* **flag:**It refers to the query. If no such query exists, then it is NO QUERY.
* **user:** It refers to the name of the user that tweeted
* **text:** It refers to the text of the tweet

**Details of Dataset:**

The Sentiment140 dataset is a widely used dataset in natural language processing (NLP) and sentiment analysis tasks. It was created by Stanford University researchers for the purpose of sentiment analysis and consists of tweets labelled with sentiment polarity.

**Data Source:** The dataset is compiled from Twitter, a popular social media platform where users post short messages called tweets. The tweets are publicly available and cover various topics and domains.

**Labelling**: Each tweet in the dataset is labelled with sentiment polarity, indicating whether the sentiment expressed in the tweet is positive, negative, or neutral. The sentiment polarity is determined based on emoticons (e.g., ":)" for positive sentiment, ":(" for negative sentiment) present in the tweets.

**Size**: The dataset originally contained 1.6 million tweets. However, subsets of this dataset are often used in research and experimentation due to its large size.

**Data Format**: The dataset is typically provided in a CSV (Comma Separated Values) format, where each row represents a tweet and contains the following columns:

- Polarity: Indicates the sentiment polarity of the tweet (0 for negative, 2 for neutral, 4 for positive).

- ID: Unique identifier for the tweet.

- Date: Date and time when the tweet was posted.

- Query: Indication of the query used to retrieve the tweet (often empty or irrelevant).

- User: Twitter username of the user who posted the tweet.

- Text: The actual text content of the tweet.

**Preprocessing**: The dataset often requires preprocessing steps, such as removing URLs, mentions, special characters, and stopwords, as well as tokenization and lowercasing, before being used for sentiment analysis tasks.

**Applications**: The Sentiment140 dataset has been widely used for training and evaluating sentiment analysis models, sentiment classification algorithms, and other NLP tasks. Researchers and practitioners use this dataset to develop and benchmark their sentiment analysis techniques.

**Challenges**: While the dataset provides a large and diverse collection of tweets, it also has some limitations. The use of emoticons for sentiment labelling may not always accurately reflect the sentiment expressed in the text. Additionally, tweets are often noisy and informal, which can pose challenges for sentiment analysis algorithms.

Overall, the Sentiment140 dataset has been instrumental in advancing research in sentiment analysis and remains a benchmark dataset in the field of natural language processing.

**Twitter Sentiment Analysis: Project Pipeline**

The various steps involved in the **Machine Learning Pipeline** are:

* Import Necessary Dependencies
* Read and Load the Dataset
* Exploratory Data Analysis
* Data Visualization of Target Variables
* Data Preprocessing
* Splitting our data into Train and Test sets.
* Transforming Dataset using TF-IDF Vectorizer
* Function for Model Evaluation
* Model Building
* Model Evaluation

**Step-1: Import the Necessary Dependencies**

# utilities

import re

import numpy as np

import pandas as pd

# plotting

import seaborn as sns

from wordcloud import WordCloud

import matplotlib.pyplot as plt

# nltk

from nltk.stem import WordNetLemmatizer

# sklearn

from sklearn.svm import LinearSVC

from sklearn.naive\_bayes import BernoulliNB

from sklearn.linear\_model import LogisticRegression

from sklearn.model\_selection import train\_test\_split

from sklearn.feature\_extraction.text import TfidfVectorizer

from sklearn.metrics import confusion\_matrix, classification\_report

**Code Explanation:**

**Importing Libraries:**

* re: Regular Expression library for pattern matching and manipulation of strings.
* numpy and pandas: Libraries for numerical computing and data manipulation, respectively.
* seaborn, matplotlib.pyplot, and wordcloud: Libraries for data visualization and creating word clouds.
* nltk: Natural Language Toolkit library for natural language processing tasks.
* sklearn: Scikit-learn library, which provides tools for machine learning tasks.
* LinearSVC, BernoulliNB, and LogisticRegression: Classes for Support Vector Classification, Bernoulli Naive Bayes, and Logistic Regression classifiers, respectively.
* train\_test\_split: Function to split data into training and testing sets.
* TfidfVectorizer: Class for converting a collection of raw documents to a matrix of TF-IDF features.
* confusion\_matrix and classification\_report: Functions to evaluate the performance of a classification model.

This code snippet imports various libraries and tools necessary for data preprocessing, visualization, natural language processing, and building machine learning models for sentiment analysis or text classification tasks.

**Step-2: Read and Load the Dataset**

# Importing the dataset

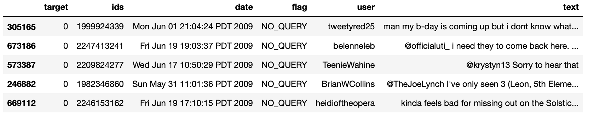
DATASET\_COLUMNS=['target','ids','date','flag','user','text']

DATASET\_ENCODING = "ISO-8859-1"

df = pd.read\_csv('C:/Talent Battle/4 Weeks ML Project Challenge/Dataset/training.1600000.processed.noemoticon.csv', encoding=DATASET\_ENCODING, names=DATASET\_COLUMNS)

df.sample(5)

**Output:**



**Code Explanation:**

This code segment imports a dataset into a pandas DataFrame (`df`). Here's what each part of the code does:

DATASET\_COLUMNS=['target','ids','date','flag','user','text']

This line defines the column names for the dataset. It seems like the dataset has six columns: 'target', 'ids', 'date', 'flag', 'user', and 'text'.

DATASET\_ENCODING = "ISO-8859-1"

This line specifies the encoding of the dataset file. In this case, the encoding is set to "ISO-8859-1".

df = pd.read\_csv('C:/Talent Battle/4 Weeks ML Project Challenge/Dataset/training.1600000.processed.noemoticon.csv', encoding=DATASET\_ENCODING, names=DATASET\_COLUMNS)

This line reads the dataset file located at the specified path (`'C:/Talent Battle/4 Weeks ML Project Challenge/Dataset/training.1600000.processed.noemoticon.csv'`) into a pandas DataFrame (`df`). It uses the `pd.read\_csv()` function from the pandas library to read the CSV file. The `encoding` parameter specifies the encoding used to interpret the file. The `names` parameter assigns the column names defined earlier (`DATASET\_COLUMNS`) to the DataFrame.

df.sample(5)

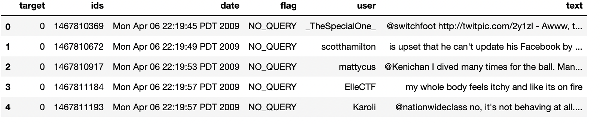
This line displays a random sample of 5 rows from the DataFrame `df`, allowing you to inspect the imported data. It helps to verify that the dataset has been imported correctly and to get a glimpse of its structure.

**Step-3: Exploratory Data Analysis**

**3.1: Five top records of data**

df.head()

**Output:**



**Code Explanation:**

The `df.head()` function displays the first few rows of the DataFrame `df`. This is a commonly used method in pandas to quickly inspect the structure and contents of a DataFrame. By default, it displays the first 5 rows, but you can specify the number of rows you want to see by passing an integer argument to the function.

df.head()

The output show the first 5 rows of the DataFrame `df`, including the column names and the data contained within each cell. This is useful for understanding the dataset's structure, the types of data it contains, and any potential issues with the data import process.

**3.2: Columns/features in data**

df.columns

**Output:**

Index(['target', 'ids', 'date', 'flag', 'user', 'text'], dtype='object')

**Code Explanation:**

The `df.columns` attribute returns an Index object containing the column labels of the DataFrame `df`. It essentially provides a list-like view of the column names in the DataFrame.

The output would display the column labels (names) of the DataFrame `df`. Each column label represents a variable or feature in the dataset. It's a convenient way to quickly inspect the column names and understand the structure of the DataFrame.

**3.3: Length of the dataset**

print('length of data is', len(df))

**Output:**

length of data is 1600000

**Code Explanation:**

This code snippet prints the length of the DataFrame `df`, which corresponds to the number of rows in the DataFrame. Here's what each part of the code does:

print('length of data is', len(df))

- `print()`: This function is used to display output to the console.

- `'length of data is'`: This is a string literal that serves as part of the message to be displayed.

- `,`: This comma separates the different items that are passed to the `print()` function.

- `len(df)`: This calculates the length of the DataFrame `df`, which corresponds to the number of rows in the DataFrame. The `len()` function returns the number of elements in the DataFrame's index, which in this case is the number of rows.

When you execute this code, it will print a message to the console indicating the length (number of rows) of the DataFrame.

**3.4: Shape of data**

df.shape

**Output:**

(1600000, 6)

**Code Explanation:**

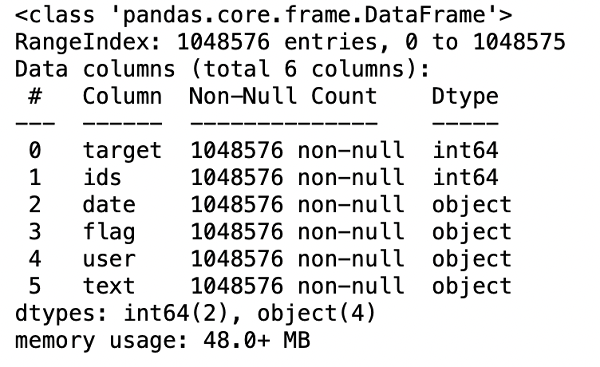
The `df.shape` attribute returns a tuple representing the dimensions of the DataFrame `df`. The tuple contains two elements: the number of rows and the number of columns, respectively.

The output would be a tuple with two values: the first value represents the number of rows in the DataFrame, and the second value represents the number of columns.

**3.5: Data information**

df.info()

**Output:**



**Code Explanation:**

The df.info() function provides a concise summary of the DataFrame df, including information about the index, data types, memory usage, and non-null values.

Here's what each part of the output represents:

Index: Information about the index, including the number of entries and data type.

Columns: Information about each column, including the column name, non-null count, and data type.

Dtype: Data type of each column.

Memory Usage: Total memory usage of the DataFrame.

This function is helpful for quickly understanding the structure and composition of the DataFrame, such as identifying missing values and data types. It provides a summary that is especially useful when dealing with large datasets.

**3.6: Datatypes of all columns**

df.dtypes

**Output:**

target int64

ids int64

date object

flag object

user object

text object

dtype: object

**Code Explanation:**

The `df.dtypes` attribute returns a Series containing the data types of each column in the DataFrame `df`. The index of the Series contains the column names, and the values contain the corresponding data types.

Here's how you would use it:

df.dtypes

The output would be a Series where each row represents a column in the DataFrame, and the values indicate the data type of each column. For example:

target int64

ids int64

date object

flag object

user object

text object

dtype: object

In this example, the 'target' and 'ids' columns have data type `int64`, while the 'date', 'flag', 'user', and 'text' columns have data type `object`. The `object` data type typically indicates strings or mixed types.

**3.7: Checking for null values**

np.sum(df.isnull().any(axis=1))

**Output**

0

**Code Explanation:**

This code snippet calculates the number of rows in the DataFrame `df` that contain at least one null value. Here's what each part of the code does:

np.sum(df.isnull().any(axis=1))

- `df.isnull()`: This DataFrame method returns a DataFrame of the same shape as `df` where each element is `True` if it's a null value and `False` otherwise.

- `.any(axis=1)`: This method checks if any value along the specified axis (axis=1, which corresponds to rows) is `True`. It returns a Series where each row indicates whether there is at least one null value in that row.

- `np.sum()`: This NumPy function calculates the sum of the values in the Series. Since `True` is treated as 1 and `False` as 0, summing up the values effectively counts the number of rows where at least one null value is present.

