

**School of Computer Science and Engineering**

**(WINTER 2022-2023)**

Student Name: **RUDRAJIT DAS**

Reg No: **22MCB0008**

Email: [rudrajit.das2022@vitstudent.ac.in](mailto:rudrajit.das2022@vitstudent.ac.in)

Mobile:9831060863

Faculty Name: **DURGESH KUMAR**

Subject Name: SOCIAL NETWORK ANALYTICS LAB

Assessment No: 3

**LAB EXERCISE 3**

**AIM:** SENTIMENT ANALYSIS OF TWEETS/PRODUCTS REVIEW

**1.AIM:**

Sentiment analysis, also known as opinion mining, is the process of determining and classifying the sentiment expressed in a piece of text. It involves using natural language processing (NLP) techniques and machine learning algorithms to analyze the emotional tone and subjective information present in textual data.The goal of sentiment analysis is to understand the underlying sentiment or attitude of the writer or speaker towards a particular subject, whether it is positive, negative, or neutral. It can be applied to various types of text, such as customer reviews, social media posts, surveys, news articles, and more.There are several approaches to sentiment analysis, ranging from rule-based methods to more advanced machine learning techniques. Here are a few common steps involved in sentiment analysis:

1. Text preprocessing: This step involves cleaning and preparing the text data by removing noise, such as special characters and punctuation, and converting the text to a standard format.
2. Feature extraction: Relevant features or attributes are extracted from the text data. This could involve techniques like bag-of-words, word embeddings (such as Word2Vec or GloVe), or more advanced approaches like contextual embeddings (e.g., BERT).
3. Sentiment classification: Once the features are extracted, a machine learning model is trained to classify the sentiment of the text. Common techniques include logistic regression, support vector machines (SVM), random forests, or more sophisticated deep learning models like recurrent neural networks (RNNs) or convolutional neural networks (CNNs).
4. Evaluation: The performance of the sentiment analysis model is assessed using appropriate evaluation metrics, such as accuracy, precision, recall, or F1 score. This is typically done by comparing the predicted sentiment labels with manually annotated or labeled data.

Sentiment analysis has a wide range of applications. Some examples include:

* Customer feedback analysis: Companies can analyze customer reviews, comments, and social media posts to understand customer satisfaction, identify product issues, and improve their offerings.
* Brand monitoring: Sentiment analysis can be used to monitor public sentiment and perception about a brand, product, or service in real-time.
* Social media analysis: It helps in tracking and understanding public sentiment on social media platforms, enabling organizations to respond to customer concerns or identify emerging trends.
* Market research: Sentiment analysis can provide insights into consumer opinions, preferences, and behaviours, aiding in market research and decision-making.
* Political analysis: Sentiment analysis can be applied to political speeches, news articles, and social media conversations to gauge public opinion during elections or on specific issues.

It is worth noting that while sentiment analysis has made significant progress, it still faces challenges in accurately interpreting nuanced sentiments, sarcasm, context-dependent sentiment, and cultural variations in language usage.

**2. Implementation of Machine Learning Models:**

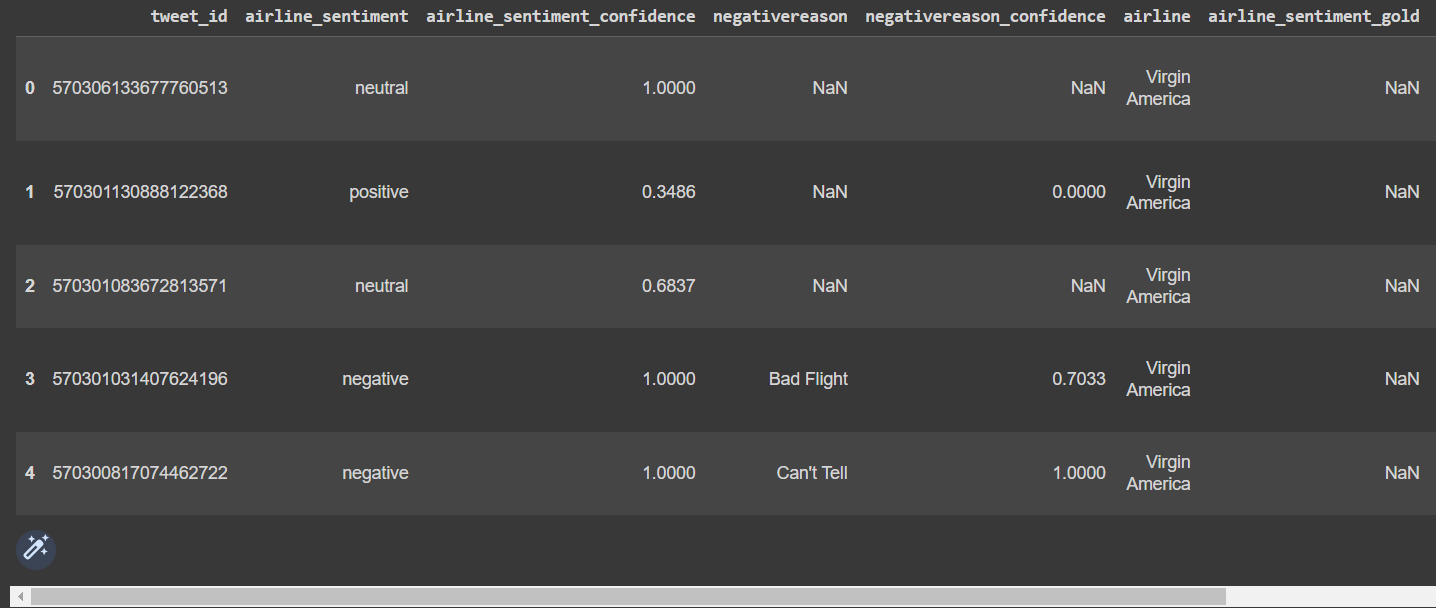
**2a.Classic Machine Learning Models:**

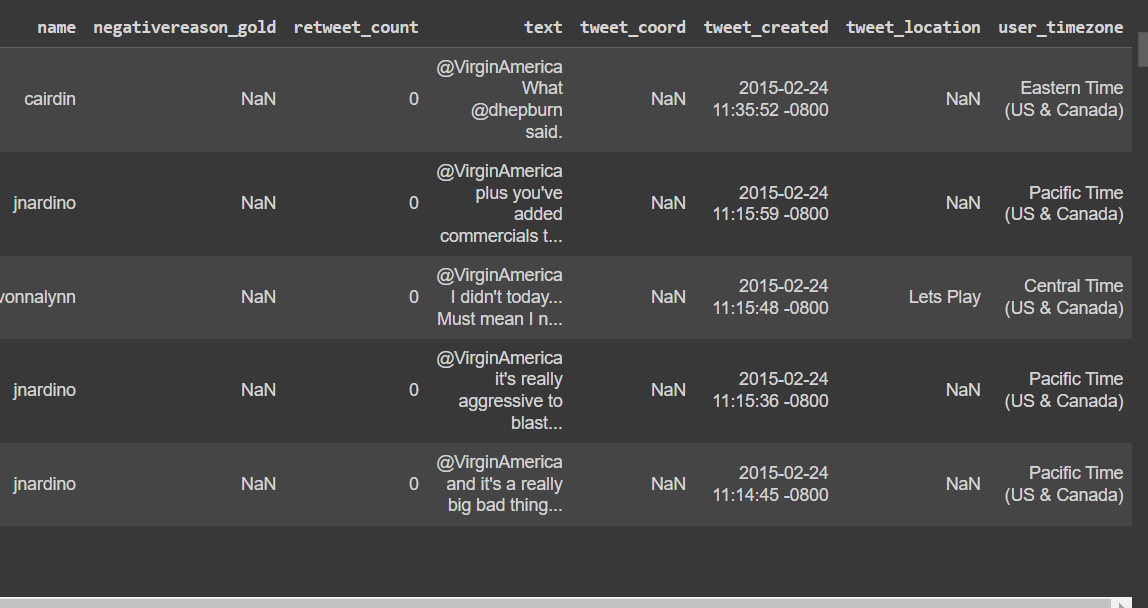
SENTIMENT ANALYSIS OF TWEETS USING SVM MACHINE LEARNING MODEL:

**Objective**:

You are given a data of US Airline tweets and their sentiment. The task is to do sentiment analysis about the problems of each major U.S. airline. Twitter data was scraped from February of 2015 and contributors were asked to first classify positive, negative, and neutral tweets, followed by categorizing negative reasons (such as "late flight" or "rude service").

**Dataset:**





Here we have given the snippets of the dataset used using the data.head() file for finding the top few rows of data from the large dataset. As of now we are trying to clean the data and get started with preprocessing the dataset so that we can go in for further analysis.