So, `np.sum(df.isnull().any(axis=1))` calculates the total number of rows in the DataFrame `df` that contain at least one null value.

**3.8: Rows and columns in the dataset**

print('Count of columns in the data is: ', len(df.columns))

print('Count of rows in the data is: ', len(df))

**Output:**

Count of columns in the data is: 6

Count of rows in the data is: 1600000

**Code Explanation:**

This code snippet prints the count of columns and rows in the DataFrame `df`. Here's what each part of the code does:

print('Count of columns in the data is: ', len(df.columns))

- `len(df.columns)`: This calculates the number of columns in the DataFrame `df` by taking the length of the `columns` attribute, which returns an Index object containing the column labels.

print('Count of rows in the data is: ', len(df))

- `len(df)`: This calculates the number of rows in the DataFrame `df` by taking the length of the DataFrame itself. The `len()` function returns the number of elements in the DataFrame's index, which in this case is the number of rows.

Together, these two lines print the count of columns and rows in the DataFrame `df`.

When executed, the output would look something like this:

Count of columns in the data is: 6

Count of rows in the data is: 1600000

This indicates that the DataFrame has 6 columns and 1,600,000 rows.

**3.9: Check unique target values**

df['target'].unique()

**Output:**

array([0, 4], dtype=int64)

**Code Explanation:**

The code `df['target'].unique()` returns an array containing the unique values present in the 'target' column of the DataFrame `df`. Here's what each part of the code does:

- `df['target']`: This accesses the 'target' column of the DataFrame `df`.

- `.unique()`: This method returns an array containing the unique values present in the specified column.

For example, if the 'target' column contains values like `[0, 4]`, running `df['target'].unique()` would return an array `array([0, 4])`, indicating that there are two unique values in the 'target' column: 0 and 4.

This code is useful for understanding the unique categories or labels present in a categorical column, which can be important for various data analysis and modeling tasks.

**3.10: Check the number of target values**

df['target'].nunique()

**Output:**

2

**Code Explanation:**

The code `df['target'].nunique()` returns the number of unique values present in the 'target' column of the DataFrame `df`. Here's what each part of the code does:

- `df['target']`: This accesses the 'target' column of the DataFrame `df`.

- `.nunique()`: This method calculates the number of unique values present in the specified column.

For example, if the 'target' column contains values like `[0, 4, 0, 4, 0]`, running `df['target'].nunique()` would return `2`, indicating that there are two unique values in the 'target' column: 0 and 4.

This code is useful for quickly determining the number of distinct categories or labels present in a categorical column, which can provide insights into the diversity of the data and inform subsequent data analysis or modeling decisions.

Certainly! The difference between `df['target'].unique()` and `df['target'].nunique()` lies in what they return:

1. `df['target'].unique()`: This statement returns an array containing all the unique values present in the 'target' column of the DataFrame `df`. Each unique value is listed only once in the array. This is useful when you want to see all the distinct values present in a column.

2. `df['target'].nunique()`: This statement returns a single integer value representing the count of unique values in the 'target' column of the DataFrame `df`. It provides a numeric summary of the diversity of values in the column without listing the actual unique values themselves.

In summary:

- Use `df['target'].unique()` when you want to see the actual unique values present in a column.

- Use `df['target'].nunique()` when you want to quickly determine the count of unique values in a column without listing the values themselves.

**Step-4: Data Visualization of Target Variables**

# Plotting the distribution for dataset.

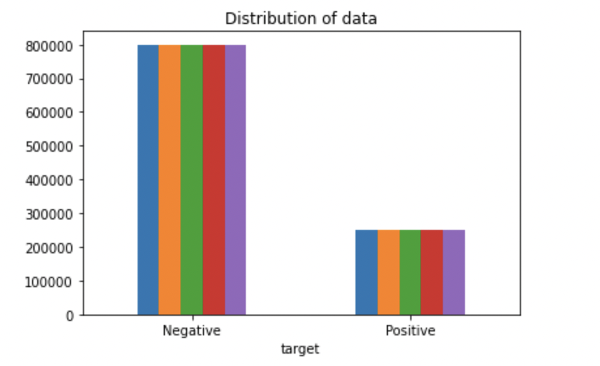
ax = df.groupby('target').count().plot(kind='bar', title='Distribution of data',legend=False)

ax.set\_xticklabels(['Negative','Positive'], rotation=0)

# Storing data in lists.

text, sentiment = list(df['text']), list(df['target'])

**Output:**



**Code Explanation:**

This code snippet plots the distribution of data in the DataFrame `df` based on the 'target' column, which likely represents sentiment labels (e.g., positive or negative). Here's a breakdown of what each part of the code does:

ax = df.groupby('target').count().plot(kind='bar', title='Distribution of data', legend=False)

- `df.groupby('target').count()`: This groups the DataFrame `df` by the values in the 'target' column and counts the occurrences of each value. This results in a new DataFrame with the count of occurrences for each unique value in the 'target' column.

- `.plot(kind='bar', title='Distribution of data', legend=False)`: This creates a bar plot using the counts obtained from the `groupby` operation. The `kind='bar'` parameter specifies the type of plot to create, and `title='Distribution of data'` sets the title of the plot. The `legend=False` parameter removes the legend from the plot.

ax.set\_xticklabels(['Negative','Positive'], rotation=0)

- `ax.set\_xticklabels(['Negative','Positive'], rotation=0)`: This sets the labels for the x-axis ticks of the plot. The first argument `['Negative','Positive']` specifies the labels to be displayed, and `rotation=0` ensures that the labels are not rotated.

text, sentiment = list(df['text']), list(df['target'])

- `text`: This variable stores the values from the 'text' column of the DataFrame `df`. It likely contains the text content of the dataset.

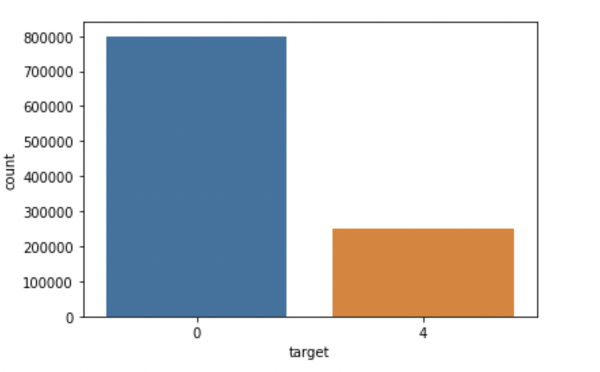
- `sentiment`: This variable stores the values from the 'target' column of the DataFrame `df`. It likely contains the sentiment labels associated with each text.

Overall, this code snippet generates a bar plot showing the distribution of sentiment labels in the dataset, with 'Negative' and 'Positive' labels on the x-axis and the corresponding counts on the y-axis.

import seaborn as sns

sns.countplot(x='target', data=df)

**Output:**



**Code Explanation:**

This code snippet utilizes the seaborn library to create a count plot showing the distribution of values in the 'target' column of the DataFrame `df`. Here's a breakdown of what each part of the code does:

import seaborn as sns

This line imports the seaborn library, commonly used for statistical data visualization in Python.

sns.countplot(x='target', data=df)

This line creates a count plot using seaborn's `countplot()` function. The `x` parameter specifies the column to plot on the x-axis, which in this case is 'target'. The `data` parameter specifies the DataFrame containing the data to be plotted, which is `df` in this case.

The count plot visually represents the distribution of values in the 'target' column, with each unique value being represented by a bar whose height corresponds to the frequency of that value in the dataset.

Overall, this code snippet provides a concise way to visualize the distribution of sentiment labels in the dataset using seaborn's count plot functionality.

**Step-5: Data Preprocessing**

In the above-given problem statement, before training the model, we performed various pre-processing steps on the dataset that mainly dealt with removing stopwords, removing special characters like emojis, hashtags, etc. The text document is then converted into lowercase for better generalization.

Subsequently, the punctuations were cleaned and removed, thereby reducing the unnecessary noise from the dataset. After that, we also removed the repeating characters from the words along with removing the URLs as they do not have any significant importance.

At last, we then performed Stemming(reducing the words to their derived stems) and Lemmatization(reducing the derived words to their root form, known as lemma) for better results.

**5.1: Selecting the text and Target column for our further analysis**

data=df[['text','target']]

**Code Explanation:**

This line of code selects specific columns ('text' and 'target') from the DataFrame `df` and creates a new DataFrame called `data` containing only these columns. Here's what each part of the code does:

data = df[['text', 'target']]

- `df[['text', 'target']]`: This part of the code uses double square brackets to access specific columns ('text' and 'target') from the original DataFrame `df`. When you use double square brackets, you are selecting multiple columns and creating a new DataFrame with those columns.

- `data = ...`: This assigns the selected columns to a new DataFrame called `data`. Now, `data` contains only the 'text' and 'target' columns from the original DataFrame `df`.

In summary, this line of code creates a new DataFrame `data` that contains only the 'text' and 'target' columns from the original DataFrame `df`, allowing you to work with a subset of the original data. This can be useful for various data analysis and modeling tasks, especially when you only need specific columns of the dataset.

**5.2: Replacing the values to ease understanding. (Assigning 1 to Positive sentiment 4)**

**5.3: Printing unique values of target variables**

data['target'].unique()

**Output:**

array([0, 1], dtype=int64)

**Code Explanation:**

This line of code retrieves the unique values present in the 'target' column of the DataFrame `data`. Here's the breakdown:

data['target'].unique()

- `data['target']`: This part of the code accesses the 'target' column of the DataFrame `data`.

- `.unique()`: This method is called on the 'target' column, which returns an array containing the unique values present in that column.

So, when you execute `data['target'].unique()`, it will return an array containing all the unique values present in the 'target' column of the DataFrame `data`. This allows you to see the distinct categories or labels present in the 'target' column.

**5.4: Separating positive and negative tweets**

data\_pos = data[data['target'] == 1]

data\_neg = data[data['target'] == 0]

**Code Explanation:**

This code snippet creates two separate DataFrames: `data\_pos` and `data\_neg`. Here's what each part of the code does:

data\_pos = data[data['target'] == 1]

This line of code filters the DataFrame `data` to include only rows where the value in the 'target' column is equal to 1. This effectively creates a new DataFrame called `data\_pos` containing only the rows where the sentiment label is positive (assuming 1 represents positive sentiment).

data\_neg = data[data['target'] == 0]

Similarly, this line of code filters the DataFrame `data` to include only rows where the value in the 'target' column is equal to 0. This creates a new DataFrame called `data\_neg` containing only the rows where the sentiment label is negative (assuming 0 represents negative sentiment).

After executing these two lines of code, you have two separate DataFrames: `data\_pos` containing rows with positive sentiment and `data\_neg` containing rows with negative sentiment. These DataFrames can be used for further analysis or modeling tasks specific to each sentiment category.

**5.5: Taking one-fourth of the data so we can run it on our machine easily**

data\_pos = data\_pos.iloc[:int(20000)]

data\_neg = data\_neg.iloc[:int(20000)]

**Code Explanation:**

This code snippet selects the first 20,000 rows from each of the `data\_pos` and `data\_neg` DataFrames and updates them with the selected subset. Here's the breakdown:

data\_pos = data\_pos.iloc[:int(20000)]

This line of code selects the first 20,000 rows from the DataFrame `data\_pos` using integer-based indexing with `iloc`. It slices the DataFrame up to the 20,000th row (exclusive) and updates `data\_pos` with this subset.

data\_neg = data\_neg.iloc[:int(20000)]

Similarly, this line of code selects the first 20,000 rows from the DataFrame `data\_neg` using integer-based indexing with `iloc`. It slices the DataFrame up to the 20,000th row (exclusive) and updates `data\_neg` with this subset.