Our main objective will be to preprocess the dataset and do EDA followed by vectorization and then building the SVM model followed by creating CONFUSION Matrix and getting the accuracy percentage of model created.

**PROCEDURE:**

**# Step 1: Data preprocessing and feature extraction**

1. Read and preprocess the text data (cleaning, normalization, tokenization, etc.).

2. Extract features from the preprocessed text data (e.g., bag-of-words, word embeddings).

**# Step 2: Training the SVM model**

1. Split the dataset into training and testing sets.

2. Train the SVM model on the training set using the extracted features.

- Initialize the SVM model with appropriate hyperparameters.

- Use the training data and their corresponding sentiment labels.

- Apply feature scaling if necessary.

- Fit the SVM model to the training data.

**# Step 3: Evaluating the SVM model**

1. Apply the trained SVM model to the testing set.

2. Calculate the accuracy, precision, recall, F1 score, or other evaluation metrics

by comparing the predicted sentiment labels with the ground truth labels.

**# Step 4: Making predictions with the SVM model**

1. Preprocess and extract features from the new text data you want to predict.

2. Apply the trained SVM model to the new data to obtain the predicted sentiment label.

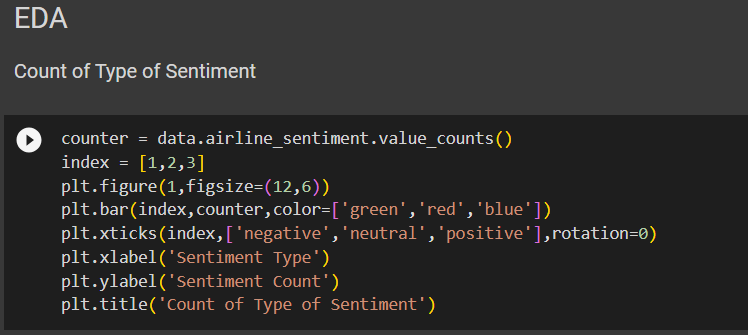
# Example code snippet using scikit-learn in Python:

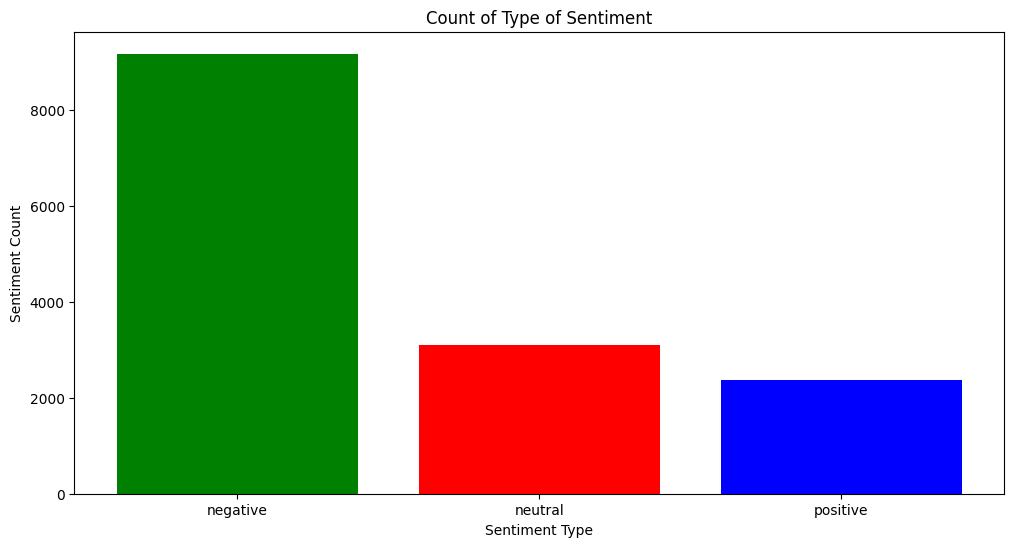
from sklearn.svm import SVC

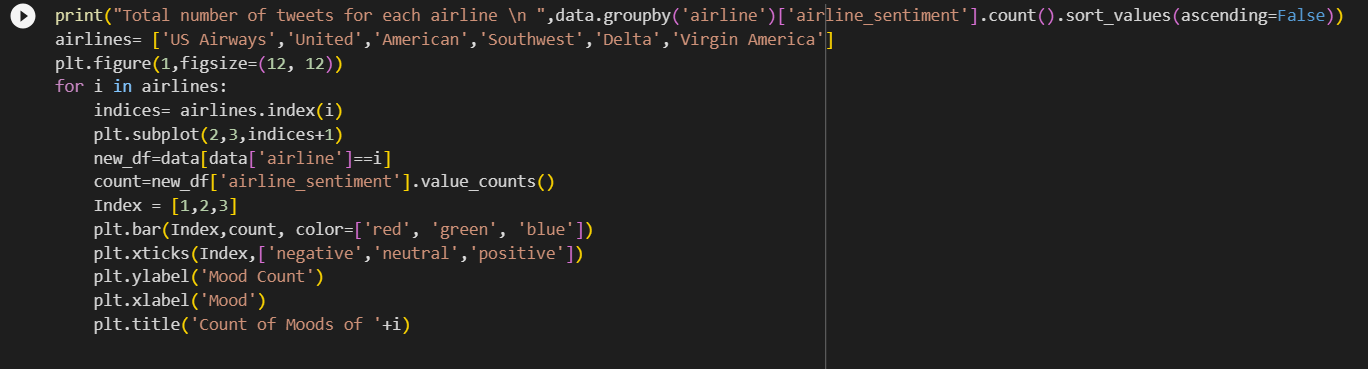
from sklearn.feature\_extraction.text import TfidfVectorizer

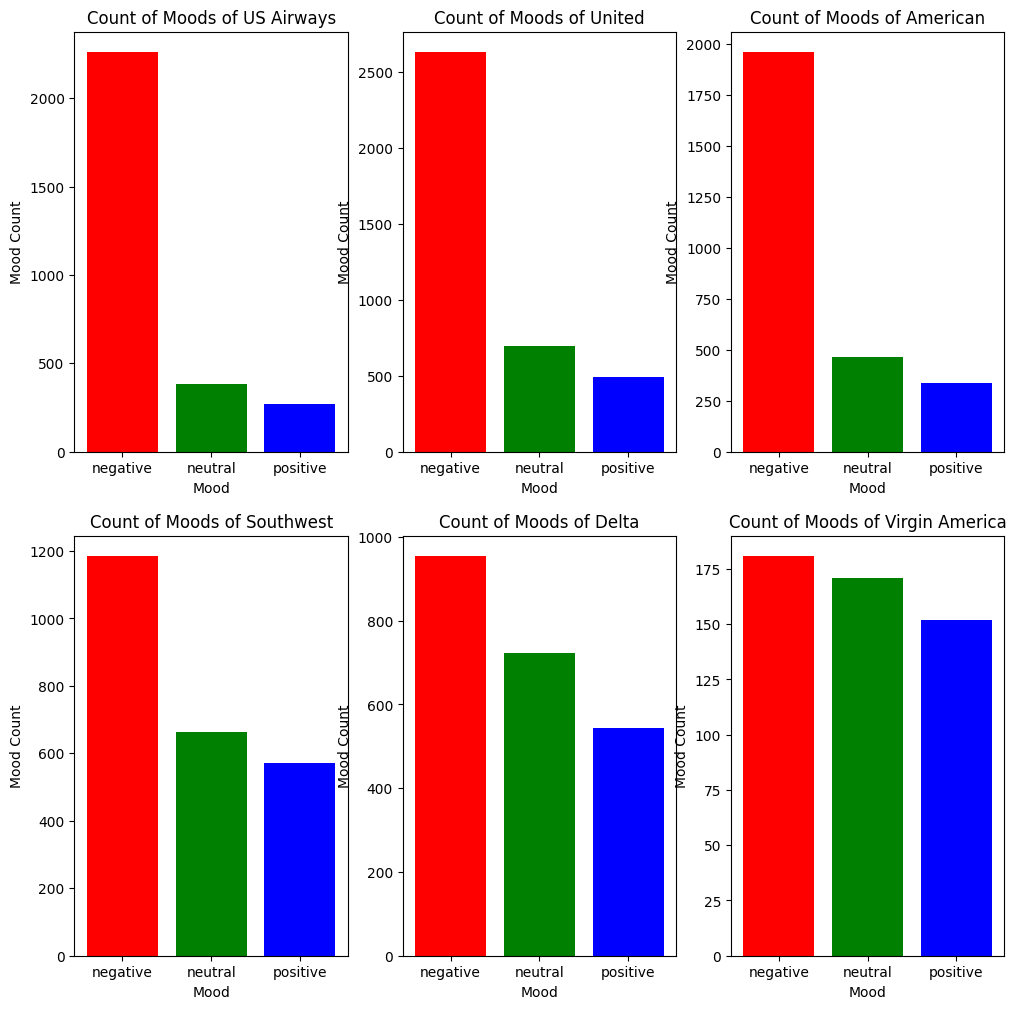
from sklearn.model\_selection import train\_test\_split