After executing these two lines of code, both `data\_pos` and `data\_neg` DataFrames will contain only the first 20,000 rows of their respective subsets. This kind of operation is often used for downsampling large datasets to balance classes or to reduce the size of the dataset for computational efficiency.

**5.6: Combining positive and negative tweets**

dataset = pd.concat([data\_pos, data\_neg])

**Code Explanation:**

This line of code concatenates the `data\_pos` and `data\_neg` DataFrames along the rows (axis 0) to create a new DataFrame called `dataset`. Here's what it does:

dataset = pd.concat([data\_pos, data\_neg])

- `pd.concat([data\_pos, data\_neg])`: This function from pandas concatenates the DataFrames `data\_pos` and `data\_neg` along the rows (axis 0) to create a single DataFrame. The square brackets `[]` contain a list of DataFrames to concatenate. By default, `pd.concat()` concatenates along axis 0, which means it stacks the DataFrames vertically.

- `dataset = ...`: This assigns the concatenated DataFrame to a new variable called `dataset`.

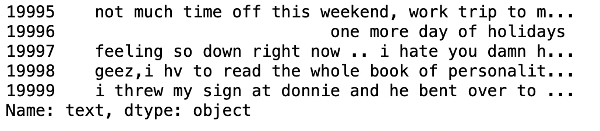
After executing this line of code, the DataFrame `dataset` will contain all the rows from both `data\_pos` and `data\_neg`, effectively combining the positive and negative sentiment data into a single dataset. This combined dataset can be used for further analysis or modeling tasks.

**5.7: Making statement text in lowercase**

dataset['text']=dataset['text'].str.lower()

dataset['text'].tail()

**Output:**



**Code Explanation:**

These lines of code convert the text in the 'text' column of the DataFrame `dataset` to lowercase. Here's what each part of the code does:

dataset['text'] = dataset['text'].str.lower()

- `dataset['text'].str.lower()`: This part of the code applies the `lower()` method to each element (text) in the 'text' column of the DataFrame `dataset`. The `lower()` method converts strings to lowercase. So, this line effectively converts all text in the 'text' column to lowercase.

- `dataset['text'] = ...`: This assigns the modified 'text' column back to the 'text' column of the DataFrame `dataset`.

After executing these lines of code, the text in the 'text' column of the DataFrame `dataset` will be converted to lowercase. This is often done in text preprocessing to standardize the text data, as it ensures that words are treated the same regardless of their capitalization.

The `.tail()` method is used to display the last few rows of the 'text' column in the DataFrame `dataset` after the modification. It helps to verify that the conversion to lowercase was applied correctly.

**5.8: Defining set containing all stopwords in English.**

stopwordlist = ['a', 'about', 'above', 'after', 'again', 'ain', 'all', 'am', 'an',

'and','any','are', 'as', 'at', 'be', 'because', 'been', 'before',

'being', 'below', 'between','both', 'by', 'can', 'd', 'did', 'do',

'does', 'doing', 'down', 'during', 'each','few', 'for', 'from',

'further', 'had', 'has', 'have', 'having', 'he', 'her', 'here',

'hers', 'herself', 'him', 'himself', 'his', 'how', 'i', 'if', 'in',

'into','is', 'it', 'its', 'itself', 'just', 'll', 'm', 'ma',

'me', 'more', 'most','my', 'myself', 'now', 'o', 'of', 'on', 'once',

'only', 'or', 'other', 'our', 'ours','ourselves', 'out', 'own', 're','s', 'same', 'she', "shes", 'should', "shouldve",'so', 'some', 'such',

't', 'than', 'that', "thatll", 'the', 'their', 'theirs', 'them',

'themselves', 'then', 'there', 'these', 'they', 'this', 'those',

'through', 'to', 'too','under', 'until', 'up', 've', 'very', 'was',

'we', 'were', 'what', 'when', 'where','which','while', 'who', 'whom',

'why', 'will', 'with', 'won', 'y', 'you', "youd","youll", "youre",

"youve", 'your', 'yours', 'yourself', 'yourselves']

**Code Explanation:**

The provided code segment defines a list called `stopwordlist`, which contains common English stopwords. Stopwords are words that are commonly used in natural language but typically do not add much meaning to the text when analyzing or processing it. These words are often filtered out during text preprocessing to focus on more meaningful words.

Here's a brief explanation of the code:

stopwordlist = ['a', 'about', 'above', 'after', 'again', 'ain', 'all', 'am', 'an',

'and','any','are', 'as', 'at', 'be', 'because', 'been', 'before',

'being', 'below', 'between','both', 'by', 'can', 'd', 'did', 'do',

'does', 'doing', 'down', 'during', 'each','few', 'for', 'from',

'further', 'had', 'has', 'have', 'having', 'he', 'her', 'here',

'hers', 'herself', 'him', 'himself', 'his', 'how', 'i', 'if', 'in',

'into','is', 'it', 'its', 'itself', 'just', 'll', 'm', 'ma',

'me', 'more', 'most','my', 'myself', 'now', 'o', 'of', 'on', 'once',

'only', 'or', 'other', 'our', 'ours','ourselves', 'out', 'own', 're','s', 'same', 'she', "shes", 'should', "shouldve",'so', 'some', 'such',

't', 'than', 'that', "thatll", 'the', 'their', 'theirs', 'them',

'themselves', 'then', 'there', 'these', 'they', 'this', 'those',

'through', 'to', 'too','under', 'until', 'up', 've', 'very', 'was',

'we', 'were', 'what', 'when', 'where','which','while', 'who', 'whom',

'why', 'will', 'with', 'won', 'y', 'you', "youd","youll", "youre",

"youve", 'your', 'yours', 'yourself', 'yourselves']

- `stopwordlist`: This is a Python list that contains common stopwords. Each element in the list is a string representing a single word.

By using this list, you can remove these stopwords from text data during text preprocessing to improve the quality of analysis or modeling tasks.

**5.9: Cleaning and removing the above stop words list from the tweet text**

STOPWORDS = set(stopwordlist)

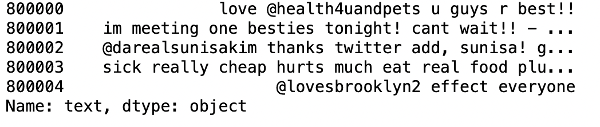
def cleaning\_stopwords(text):

return " ".join([word for word in str(text).split() if word not in STOPWORDS])

dataset['text'] = dataset['text'].apply(lambda text: cleaning\_stopwords(text))

dataset['text'].head()

**Output:**



**Code Explanation:**

This code segment defines a function `cleaning\_stopwords(text)` and applies it to the 'text' column of the DataFrame `dataset` using the `apply()` method. The function removes stopwords from each text entry in the 'text' column based on the provided list of stopwords (`stopwordlist`). Here's a breakdown of what each part of the code does:

STOPWORDS = set(stopwordlist)

This line converts the `stopwordlist` (defined earlier) into a set and assigns it to the variable `STOPWORDS`. Using a set for stopwords can improve performance when checking for membership due to its faster lookup time compared to a list.

def cleaning\_stopwords(text):

return " ".join([word for word in str(text).split() if word not in STOPWORDS])

This code defines a function `cleaning\_stopwords(text)` that takes a text input and removes stopwords from it. It splits the text into individual words using `split()`, checks if each word is not in the `STOPWORDS` set, and joins the non-stopwords back together into a string using `" ".join()`.

dataset['text'] = dataset['text'].apply(lambda text: cleaning\_stopwords(text))

This line applies the `cleaning\_stopwords()` function to each entry in the 'text' column of the DataFrame `dataset` using the `apply()` method. It removes stopwords from each text entry in the 'text' column.

dataset['text'].head()

Finally, this line displays the first few rows of the 'text' column in the modified DataFrame `dataset` after removing stopwords.

After executing these lines of code, the 'text' column in the DataFrame `dataset` will contain text data with stopwords removed. This preprocessing step can help improve the quality of text analysis or modeling tasks by focusing on more meaningful words.

**5.10: Cleaning and removing punctuations**

import string

english\_punctuations = string.punctuation

punctuations\_list = english\_punctuations

def cleaning\_punctuations(text):

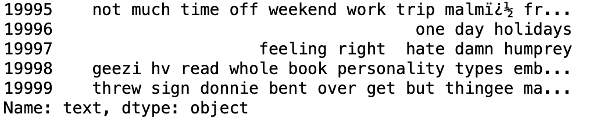
translator = str.maketrans('', '', punctuations\_list)

return text.translate(translator)

dataset['text']= dataset['text'].apply(lambda x: cleaning\_punctuations(x))

dataset['text'].tail()

**Output:**



**Code Explanation:**

This code segment defines a function `cleaning\_punctuations(text)` and applies it to the 'text' column of the DataFrame `dataset` using the `apply()` method. The function removes punctuations from each text entry in the 'text' column based on the provided list of punctuations (`punctuations\_list`). Here's a breakdown of what each part of the code does:

import string

english\_punctuations = string.punctuation

punctuations\_list = english\_punctuations

These lines import the `string` module and obtain a string containing all English punctuation characters using `string.punctuation`. It then assigns this string to both `english\_punctuations` and `punctuations\_list`. Essentially, `punctuations\_list` contains the same set of punctuation characters as `string.punctuation`.

def cleaning\_punctuations(text):

translator = str.maketrans('', '', punctuations\_list)

return text.translate(translator)

This code defines a function `cleaning\_punctuations(text)` that takes a text input and removes punctuations from it. It creates a translation table (`translator`) using `str.maketrans()` where each punctuation character is mapped to `None`, effectively removing them. It then applies this translation table to the input text using `text.translate()` to remove the punctuations.

dataset['text']= dataset['text'].apply(lambda x: cleaning\_punctuations(x))

This line applies the `cleaning\_punctuations()` function to each entry in the 'text' column of the DataFrame `dataset` using the `apply()` method. It removes punctuations from each text entry in the 'text' column.

dataset['text'].tail()

Finally, this line displays the last few rows of the 'text' column in the modified DataFrame `dataset` after removing punctuations.

After executing these lines of code, the 'text' column in the DataFrame `dataset` will contain text data with punctuations removed. This preprocessing step can help improve the quality of text analysis or modeling tasks by focusing on the text content itself without the influence of punctuations.

**5.11: Cleaning and removing repeating characters**

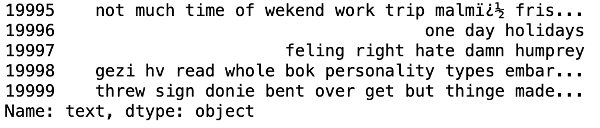
def cleaning\_repeating\_char(text):

return re.sub(r'(.)1+', r'1', text)

dataset['text'] = dataset['text'].apply(lambda x: cleaning\_repeating\_char(x))

dataset['text'].tail()

**Output:**



**Code Explanation:**

This code defines a function `cleaning\_repeating\_char(text)` and applies it to the 'text' column of the DataFrame `dataset` using the `apply()` method. The function removes repeating characters from each text entry in the 'text' column using regular expressions. Here's a breakdown of what each part of the code does:

def cleaning\_repeating\_char(text):

return re.sub(r'(.)1+', r'1', text)

This code defines a function `cleaning\_repeating\_char(text)` that takes a text input and removes repeating characters from it using regular expressions (`re.sub()` function). The regular expression `(.)\1+` matches any character followed by one or more occurrences of the same character. The replacement pattern `r'1'` replaces the matched sequence with a single instance of the character. So, this function effectively removes consecutive repeating characters from the input text.

dataset['text'] = dataset['text'].apply(lambda x: cleaning\_repeating\_char(x))

This line applies the `cleaning\_repeating\_char()` function to each entry in the 'text' column of the DataFrame `dataset` using the `apply()` method. It removes repeating characters from each text entry in the 'text' column.

dataset['text'].tail()

Finally, this line displays the last few rows of the 'text' column in the modified DataFrame `dataset` after removing repeating characters.