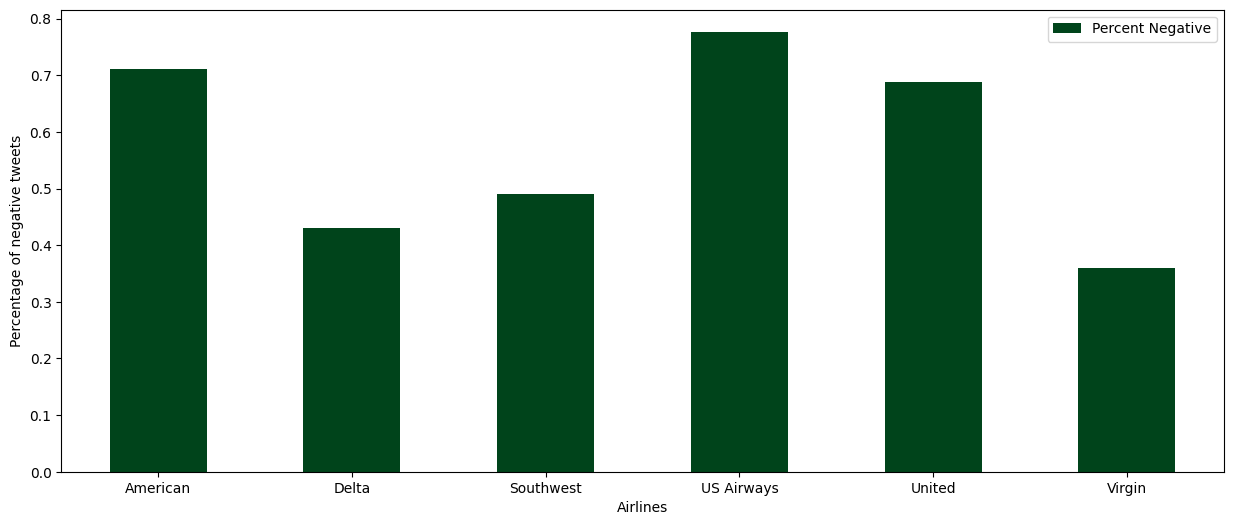
from sklearn.metrics import accuracy\_score