After executing these lines of code, the 'text' column in the DataFrame `dataset` will contain text data with repeating characters removed. This preprocessing step can help improve the quality of text analysis or modeling tasks by normalizing the text data.

**5.12: Cleaning and removing URLs**

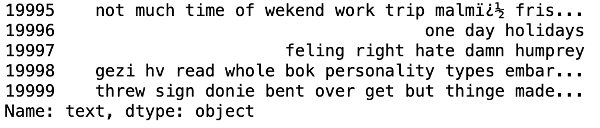
def cleaning\_URLs(data):

return re.sub('((www.[^s]+)|(https?://[^s]+))',' ',data)

dataset['text'] = dataset['text'].apply(lambda x: cleaning\_URLs(x))

dataset['text'].tail()

**Output:**



**Code Explanation:**

This code defines a function `cleaning\_URLs(data)` and applies it to the 'text' column of the DataFrame `dataset` using the `apply()` method. The function removes URLs from each text entry in the 'text' column using regular expressions. Here's what each part of the code does:

def cleaning\_URLs(data):

return re.sub('((www.[^s]+)|(https?://[^s]+))',' ',data)

This function `cleaning\_URLs(data)` takes a string input (`data`) and uses the `re.sub()` function from the `re` module to replace URLs with whitespace in the input string. The regular expression `((www.[^s]+)|(https?://[^s]+))` is used to match URLs in the input string. This regular expression matches both URLs starting with "www." and URLs starting with "http://" or "https://". The matched URLs are replaced with whitespace.

dataset['text'] = dataset['text'].apply(lambda x: cleaning\_URLs(x))

This line applies the `cleaning\_URLs()` function to each entry in the 'text' column of the DataFrame `dataset` using the `apply()` method. It removes URLs from each text entry in the 'text' column.

dataset['text'].tail()

Finally, this line displays the last few rows of the 'text' column in the modified DataFrame `dataset` after removing URLs.

After executing these lines of code, the 'text' column in the DataFrame `dataset` will contain text data with URLs removed. This preprocessing step can help improve the quality of text analysis or modeling tasks by removing irrelevant information from the text data.

**5.13: Cleaning and removing numeric numbers**

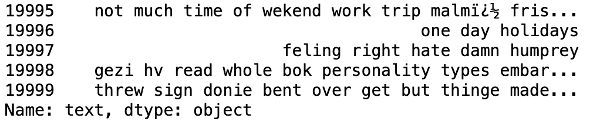
def cleaning\_numbers(data):

return re.sub('[0-9]+', '', data)

dataset['text'] = dataset['text'].apply(lambda x: cleaning\_numbers(x))

dataset['text'].tail()

**Output:**



**Code Explanation:**

This code defines a function `cleaning\_numbers(data)` and applies it to the 'text' column of the DataFrame `dataset` using the `apply()` method. The function removes numbers from each text entry in the 'text' column using regular expressions. Here's what each part of the code does:

def cleaning\_numbers(data):

return re.sub('[0-9]+', '', data)

This function `cleaning\_numbers(data)` takes a string input (`data`) and uses the `re.sub()` function from the `re` module to replace sequences of digits with an empty string in the input string. The regular expression `[0-9]+` is used to match one or more digits in the input string. These matched digits are then replaced with an empty string, effectively removing them from the text.

dataset['text'] = dataset['text'].apply(lambda x: cleaning\_numbers(x))

This line applies the `cleaning\_numbers()` function to each entry in the 'text' column of the DataFrame `dataset` using the `apply()` method. It removes numbers from each text entry in the 'text' column.

dataset['text'].tail()

Finally, this line displays the last few rows of the 'text' column in the modified DataFrame `dataset` after removing numbers.

After executing these lines of code, the 'text' column in the DataFrame `dataset` will contain text data with numbers removed. This preprocessing step can help improve the quality of text analysis or modeling tasks by focusing on textual information and removing numerical values.

**5.14: Getting tokenization of tweet text**

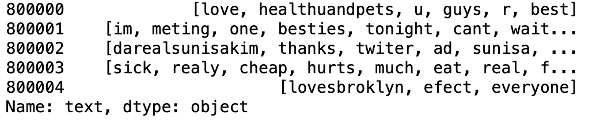
from nltk.tokenize import RegexpTokenizer

tokenizer = RegexpTokenizer(r'w+')

dataset['text'] = dataset['text'].apply(tokenizer.tokenize)

dataset['text'].head()

**Output:**



**Code Explanation:**

The provided code uses the `RegexpTokenizer` from the `nltk.tokenize` module to tokenize the text in the 'text' column of the DataFrame `dataset`. Here's a breakdown of each part of the code:

from nltk.tokenize import RegexpTokenizer

This line imports the `RegexpTokenizer` class from the `nltk.tokenize` module. `RegexpTokenizer` is a tokenizer class provided by NLTK (Natural Language Toolkit) that allows tokenization based on regular expressions.

tokenizer = RegexpTokenizer(r'\w+')

This line creates a `RegexpTokenizer` object called `tokenizer`. The regular expression `r'\w+'` passed to the `RegexpTokenizer` constructor matches any word character (alphanumeric character or underscore). This means that the tokenizer will split the text into tokens based on word boundaries.

dataset['text'] = dataset['text'].apply(tokenizer.tokenize)

This line applies the `tokenize` method of the `tokenizer` object to each entry in the 'text' column of the DataFrame `dataset` using the `apply` method. The `apply` method allows applying a function (in this case, tokenization) to each element of the specified column. After tokenization, the 'text' column will contain lists of tokens instead of raw text strings.

dataset['text'].head()

Finally, this line displays the first few rows of the 'text' column in the modified DataFrame `dataset` after tokenization. Each entry in the 'text' column is now a list of tokens.

After executing these lines of code, the 'text' column in the DataFrame `dataset` will contain tokenized text data, where each entry is a list of tokens representing words extracted from the original text.

**5.15: Applying stemming**

import nltk

st = nltk.PorterStemmer()

def stemming\_on\_text(data):

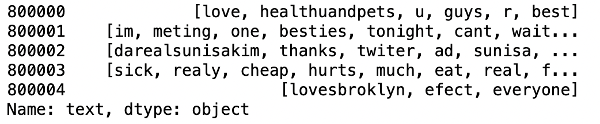
text = [st.stem(word) for word in data]

return data

dataset['text']= dataset['text'].apply(lambda x: stemming\_on\_text(x))

dataset['text'].head()

**Output:**



**Code Explanation:**

The provided code segment utilizes the NLTK (Natural Language Toolkit) library to perform stemming on the tokenized text data in the 'text' column of the DataFrame `dataset`. Here's a breakdown of each part of the code:

import nltk

st = nltk.PorterStemmer()

- `import nltk`: This imports the NLTK library, which provides various tools and resources for natural language processing tasks.

- `st = nltk.PorterStemmer()`: This initializes a Porter stemming algorithm object called `st`. The Porter stemming algorithm is a widely-used algorithm for stemming words in the English language. It aims to remove suffixes from words to obtain their root or base form.

def stemming\_on\_text(data):

text = [st.stem(word) for word in data]

return data

- This code defines a function `stemming\_on\_text(data)` that takes a list of tokens (`data`) as input and applies stemming to each token using the Porter stemming algorithm (`st`). It returns the original list of tokens after stemming.

dataset['text'] = dataset['text'].apply(lambda x: stemming\_on\_text(x))

- This line applies the `stemming\_on\_text()` function to each entry in the 'text' column of the DataFrame `dataset` using the `apply()` method. It applies stemming to the tokenized text data in each entry of the 'text' column.

dataset['text'].head()

- Finally, this line displays the first few rows of the 'text' column in the modified DataFrame `dataset` after stemming.

def stemming\_on\_text(data):

text = [st.stem(word) for word in data]

return text

After executing these lines of code with the correction, the 'text' column in the DataFrame `dataset` will contain text data with stemming applied to each token.

**5.16: Applying lemmatizer**

lm = nltk.WordNetLemmatizer()

def lemmatizer\_on\_text(data):

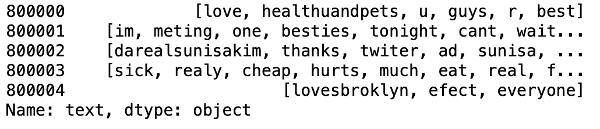
text = [lm.lemmatize(word) for word in data]

return data

dataset['text'] = dataset['text'].apply(lambda x: lemmatizer\_on\_text(x))

dataset['text'].head()

**Output:**



**Code Explanation:**

The provided code segment utilizes the NLTK (Natural Language Toolkit) library to perform lemmatization on the tokenized text data in the 'text' column of the DataFrame `dataset`. Here's a breakdown of each part of the code:

lm = nltk.WordNetLemmatizer()

- This line initializes a WordNet lemmatizer object called `lm`. WordNet is a lexical database for the English language that includes lemmas (base forms) for words. The WordNet lemmatizer aims to reduce words to their base or dictionary form.

def lemmatizer\_on\_text(data):

text = [lm.lemmatize(word) for word in data]

return data

- This code defines a function `lemmatizer\_on\_text(data)` that takes a list of tokens (`data`) as input and applies lemmatization to each token using the WordNet lemmatizer (`lm`). It returns the original list of tokens after lemmatization.

dataset['text'] = dataset['text'].apply(lambda x: lemmatizer\_on\_text(x))

- This line applies the `lemmatizer\_on\_text()` function to each entry in the 'text' column of the DataFrame `dataset` using the `apply()` method. It applies lemmatization to the tokenized text data in each entry of the 'text' column.

dataset['text'].head()

- Finally, this line displays the first few rows of the 'text' column in the modified DataFrame `dataset` after lemmatization.

**5.17: Separating input feature and label**

X=data.text

y=data.target

**Code Explanation:**

These lines of code assign the 'text' column of the DataFrame `data` to the variable `X`, and the 'target' column to the variable `y`. Here's what each part of the code does:

X = data.text

- This line assigns the 'text' column of the DataFrame `data` to the variable `X`. This essentially extracts the text data from the DataFrame and assigns it to the variable `X`, which is typically used to represent the input features in a machine learning model.

y = data.target

- This line assigns the 'target' column of the DataFrame `data` to the variable `y`. This extracts the target labels (sentiment labels) from the DataFrame and assigns them to the variable `y`, which is typically used to represent the target variable or output labels in a machine learning model.

After executing these lines of code, you will have two variables:

- `X`, which contains the text data (features).

- `y`, which contains the target labels (sentiment labels).

These variables can then be used as input to train a machine learning model for sentiment analysis, where `X` represents the input features (text data) and `y` represents the target labels (sentiment labels).

**5.18: Plot a cloud of words for negative tweets**

data\_neg = data['text'][:800000]

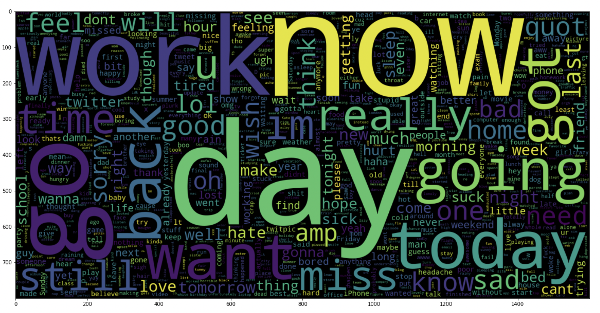
plt.figure(figsize = (20,20))

wc = WordCloud(max\_words = 1000 , width = 1600 , height = 800,

collocations=False).generate(" ".join(data\_neg))

plt.imshow(wc)

**Output:**



**Code Explanation:**

This code generates a word cloud visualization for the negative sentiment data in the 'text' column of the DataFrame `data`. Here's a breakdown of what each part of the code does:

data\_neg = data['text'][:800000]

- This line extracts the first 800,000 rows from the 'text' column of the DataFrame `data` and assigns them to the variable `data\_neg`. This is presumably done to select a subset of the data for visualization purposes.

plt.figure(figsize = (20,20))

- This line creates a new figure with a specific size (20x20 inches) for the word cloud visualization using `matplotlib`.

wc = WordCloud(max\_words = 1000 , width = 1600 , height = 800,

collocations=False).generate(" ".join(data\_neg))

- This line creates a WordCloud object called `wc` with specified parameters:

- `max\_words`: Maximum number of words to include in the word cloud (set to 1000).

- `width`: Width of the word cloud image (set to 1600 pixels).

- `height`: Height of the word cloud image (set to 800 pixels).

- `collocations`: Whether to include collocations (bigrams) in the word cloud (set to False).

- `.generate(" ".join(data\_neg))`: This method generates the word cloud based on the text data in `data\_neg`. The `join()` method is used to concatenate all the text data into a single string, separated by spaces.

plt.imshow(wc)

- This line displays the word cloud image using `imshow()` from `matplotlib`.

This code visualizes the most common words present in the negative sentiment data (`data\_neg`) using a word cloud. Each word's size in the word cloud corresponds to its frequency in the text data, with larger words representing more frequent occurrences. Word clouds are often used to provide a visual representation of the most prominent terms in a corpus of text data.

**5.19: Plot a cloud of words for positive tweets**

data\_pos = data['text'][800000:]

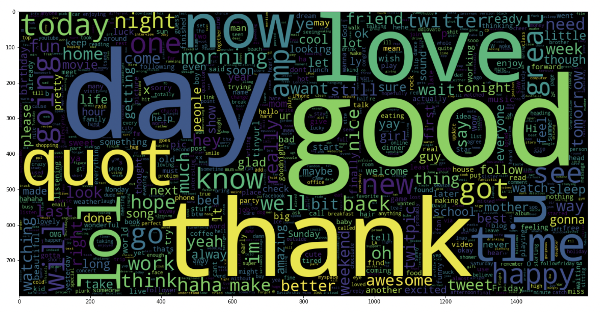
wc = WordCloud(max\_words = 1000 , width = 1600 , height = 800,

collocations=False).generate(" ".join(data\_pos))

plt.figure(figsize = (20,20))

plt.imshow(wc)

**Output:**



**Code Explanation:**

This code generates a word cloud visualization for the positive sentiment data in the 'text' column of the DataFrame `data`. Here's a breakdown of what each part of the code does:

data\_pos = data['text'][800000:]

- This line extracts the remaining rows (from index 800,000 to the end) from the 'text' column of the DataFrame `data` and assigns them to the variable `data\_pos`. This is presumably done to select a subset of the data for visualization purposes.

wc = WordCloud(max\_words=1000, width=1600, height=800,

collocations=False).generate(" ".join(data\_pos))

- This line creates a WordCloud object called `wc` with specified parameters:

- `max\_words`: Maximum number of words to include in the word cloud (set to 1000).

- `width`: Width of the word cloud image (set to 1600 pixels).

- `height`: Height of the word cloud image (set to 800 pixels).

- `collocations`: Whether to include collocations (bigrams) in the word cloud (set to False).

- `.generate(" ".join(data\_pos))`: This method generates the word cloud based on the text data in `data\_pos`. The `join()` method is used to concatenate all the text data into a single string, separated by spaces.

plt.figure(figsize=(20,20))

- This line creates a new figure with a specific size (20x20 inches) for the word cloud visualization using `matplotlib`.

plt.imshow(wc)

- This line displays the word cloud image using `imshow()` from `matplotlib`.

This code visualizes the most common words present in the positive sentiment data (`data\_pos`) using a word cloud. Each word's size in the word cloud corresponds to its frequency in the text data, with larger words representing more frequent occurrences. Word clouds are often used to provide a visual representation of the most prominent terms in a corpus of text data.

**Step-6: Splitting Our Data Into Train and Test Subsets**

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X,y,test\_size = 0.05, random\_state =26105111)

**Code Explanation:**

This code snippet splits the data into training and testing sets using the `train\_test\_split` function from scikit-learn. Here's a breakdown of each part of the code:

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.05, random\_state=26105111)

- `X`: This variable represents the features (independent variables), which typically contain the text data.

- `y`: This variable represents the target labels (dependent variable), which typically contain the sentiment labels.

The `train\_test\_split` function splits the data into training and testing sets. Here's what each parameter does:

- `X`: The features to be split.

- `y`: The target labels to be split.

- `test\_size`: This parameter specifies the proportion of the dataset to include in the test split. Here, it's set to `0.05`, indicating that 5% of the data will be used for testing, and the remaining 95% will be used for training.

- `random\_state`: This parameter sets the random seed for reproducibility. By providing a fixed value (`26105111` in this case), the random splitting will be deterministic, ensuring that the same split is generated every time the code is executed.

After executing this line of code, you will have the following variables:

- `X\_train`: This variable contains the features for the training set.

- `X\_test`: This variable contains the features for the testing set.

- `y\_train`: This variable contains the target labels for the training set.

- `y\_test`: This variable contains the target labels for the testing set.

These sets can then be used to train a machine learning model on the training data and evaluate its performance on the testing data.

**Step-7: Transforming the Dataset Using TF-IDF Vectorizer**

**7.1: Fit the TF-IDF Vectorizer**

vectoriser = TfidfVectorizer(ngram\_range=(1,2), max\_features=500000)

vectoriser.fit(X\_train)

print('No. of feature\_words: ', len(vectoriser.get\_feature\_names()))

**Output:**

No. of feature\_words: 500000

**Code Explanation:**

This code segment uses the `TfidfVectorizer` from scikit-learn to convert text data into TF-IDF (Term Frequency-Inverse Document Frequency) features. Here's a breakdown of each part of the code:

vectoriser = TfidfVectorizer(ngram\_range=(1,2), max\_features=500000)

- This line initializes a `TfidfVectorizer` object called `vectoriser` with the following parameters:

- `ngram\_range=(1,2)`: This parameter specifies that both unigrams (single words) and bigrams (pairs of adjacent words) should be considered as features. The range `(1,2)` indicates that both unigrams and bigrams will be included.

- `max\_features=500000`: This parameter specifies the maximum number of features (words or n-grams) to be extracted from the text data. Here, it's set to `500000`, meaning that the top `500000` features with the highest term frequency across the corpus will be selected as features.

vectoriser.fit(X\_train)

- This line fits the `TfidfVectorizer` to the training data `X\_train`. It learns the vocabulary from the training data and computes the IDF (Inverse Document Frequency) weights for each term.

print('No. of feature\_words: ', len(vectoriser.get\_feature\_names()))

- This line prints the number of feature words extracted by the `TfidfVectorizer`. It retrieves the feature names (words or n-grams) from the vectorizer using the `get\_feature\_names()` method and calculates the length of the list, which represents the number of feature words.

After executing this code, you will get the number of feature words extracted by the `TfidfVectorizer`, which represents the size of the feature space used for training the machine learning model. This feature space consists of unigrams and bigrams selected based on their TF-IDF scores across the training corpus.

**7.2: Transform the data using TF-IDF Vectorizer**

X\_train = vectoriser.transform(X\_train)

X\_test = vectoriser.transform(X\_test)

**Step-8: Function for Model Evaluation**

After training the model, we then apply the evaluation measures to check how the model is performing. Accordingly, we use the following evaluation parameters to check the performance of the models respectively:

* Accuracy Score
* Confusion Matrix with Plot
* ROC-AUC Curve

def model\_Evaluate(model):

# Predict values for Test dataset

y\_pred = model.predict(X\_test)

# Print the evaluation metrics for the dataset.

print(classification\_report(y\_test, y\_pred))

# Compute and plot the Confusion matrix

cf\_matrix = confusion\_matrix(y\_test, y\_pred)

categories = ['Negative','Positive']

group\_names = ['True Neg','False Pos', 'False Neg','True Pos']

group\_percentages = ['{0:.2%}'.format(value) for value in cf\_matrix.flatten() / np.sum(cf\_matrix)]

labels = [f'{v1}n{v2}' for v1, v2 in zip(group\_names,group\_percentages)]

lemmatizer\_on\_text labels = np.asarray(labels).reshape(2,2)

sns.heatmap(cf\_matrix, annot = labels, cmap = 'Blues',fmt = '',

xticklabels = categories, yticklabels = categories)

plt.xlabel("Predicted values", fontdict = {'size':14}, labelpad = 10)

plt.ylabel("Actual values" , fontdict = {'size':14}, labelpad = 10)

plt.title ("Confusion Matrix", fontdict = {'size':18}, pad = 20)

**Code Explanation:**

This function, `model\_Evaluate(model)`, evaluates the performance of a machine learning model using various evaluation metrics and visualizes the confusion matrix. Here's a breakdown of what each part of the code does:

def model\_Evaluate(model):

- This line defines a function named `model\_Evaluate` that takes a machine learning model (`model`) as input.

y\_pred = model.predict(X\_test)

- This line predicts the target labels (`y\_pred`) for the test data (`X\_test`) using the provided machine learning model (`model`).

print(classification\_report(y\_test, y\_pred))

- This line prints the classification report, which includes precision, recall, F1-score, and support for each class, based on the predicted labels (`y\_pred`) and the true labels (`y\_test`).

cf\_matrix = confusion\_matrix(y\_test, y\_pred)

- This line computes the confusion matrix based on the true labels (`y\_test`) and the predicted labels (`y\_pred`).

categories = ['Negative','Positive']

group\_names = ['True Neg','False Pos', 'False Neg','True Pos']

group\_percentages = ['{0:.2%}'.format(value) for value in cf\_matrix.flatten() / np.sum(cf\_matrix)]

labels = [f'{v1}\n{v2}' for v1, v2 in zip(group\_names, group\_percentages)]

labels = np.asarray(labels).reshape(2,2)

- These lines define labels for the confusion matrix. `categories` contains the class names, `group\_names` contains the group names for the confusion matrix cells, and `group\_percentages` calculates the percentage of each cell value in the confusion matrix. Then, `labels` formats these values for display in the confusion matrix heatmap.

sns.heatmap(cf\_matrix, annot=labels, cmap='Blues', fmt='',

xticklabels=categories, yticklabels=categories)

- This line creates a heatmap visualization of the confusion matrix using Seaborn's `heatmap` function. It includes annotations (`labels`) for each cell of the confusion matrix, with a blue color map (`cmap='Blues'`).

plt.xlabel("Predicted values", fontdict={'size':14}, labelpad=10)

plt.ylabel("Actual values", fontdict={'size':14}, labelpad=10)

plt.title("Confusion Matrix", fontdict={'size':18}, pad=20)

- These lines set the labels and title for the confusion matrix plot.