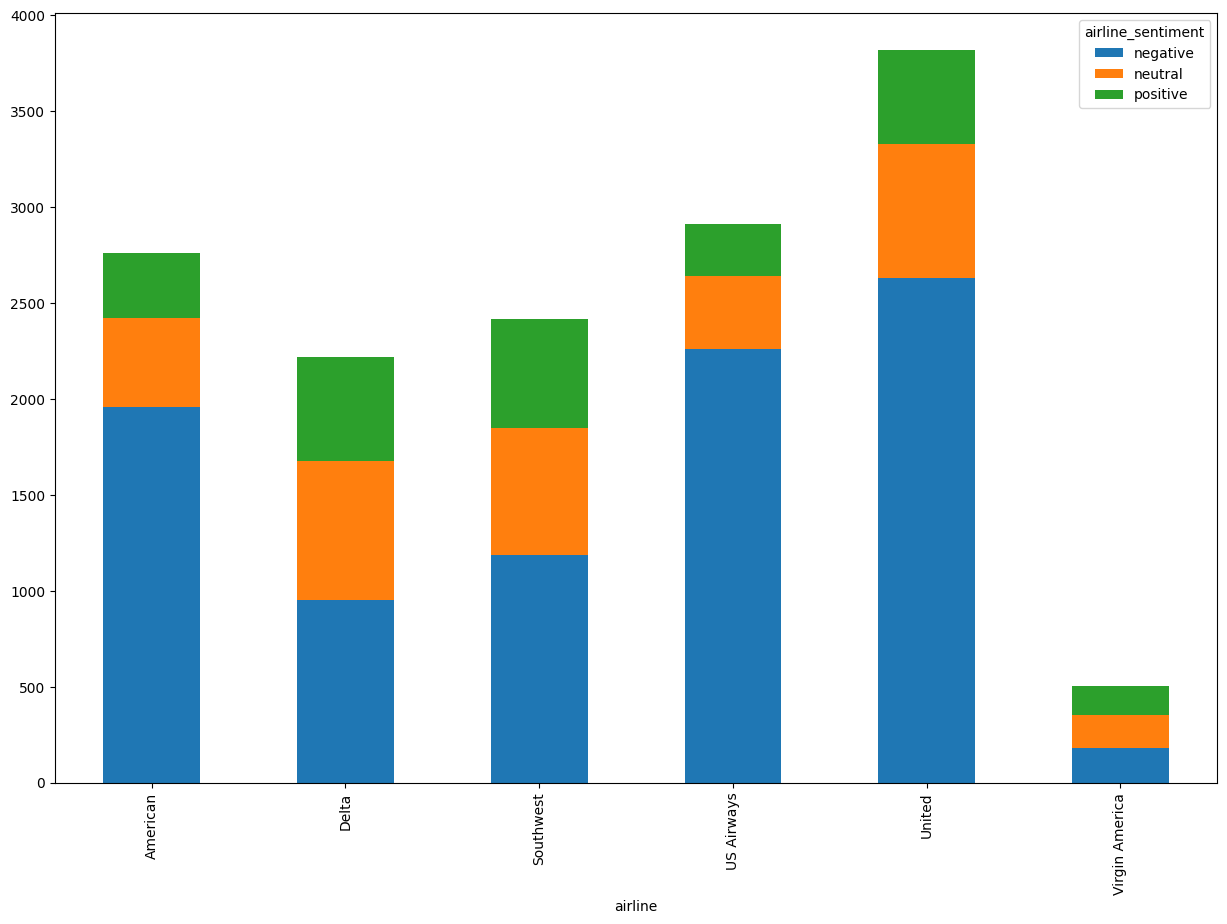
****

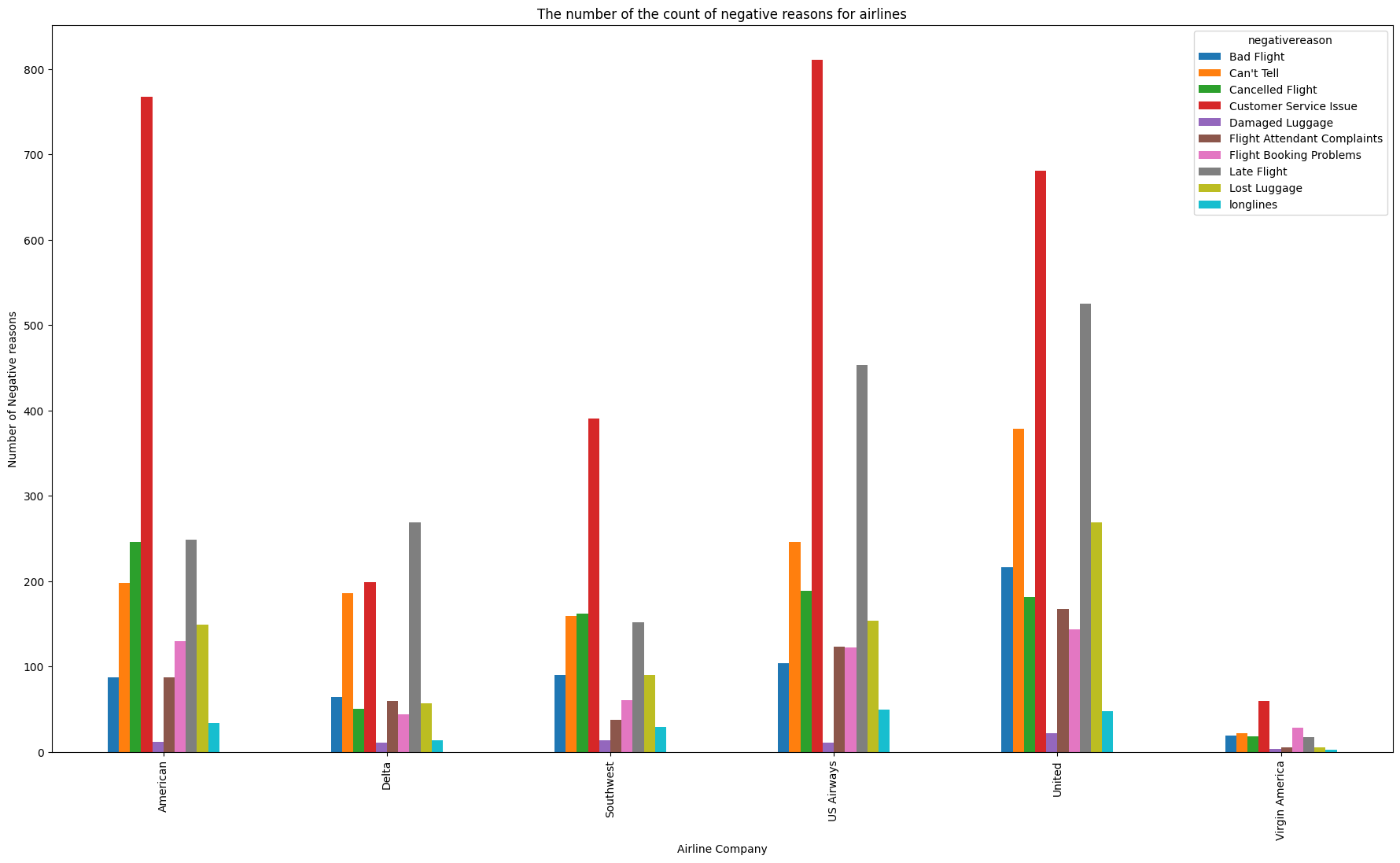
****

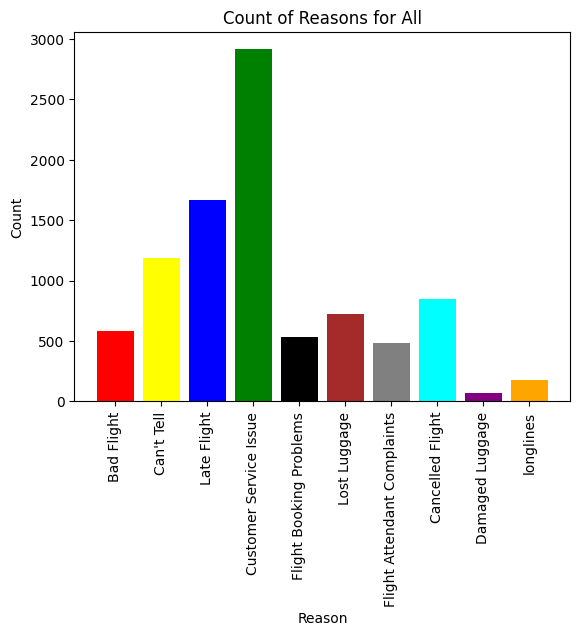
****

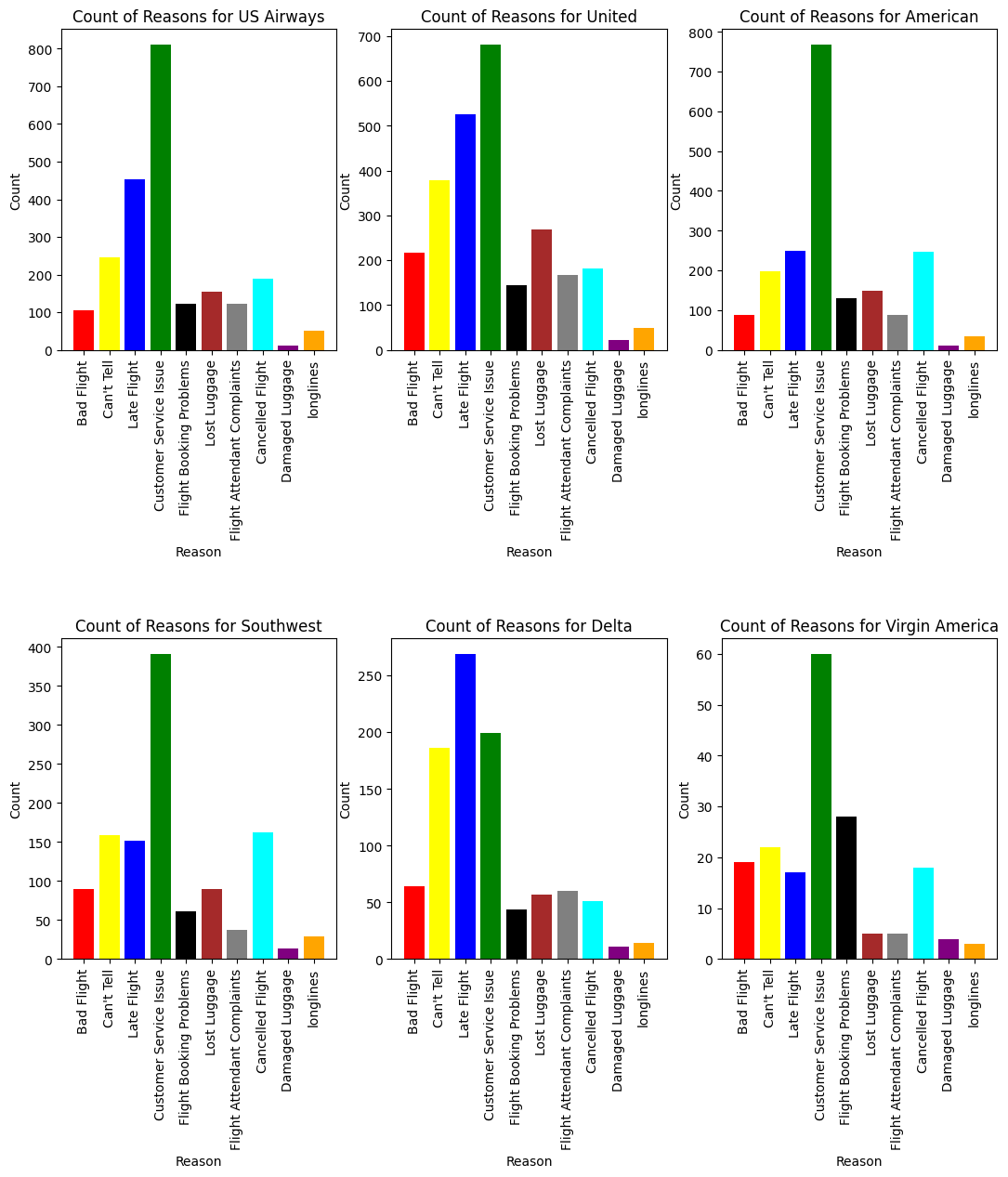
****

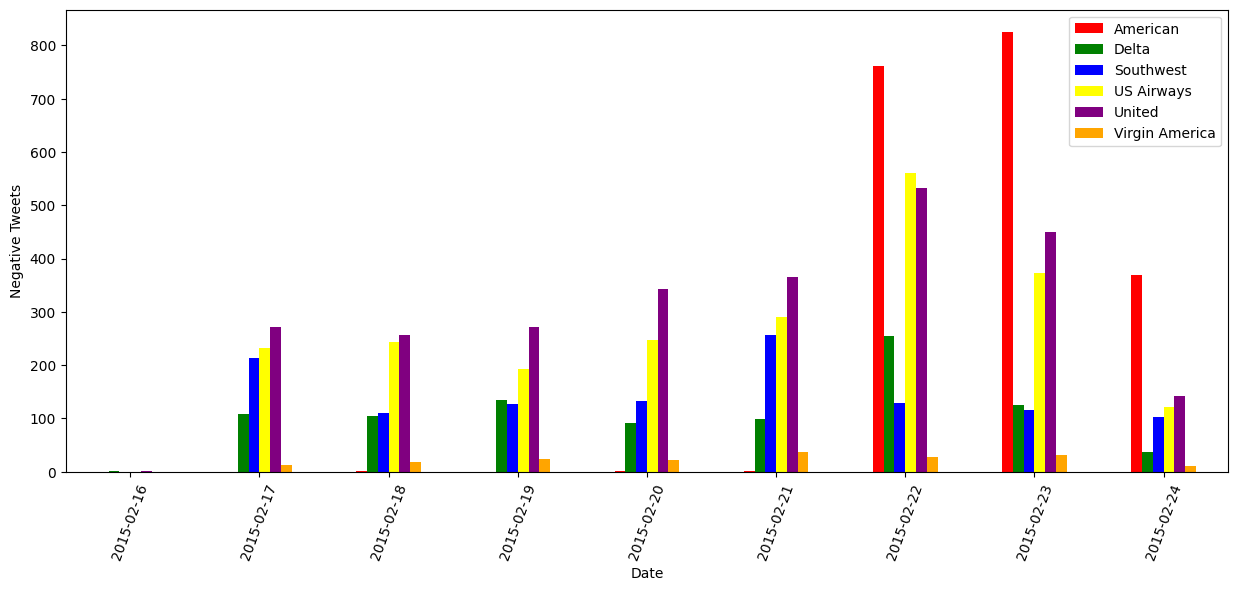


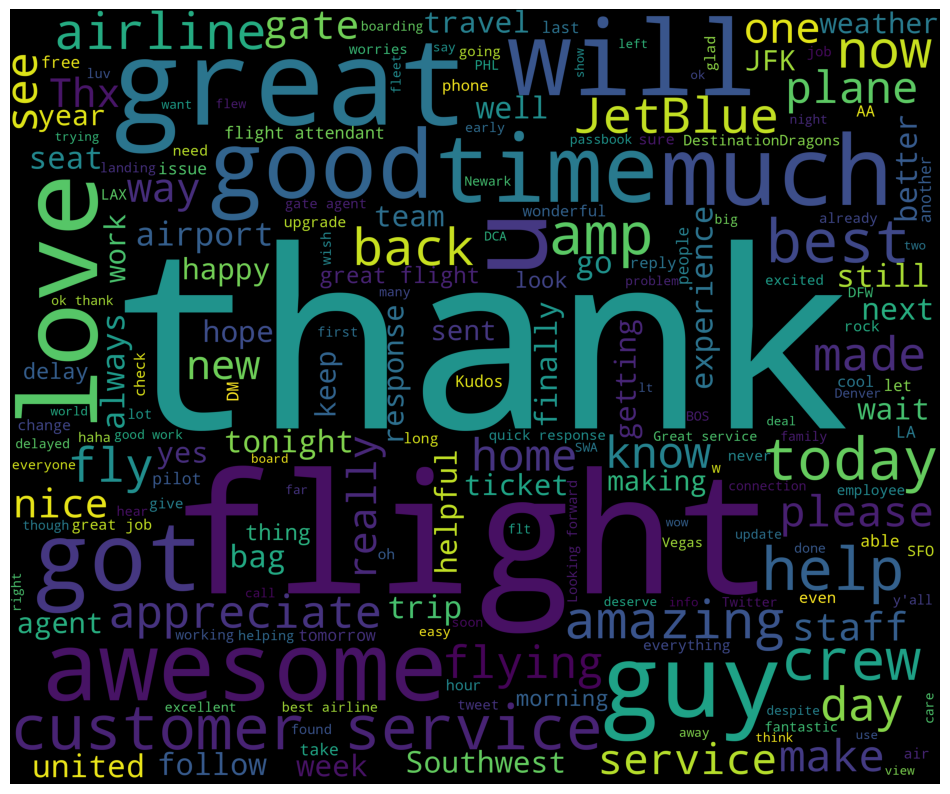


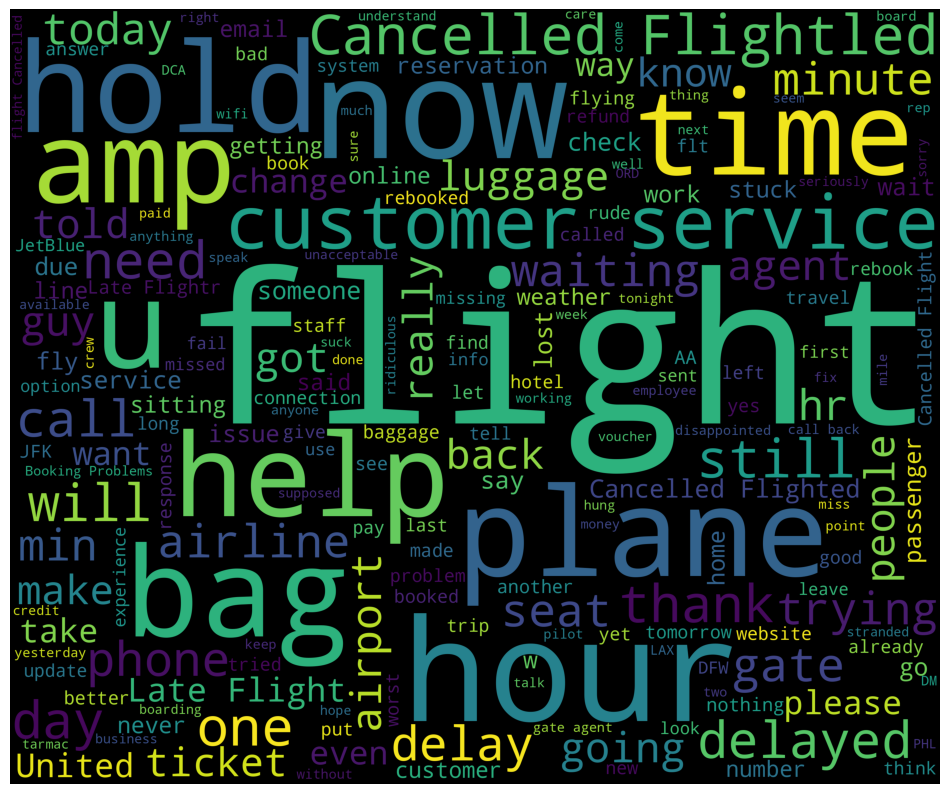


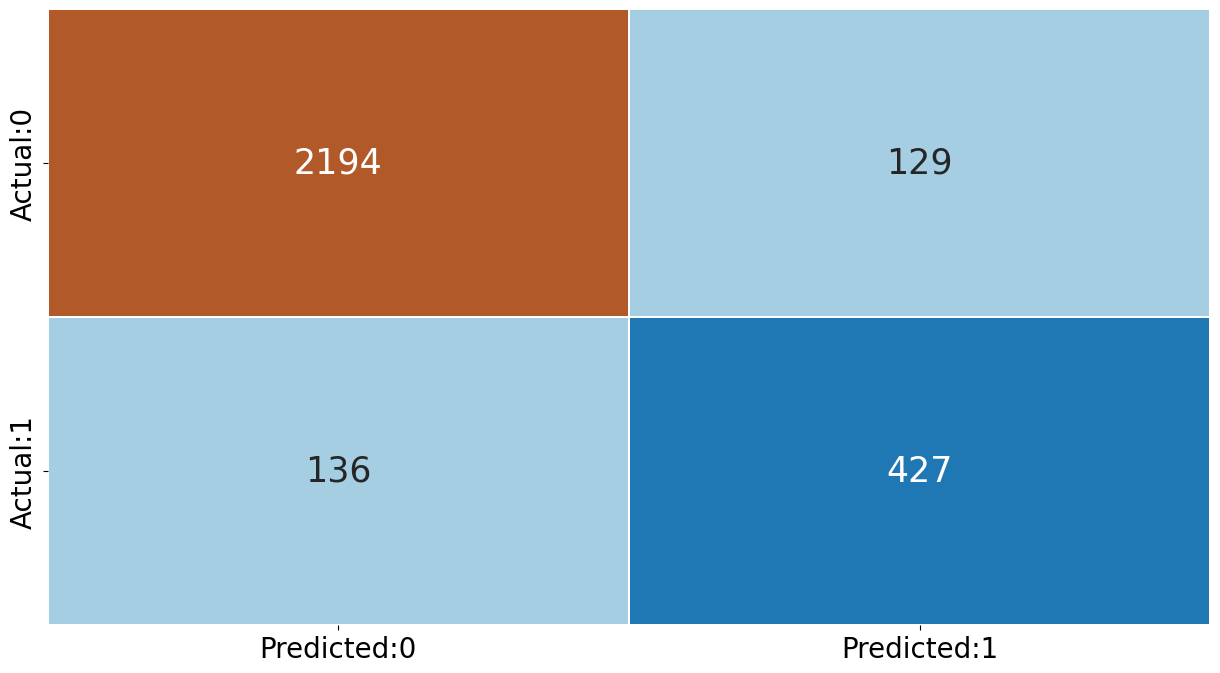


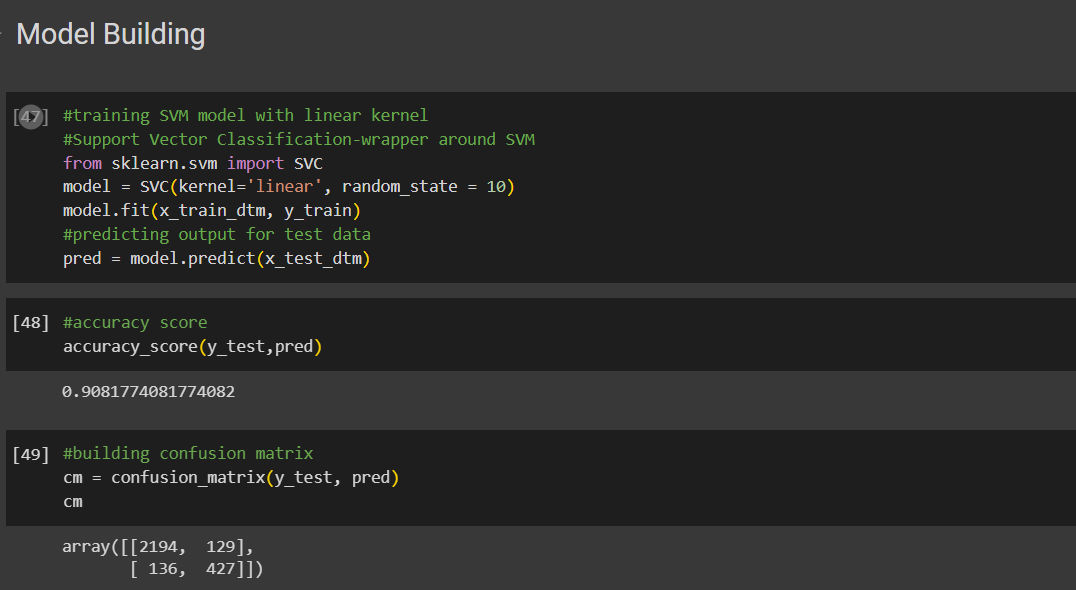


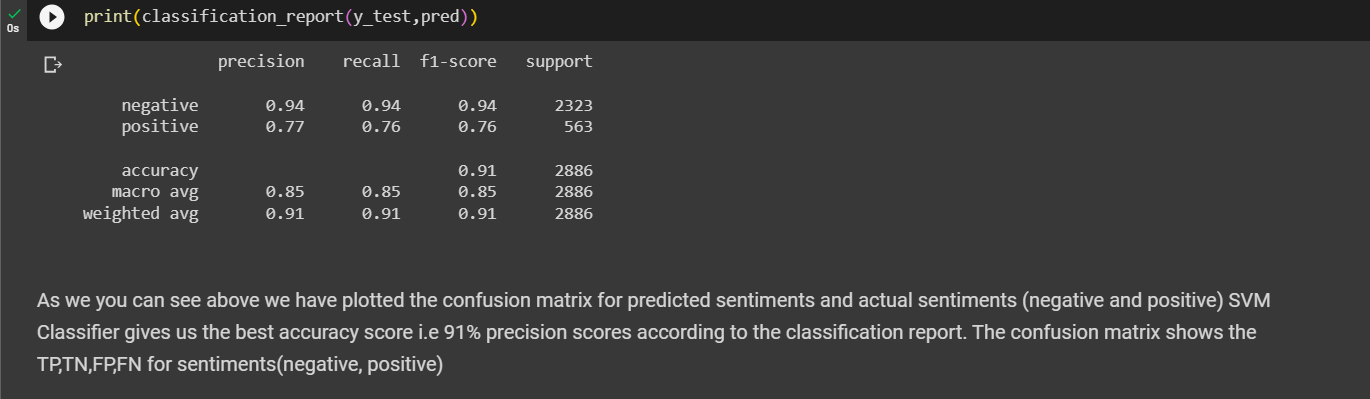












SENTIMENT ANALYSIS OF TWEETS USING NAÏVE BAYES ML MODEL:

**Objective**:

To classify negative and positive tweets using the Naïve-Bayes Model and obtaining accuracy and producing necessary graphs as required for clear demonstration.