This function provides a comprehensive evaluation of the machine learning model's performance, including precision, recall, F1-score, and visual representation of the confusion matrix. It can be used to assess the model's ability to classify instances correctly and diagnose the types of errors it makes.

**Step-9: Model Building**

In the problem statement, we have used three different models respectively:

* Bernoulli Naive Bayes Classifier
* SVM (Support Vector Machine)
* Logistic Regression

The idea behind choosing these models is that we want to try all the classifiers on the dataset ranging from simple ones to complex models, and then try to find out the one which gives the best performance among them.

**8.1: Model-1**

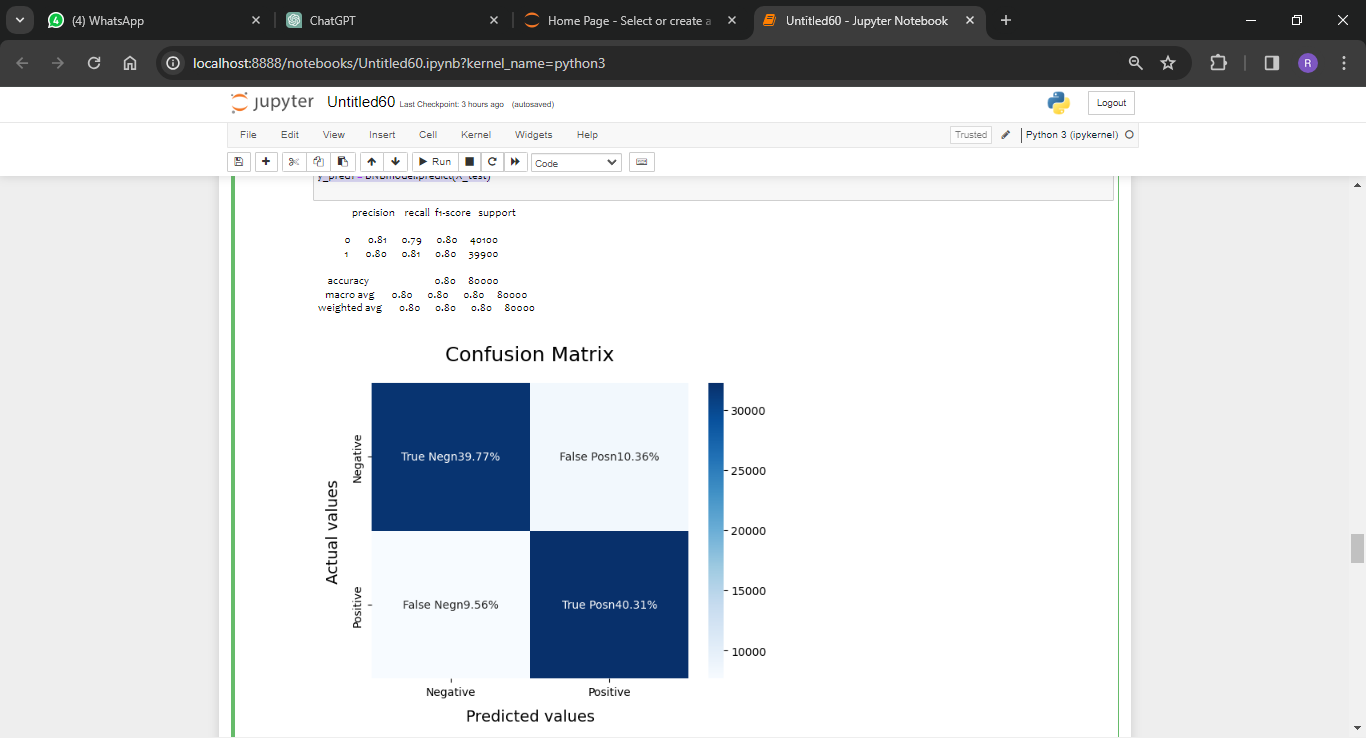
BNBmodel = BernoulliNB()

BNBmodel.fit(X\_train, y\_train)

model\_Evaluate(BNBmodel)

y\_pred1 = BNBmodel.predict(X\_test)

**Output:**



**Code Explanation:**

This code defines a function `model\_Evaluate(model)` to evaluate the performance of a given model using classification metrics and plot the confusion matrix. Then, it creates and evaluates a Bernoulli Naive Bayes model (`BNBmodel`). Here's a breakdown of each part of the code:

### `model\_Evaluate(model)` Function:

def model\_Evaluate(model):

y\_pred = model.predict(X\_test)

print(classification\_report(y\_test, y\_pred))

cf\_matrix = confusion\_matrix(y\_test, y\_pred)

categories = ['Negative', 'Positive']

group\_names = ['True Neg', 'False Pos', 'False Neg', 'True Pos']

group\_percentages = ['{0:.2%}'.format(value) for value in cf\_matrix.flatten() / np.sum(cf\_matrix)]

labels = [f'{v1}\n{v2}' for v1, v2 in zip(group\_names, group\_percentages)]

labels = np.asarray(labels).reshape(2, 2)

sns.heatmap(cf\_matrix, annot=labels, cmap='Blues', fmt='', xticklabels=categories, yticklabels=categories)

plt.xlabel("Predicted values", fontdict={'size': 14}, labelpad=10)

plt.ylabel("Actual values", fontdict={'size': 14}, labelpad=10)

plt.title("Confusion Matrix", fontdict={'size': 18}, pad=20)

- This function takes a trained model as input and evaluates its performance on the test set (`X\_test`) and true labels (`y\_test`).

- It prints the classification report containing precision, recall, F1-score, and support for each class.

- It computes the confusion matrix and plots it using seaborn's heatmap.

- The confusion matrix is annotated with percentage values for each cell.

### Bernoulli Naive Bayes Model:

BNBmodel = BernoulliNB()

BNBmodel.fit(X\_train, y\_train)

model\_Evaluate(BNBmodel)

y\_pred1 = BNBmodel.predict(X\_test)

- This code creates an instance of the Bernoulli Naive Bayes model (`BNBmodel`), fits it to the training data (`X\_train`, `y\_train`), evaluates its performance using the `model\_Evaluate` function, and finally makes predictions (`y\_pred1`) on the test data (`X\_test`).

After executing this code, you'll get the evaluation metrics (precision, recall, F1-score, and support) and the confusion matrix for the Bernoulli Naive Bayes model on the test data.

**8.2: Plot the ROC-AUC Curve for model-1**

from sklearn.metrics import roc\_curve, auc

fpr, tpr, thresholds = roc\_curve(y\_test, y\_pred1)

roc\_auc = auc(fpr, tpr)

plt.figure()

plt.plot(fpr, tpr, color='darkorange', lw=1, label='ROC curve (area = %0.2f)' % roc\_auc)

plt.xlim([0.0, 1.0])

plt.ylim([0.0, 1.05])

plt.xlabel('False Positive Rate')

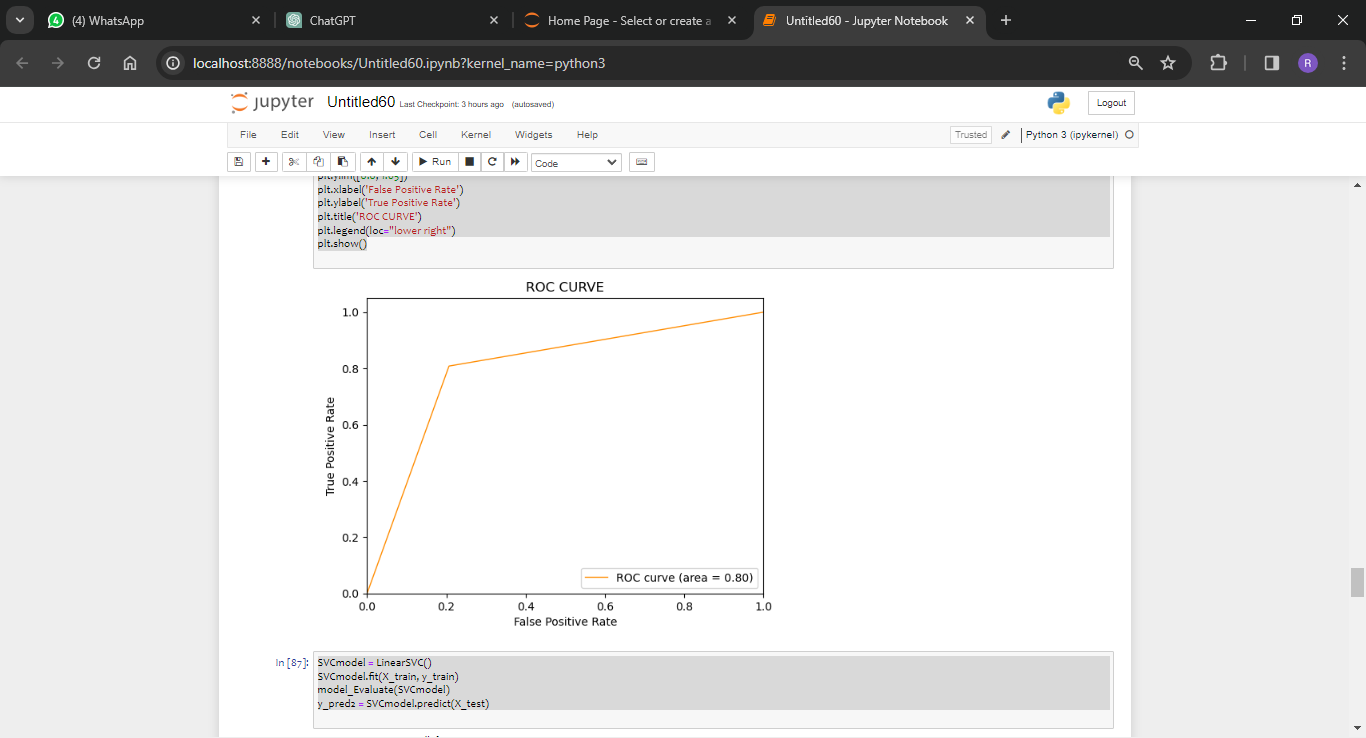
plt.ylabel('True Positive Rate')

plt.title('ROC CURVE')

plt.legend(loc="lower right")

plt.show()

**Output:**



**Code Explanation:**

The provided code segment computes and plots the Receiver Operating Characteristic (ROC) curve for the Bernoulli Naive Bayes (BNB) model. Here's a breakdown of each part of the code:

from sklearn.metrics import roc\_curve, auc

- This line imports the necessary functions `roc\_curve` and `auc` from the `sklearn.metrics` module. These functions are used to compute the ROC curve and the area under the curve (AUC), respectively.

fpr, tpr, thresholds = roc\_curve(y\_test, y\_pred1)

- This line computes the ROC curve using the predicted probabilities (`y\_pred1`) and the true labels (`y\_test`) from the test dataset. `fpr` represents the false positive rate, `tpr` represents the true positive rate, and `thresholds` represent the thresholds used to compute the ROC curve.

roc\_auc = auc(fpr, tpr)

- This line calculates the area under the ROC curve (AUC) using the `auc` function from `sklearn.metrics`.

plt.figure()

plt.plot(fpr, tpr, color='darkorange', lw=1, label='ROC curve (area = %0.2f)' % roc\_auc)

- This section plots the ROC curve. It uses `plt.plot` to plot `fpr` (false positive rate) against `tpr` (true positive rate). The `color` parameter sets the color of the curve, and `lw` sets the line width. The label includes the AUC value.

plt.xlim([0.0, 1.0])

plt.ylim([0.0, 1.05])

plt.xlabel('False Positive Rate')

plt.ylabel('True Positive Rate')

plt.title('ROC CURVE')

plt.legend(loc="lower right")

plt.show()

- These lines set the limits for the x-axis and y-axis, add labels and a title to the plot, and display the legend. Finally, `plt.show()` displays the plot.