**# Step 1: Data preprocessing and feature extraction**

1. Read and preprocess the text data (cleaning, normalization, tokenization, etc.).

2. Extract features from the preprocessed text data (e.g., bag-of-words, word embeddings).

**# Step 2: Training the Naive Bayes model**

1. Split the dataset into training and testing sets.

2. Train the Naive Bayes model on the training set using the extracted features.

- Initialize the Naive Bayes model (e.g., Gaussian Naive Bayes, Multinomial Naive Bayes).

- Use the training data and their corresponding sentiment labels.

- Fit the Naive Bayes model to the training data.

**# Step 3: Evaluating the Naive Bayes model**

1. Apply the trained Naive Bayes model to the testing set.

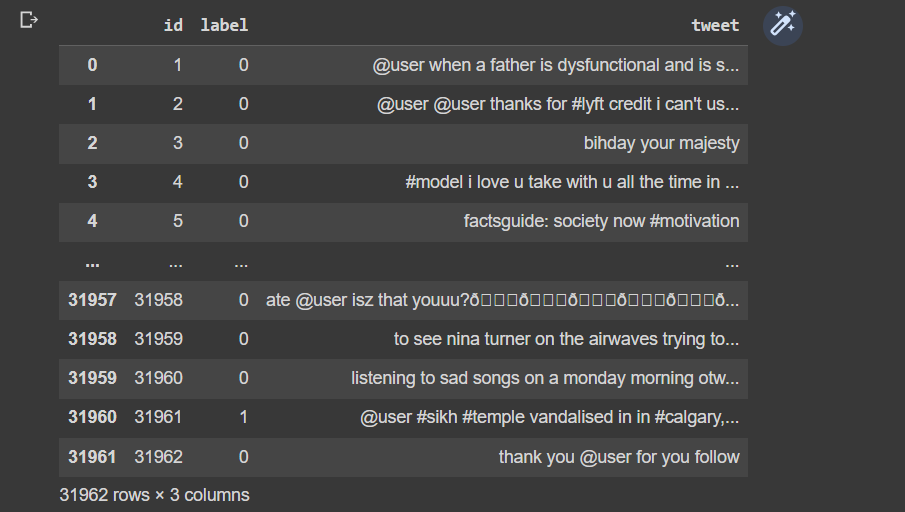
2. Calculate the accuracy, precision, recall, F1 score, or other evaluation metrics

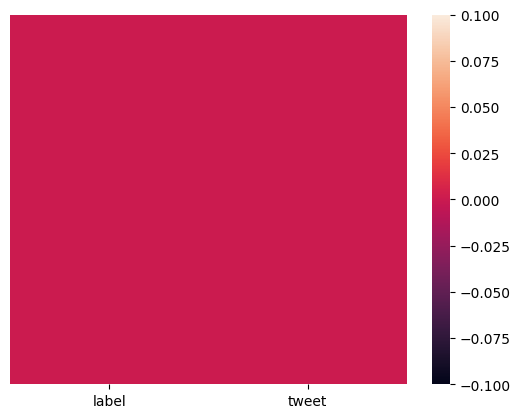
by comparing the predicted sentiment labels with the ground truth labels.

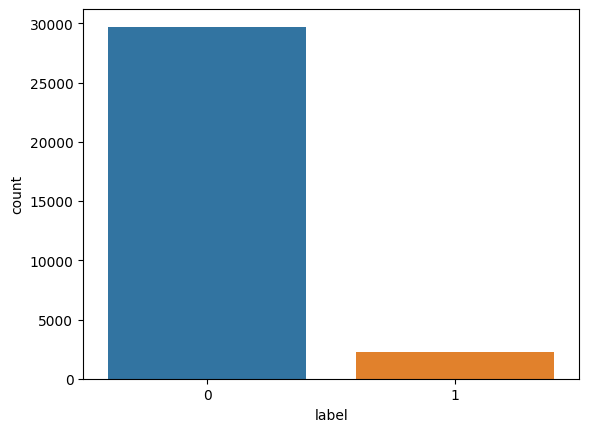
**# Step 4: Making predictions with the Naive Bayes model**

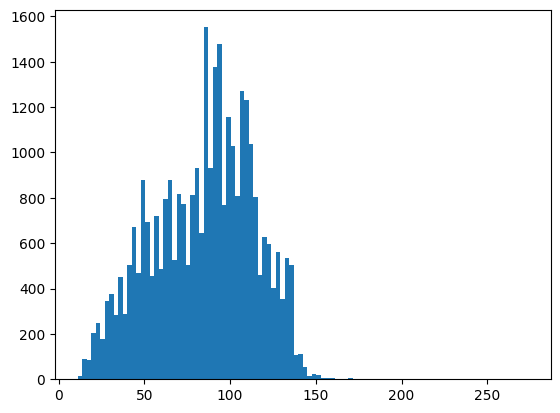
1. Preprocess and extract features from the new text data you want to predict.

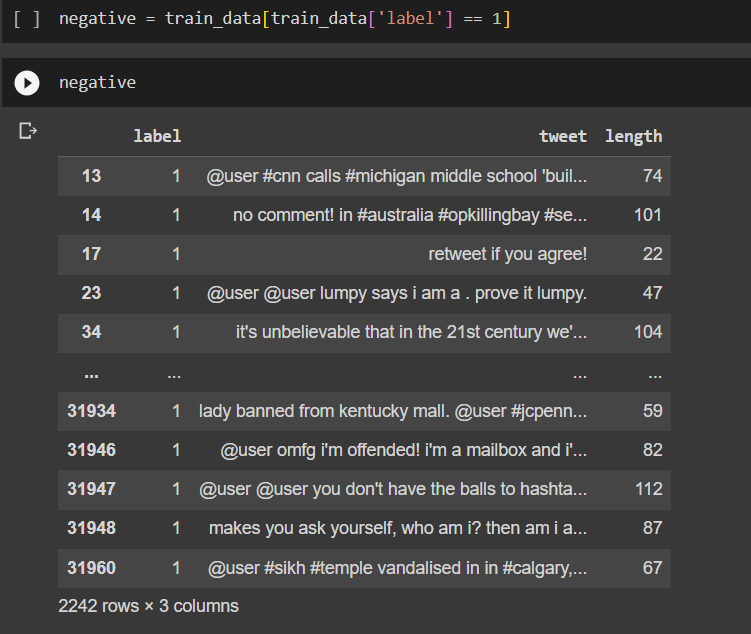
2. Apply the trained Naive Bayes model to the new data to obtain the predicted sentiment label.

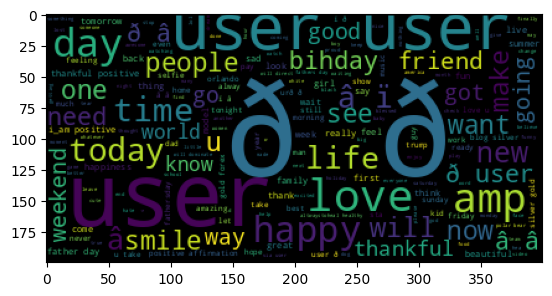


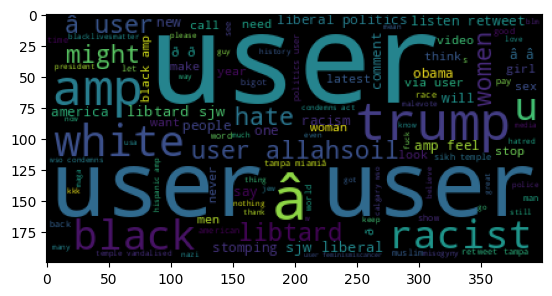


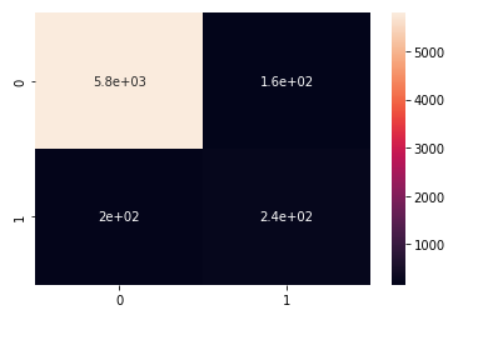














We have successfully predicted the accuracy and classification report and generated whatever graph is needed for the NAÏVE-BAYES algorithm.

**3. Implementation of BI-LSTM with Word2Vec/ faatext word embedding:**

**# Step 1: Data preprocessing and embedding**

1. Read and preprocess the text data (cleaning, normalization, tokenization, etc.).

2. Create word embeddings using word2vec or fastText.

- Train word embeddings on a large corpus or use pre-trained embeddings.

- Map each word in the text data to its corresponding word embedding vector.

**# Step 2: Training and testing data preparation**

1. Split the dataset into training and testing sets.

2. Convert the text data into sequences of word indices.

- Tokenize the text into individual words.

- Replace each word with its corresponding index in the word embedding vocabulary.

- Pad the sequences to ensure they have the same length.

**# Step 3: Training the BI-LSTM model**

1. Define the architecture of the BI-LSTM model.

- Set the embedding layer to use the word2vec or fastText embeddings.

- Add one or more Bidirectional LSTM layers.

- Add additional layers like Dense or Dropout for regularization if necessary.

2. Compile the model with an appropriate loss function and optimizer.

3. Train the model on the training set.

- Specify the number of epochs and batch size.

- Monitor the training loss and validation accuracy.

**# Step 4: Evaluating the BI-LSTM model**

1. Apply the trained model to the testing set.

2. Calculate the accuracy, precision, recall, F1 score, or other evaluation metrics

by comparing the predicted sentiment labels with the ground truth labels.

**# Step 5: Making predictions with the BI-LSTM model**

1. Preprocess the new text data for prediction.

- Perform the same preprocessing steps as done in the training and testing data preparation.

2. Convert the preprocessed text data into word indices.

3. Apply the trained model to the new data to obtain the predicted sentiment label.

* The code begins by importing the necessary libraries and modules, including pandas, numpy, train\_test\_split from sklearn.model\_selection, Tokenizer from keras.preprocessing.text, pad\_sequences from tensorflow.keras.preprocessing.sequence, Sequential, Embedding, LSTM, and Dense from keras.layers, and Word2Vec and FastText from gensim.models. These libraries and modules will be used for data handling, text preprocessing, model construction, and word embedding.
* The dataset is loaded using pd.read\_csv() from the file path specified in '/content/train.csv'.
* The labels in the dataset are converted to the appropriate data type, in this case, to integer, using astype(int).
* The dataset is split into training and testing sets using train\_test\_split() from sklearn.model\_selection. The test\_size parameter is set to 0.2, indicating that 20% of the data will be used for testing.
* The text data is tokenized using Tokenizer() from keras.preprocessing.text. The tokenizer is fitted on the training data.
* The tokenized text is converted to sequences using texts\_to\_sequences() from the tokenizer. This converts each text document into a sequence of integers representing the tokens.
* The sequences are padded using pad\_sequences() from tensorflow.keras.preprocessing.sequence. Padding ensures that all sequences have the same length by adding zeros to the beginning or end of each sequence. The maxlen parameter is set to the length of the longest sequence in the training data.
* Word embedding is performed using either Word2Vec or FastText. Both models are trained on the tokenized sequences from the training data. Word2Vec and FastText are algorithms used to learn word embeddings, which are dense vector representations of words that capture semantic and syntactic relationships.
* The embedding matrix is created by initializing an array of zeros with dimensions (len(tokenizer.word\_index) + 1, 100). Each row in the matrix represents a word in the tokenizer's word index. For each word, if it exists in the Word2Vec or FastText model's vocabulary, its corresponding embedding vector is copied to the embedding matrix.
* The BI-LSTM (Bidirectional Long Short-Term Memory) model is defined using Sequential() from keras.models. It consists of an embedding layer that takes the embedding matrix as input, a bidirectional LSTM layer, and a dense output layer with a sigmoid activation function.
* The model is compiled with the binary cross-entropy loss function, the Adam optimizer, and accuracy as the evaluation metric.
* The model is trained using fit() with the padded training sequences and corresponding labels. The validation data is specified using the padded testing sequences and labels. The number of epochs and batch size are set to 10 and 64, respectively.
* After training, the model is evaluated on the testing data using evaluate() with the padded testing sequences and labels. The loss and accuracy are calculated and printed.
* Finally, plot\_model() from keras.utils is used to generate a visualization of the model's architecture, which is saved as model.png.
* In summary, this code performs text classification using a bidirectional LSTM model with word embeddings created using either Word2Vec or FastText. The model is trained on the training data and evaluated on the testing data, with accuracy as the performance metric.