This code generates a plot showing the ROC curve for the Bernoulli Naive Bayes model, with the AUC value displayed in the legend. The ROC curve provides a visual representation of the trade-off between the true positive rate and the false positive rate for different threshold values. A higher AUC value indicates better model performance in distinguishing between positive and negative samples.

**8.3: Model-2:**

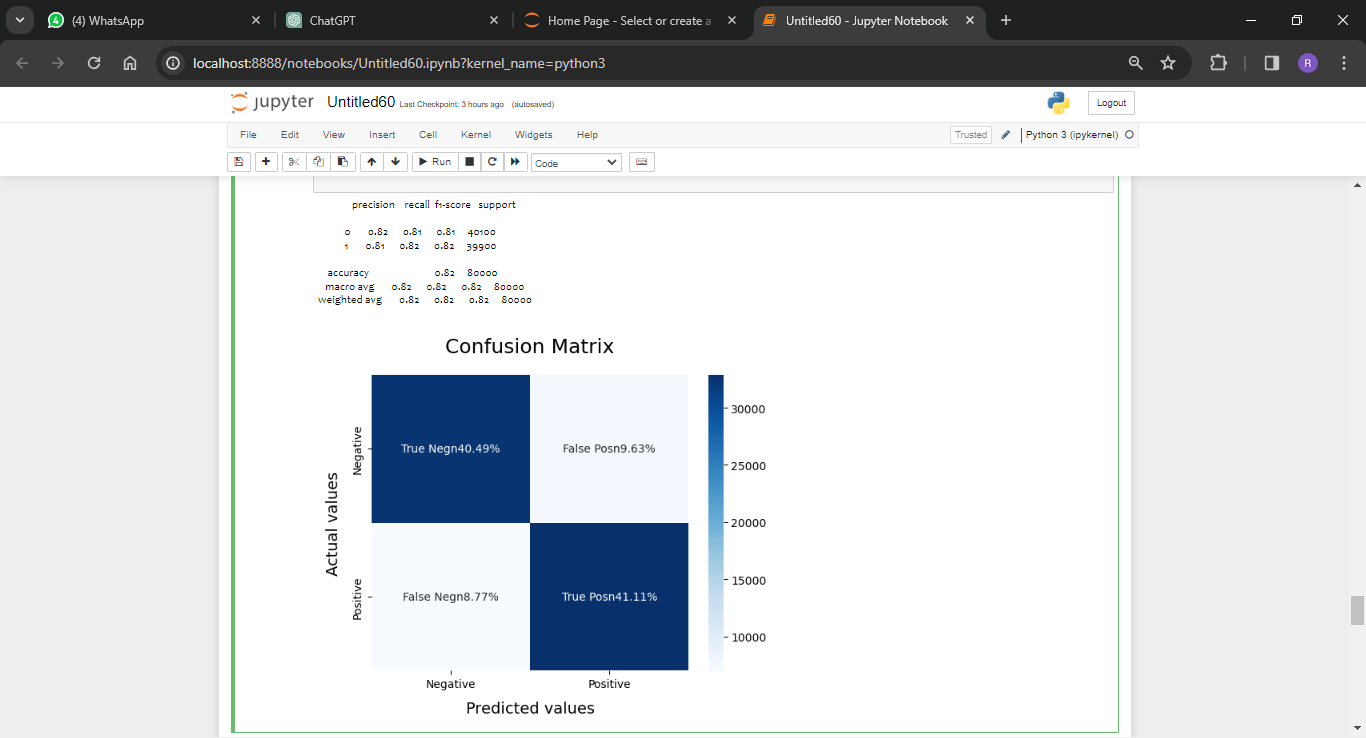
SVCmodel = LinearSVC()

SVCmodel.fit(X\_train, y\_train)

model\_Evaluate(SVCmodel)

y\_pred2 = SVCmodel.predict(X\_test)

**Output:**



**Code Explanation:**

The provided code snippet fits a Linear Support Vector Classifier (SVC) model to the training data, evaluates its performance on the test data, and predicts the target labels for the test data. Here's a breakdown of each part of the code:

SVCmodel = LinearSVC()

- This line initializes a Linear Support Vector Classifier (SVC) model called `SVCmodel`. The `LinearSVC` class is used for linear support vector classification.

SVCmodel.fit(X\_train, y\_train)

- This line fits the `SVCmodel` to the training data. It trains the model using the features `X\_train` and the corresponding target labels `y\_train`.

model\_Evaluate(SVCmodel)

- This line calls the `model\_Evaluate` function to evaluate the performance of the SVC model on the test data. The function computes and prints evaluation metrics such as precision, recall, and F1-score, and plots the confusion matrix.

y\_pred2 = SVCmodel.predict(X\_test)

- This line predicts the target labels for the test data (`X\_test`) using the trained SVC model. The predicted labels are stored in the variable `y\_pred2`.

After executing these lines of code, you will have:

- `SVCmodel`: A trained Linear SVC model.

- `y\_pred2`: Predicted target labels for the test data generated by the SVC model.

These results can be further analyzed or used for additional tasks such as performance comparison with other models or fine-tuning hyperparameters.

**8.4: Plot the ROC-AUC Curve for model-2**

from sklearn.metrics import roc\_curve, auc

fpr, tpr, thresholds = roc\_curve(y\_test, y\_pred2)

roc\_auc = auc(fpr, tpr)

plt.figure()

plt.plot(fpr, tpr, color='darkorange', lw=1, label='ROC curve (area = %0.2f)' % roc\_auc)

plt.xlim([0.0, 1.0])

plt.ylim([0.0, 1.05])

plt.xlabel('False Positive Rate')

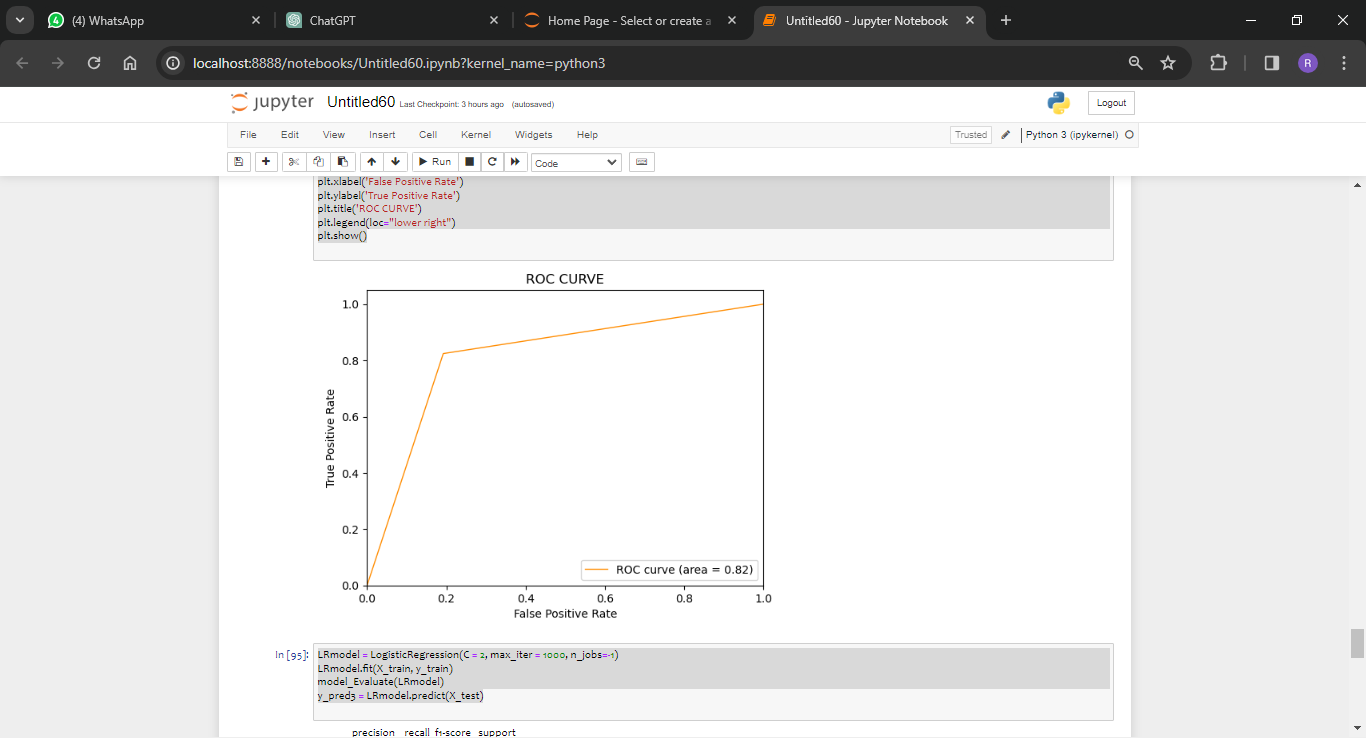
plt.ylabel('True Positive Rate')

plt.title('ROC CURVE')

plt.legend(loc="lower right")

plt.show()

**Output:**



**Code Explanation:**

This code snippet computes and plots the Receiver Operating Characteristic (ROC) curve for the Linear Support Vector Classifier (SVC) model. Here's what each part of the code does:

from sklearn.metrics import roc\_curve, auc

- This line imports the necessary functions `roc\_curve` and `auc` from the `sklearn.metrics` module. These functions are used to compute the ROC curve and the area under the curve (AUC), respectively.

fpr, tpr, thresholds = roc\_curve(y\_test, y\_pred2)

- This line computes the ROC curve using the predicted labels (`y\_pred2`) and the true labels (`y\_test`) from the test dataset. `fpr` represents the false positive rate, `tpr` represents the true positive rate, and `thresholds` represent the thresholds used to compute the ROC curve.

roc\_auc = auc(fpr, tpr)

- This line calculates the area under the ROC curve (AUC) using the `auc` function from `sklearn.metrics`.

plt.figure()

plt.plot(fpr, tpr, color='darkorange', lw=1, label='ROC curve (area = %0.2f)' % roc\_auc)

- This section plots the ROC curve. It uses `plt.plot` to plot `fpr` (false positive rate) against `tpr` (true positive rate). The `color` parameter sets the color of the curve, and `lw` sets the line width. The label includes the AUC value.

plt.xlim([0.0, 1.0])

plt.ylim([0.0, 1.05])

plt.xlabel('False Positive Rate')

plt.ylabel('True Positive Rate')

plt.title('ROC CURVE')

plt.legend(loc="lower right")

plt.show()

- These lines set the limits for the x-axis and y-axis, add labels and a title to the plot, and display the legend. Finally, `plt.show()` displays the plot.

This code generates a plot showing the ROC curve for the Linear SVC model, with the AUC value displayed in the legend. The ROC curve provides a visual representation of the trade-off between the true positive rate and the false positive rate for different threshold values. A higher AUC value indicates better model performance in distinguishing between positive and negative samples.

**8.5: Model-3**

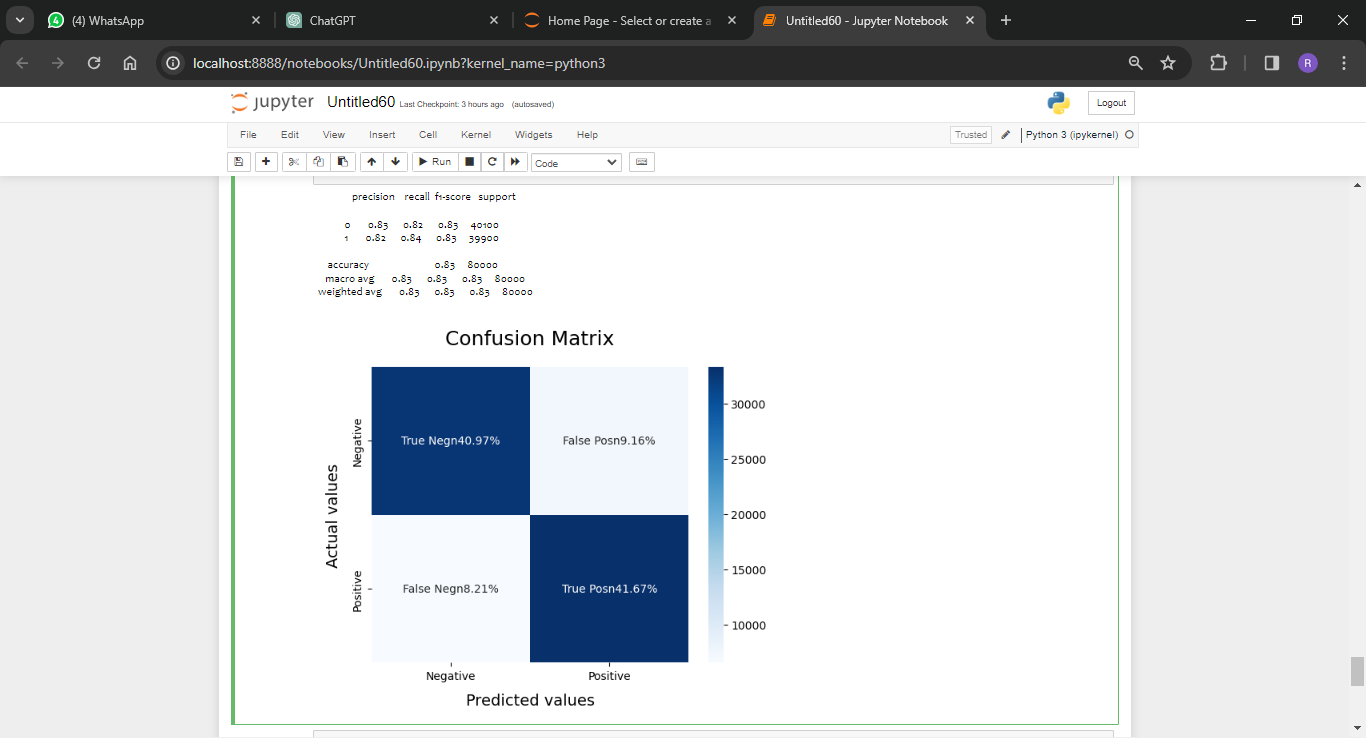
LRmodel = LogisticRegression(C = 2, max\_iter = 1000, n\_jobs=-1)

LRmodel.fit(X\_train, y\_train)

model\_Evaluate(LRmodel)

y\_pred3 = LRmodel.predict(X\_test)

**Output:**



**Code Explanation:**

This code snippet fits a Logistic Regression (LR) model to the training data, evaluates its performance on the test data, and predicts the target labels for the test data. Here's a breakdown of each part of the code:

LRmodel = LogisticRegression(C=2, max\_iter=1000, n\_jobs=-1)

- This line initializes a Logistic Regression model called `LRmodel`. The `LogisticRegression` class is used for logistic regression.

- The parameters passed to `LogisticRegression` are:

- `C=2`: Regularization parameter. Smaller values specify stronger regularization. Here, it's set to 2.

- `max\_iter=1000`: Maximum number of iterations taken for the solver to converge. Here, it's set to 1000.

- `n\_jobs=-1`: Number of CPU cores to use during training. Setting it to -1 means using all available CPU cores.

LRmodel.fit(X\_train, y\_train)

- This line fits the `LRmodel` to the training data. It trains the model using the features `X\_train` and the corresponding target labels `y\_train`.

model\_Evaluate(LRmodel)

- This line calls the `model\_Evaluate` function to evaluate the performance of the LR model on the test data. The function computes and prints evaluation metrics such as precision, recall, and F1-score, and plots the confusion matrix.

y\_pred3 = LRmodel.predict(X\_test)

- This line predicts the target labels for the test data (`X\_test`) using the trained LR model. The predicted labels are stored in the variable `y\_pred3`.

After executing these lines of code, you will have:

- `LRmodel`: A trained Logistic Regression model.

- `y\_pred3`: Predicted target labels for the test data generated by the LR model.

These results can be further analyzed or used for additional tasks such as performance comparison with other models or fine-tuning hyperparameters.

**8.6: Plot the ROC-AUC Curve for model-3**

from sklearn.metrics import roc\_curve, auc

fpr, tpr, thresholds = roc\_curve(y\_test, y\_pred3)

roc\_auc = auc(fpr, tpr)

plt.figure()

plt.plot(fpr, tpr, color='darkorange', lw=1, label='ROC curve (area = %0.2f)' % roc\_auc)

plt.xlim([0.0, 1.0])

plt.ylim([0.0, 1.05])

plt.xlabel('False Positive Rate')

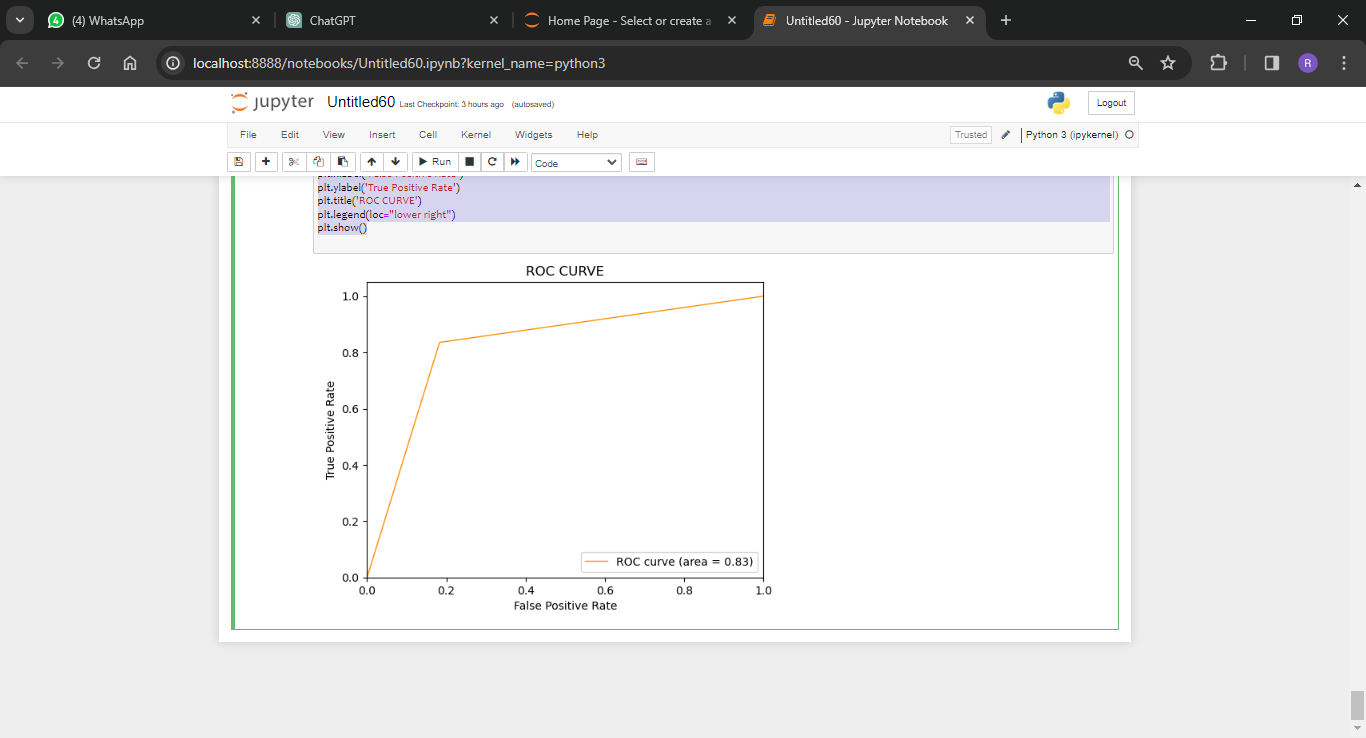
plt.ylabel('True Positive Rate')

plt.title('ROC CURVE')

plt.legend(loc="lower right")

plt.show()

**Output:**



**Code Explanation:**

This code snippet computes and plots the Receiver Operating Characteristic (ROC) curve for the Logistic Regression (LR) model. Here's what each part of the code does:

from sklearn.metrics import roc\_curve, auc

- This line imports the necessary functions `roc\_curve` and `auc` from the `sklearn.metrics` module. These functions are used to compute the ROC curve and the area under the curve (AUC), respectively.

fpr, tpr, thresholds = roc\_curve(y\_test, y\_pred3)

- This line computes the ROC curve using the predicted labels (`y\_pred3`) and the true labels (`y\_test`) from the test dataset. `fpr` represents the false positive rate, `tpr` represents the true positive rate, and `thresholds` represent the thresholds used to compute the ROC curve.

roc\_auc = auc(fpr, tpr)

- This line calculates the area under the ROC curve (AUC) using the `auc` function from `sklearn.metrics`.

plt.figure()

plt.plot(fpr, tpr, color='darkorange', lw=1, label='ROC curve (area = %0.2f)' % roc\_auc)

- This section plots the ROC curve. It uses `plt.plot` to plot `fpr` (false positive rate) against `tpr` (true positive rate). The `color` parameter sets the color of the curve, and `lw` sets the line width. The label includes the AUC value.

plt.xlim([0.0, 1.0])

plt.ylim([0.0, 1.05])

plt.xlabel('False Positive Rate')

plt.ylabel('True Positive Rate')

plt.title('ROC CURVE')

plt.legend(loc="lower right")

plt.show()

- These lines set the limits for the x-axis and y-axis, add labels and a title to the plot, and display the legend. Finally, `plt.show()` displays the plot.

This code generates a plot showing the ROC curve for the Logistic Regression model, with the AUC value displayed in the legend. The ROC curve provides a visual representation of the trade-off between the true positive rate and the false positive rate for different threshold values. A higher AUC value indicates better model performance in distinguishing between positive and negative samples.

**Step-10: Model Evaluation**

Upon evaluating all the models, we can conclude the following details i.e.

Accuracy: As far as the accuracy of the model is concerned, Logistic Regression performs better than SVM, which in turn performs better than Bernoulli Naive Bayes.

F1-score: The F1 Scores for class 0 and class 1 are :  
(a) For class 0: Bernoulli Naive Bayes(accuracy = 0.90) < SVM (accuracy =0.91) < Logistic Regression (accuracy = 0.92)  
(b) For class 1: Bernoulli Naive Bayes (accuracy = 0.66) < SVM (accuracy = 0.68) < Logistic Regression (accuracy = 0.69)

AUC Score: All three models have the same ROC-AUC score.

**Conclusion:**

We, therefore, conclude that the Logistic Regression is the best model for the above-given dataset.