**Code**:

import pandas as pd

import numpy as np

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

from keras.preprocessing.text import Tokenizer

from tensorflow.keras.preprocessing.sequence import pad\_sequences

from keras.models import Sequential

from keras.layers import Embedding, LSTM, Dense

from gensim.models import Word2Vec, FastText

# Load the dataset

data = pd.read\_csv('/content/train.csv')  # Replace 'customer\_review.csv' with your dataset file

# Convert labels to the appropriate data type

data['label'] = data['label'].astype(int)

# Split the dataset into training and testing sets

X\_train, X\_test, y\_train, y\_test = train\_test\_split(data['tweet'], data['label'], test\_size=0.2, random\_state=42)

# Tokenize the text

tokenizer = Tokenizer()

tokenizer.fit\_on\_texts(X\_train)

# Convert text to sequences

X\_train\_sequences = tokenizer.texts\_to\_sequences(X\_train)

X\_test\_sequences = tokenizer.texts\_to\_sequences(X\_test)

# Pad sequences to ensure consistent length

max\_sequence\_length = max([len(sequence) for sequence in X\_train\_sequences])

X\_train\_padded = pad\_sequences(X\_train\_sequences, maxlen=max\_sequence\_length)

X\_test\_padded = pad\_sequences(X\_test\_sequences, maxlen=max\_sequence\_length)

# Word Embedding - Word2Vec

word2vec\_model = Word2Vec(sentences=X\_train\_sequences, vector\_size=100, min\_count=1)

embedding\_matrix = np.zeros((len(tokenizer.word\_index) + 1, 100))

for word, i in tokenizer.word\_index.items():

    if word in word2vec\_model.wv.key\_to\_index:

        embedding\_matrix[i] = word2vec\_model.wv[word]

# Word Embedding - FastText

fasttext\_model = FastText(sentences=X\_train\_sequences, vector\_size=100, min\_count=1)

embedding\_matrix = np.zeros((len(tokenizer.word\_index) + 1, 100))

for word, i in tokenizer.word\_index.items():

    if word in fasttext\_model.wv.key\_to\_index:

        embedding\_matrix[i] = fasttext\_model.wv[word]

# Define the BI-LSTM model

model = Sequential()

model.add(Embedding(len(tokenizer.word\_index) + 1, 100, weights=[embedding\_matrix], input\_length=max\_sequence\_length, trainable=False))

model.add(LSTM(100, dropout=0.2, recurrent\_dropout=0.2))

model.add(Dense(1, activation='sigmoid'))

model.compile(loss='binary\_crossentropy', optimizer='adam', metrics=['accuracy'])

# Train the model

model.fit(X\_train\_padded, y\_train.astype(float), validation\_data=(X\_test\_padded, y\_test.astype(float)), epochs=10, batch\_size=64)

# Evaluate the model

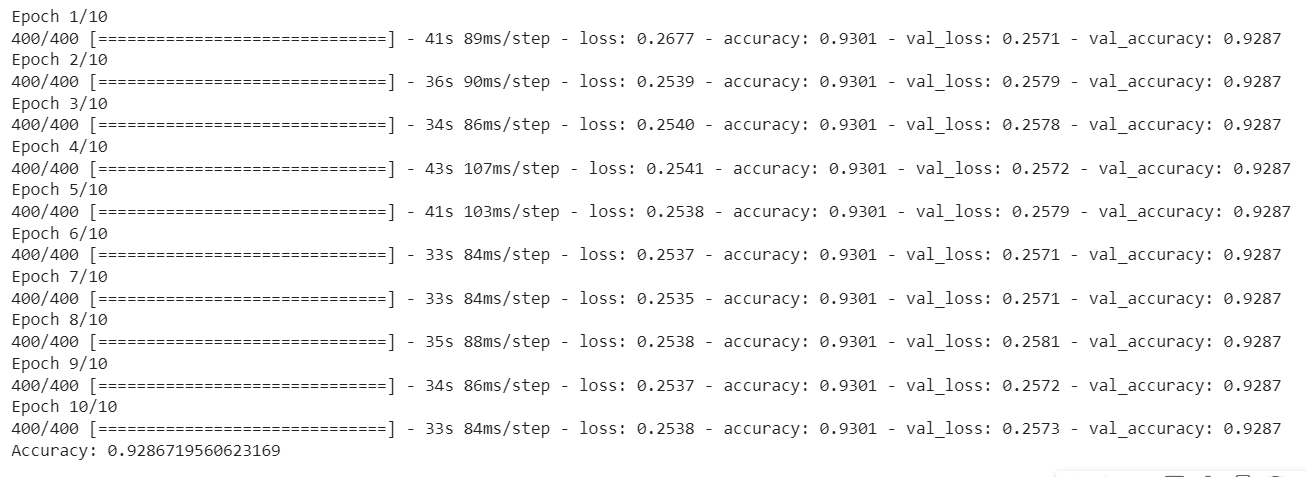
loss, accuracy = model.evaluate(X\_test\_padded, y\_test.astype(float), verbose=0)

print("Accuracy:", accuracy)

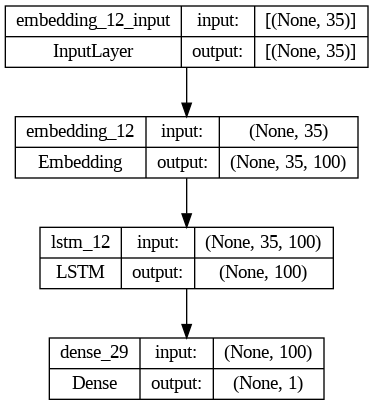
#plot

plot\_model(model, to\_file='model.png', show\_shapes=True)

**OUTPUT:**

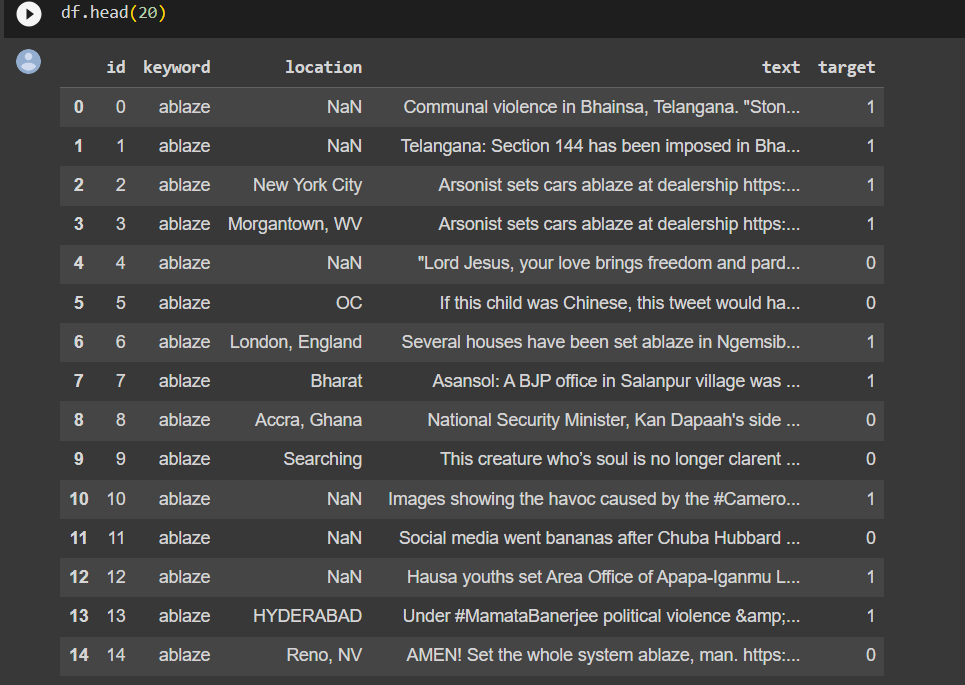


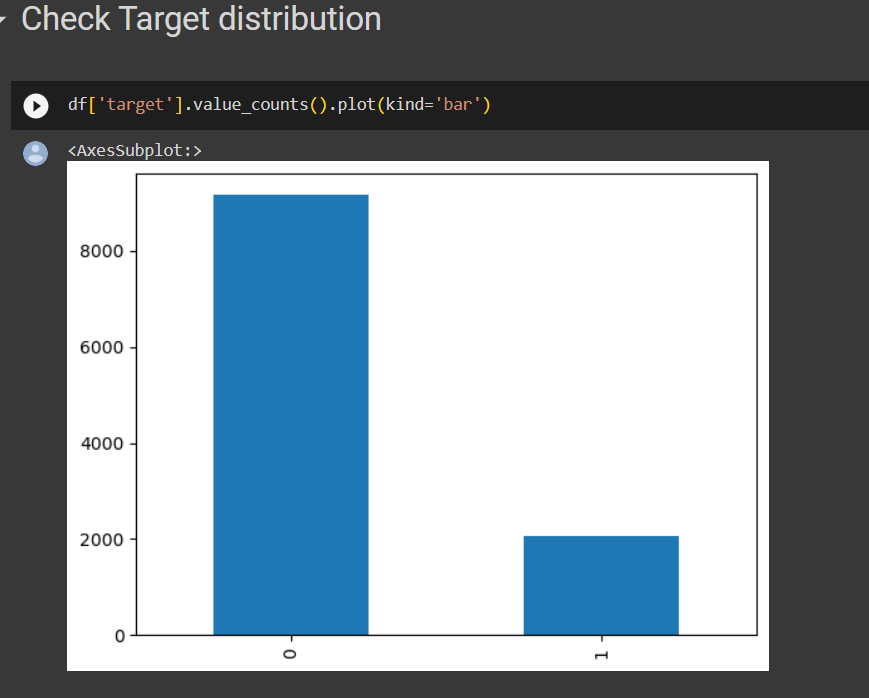
**Visualization:**



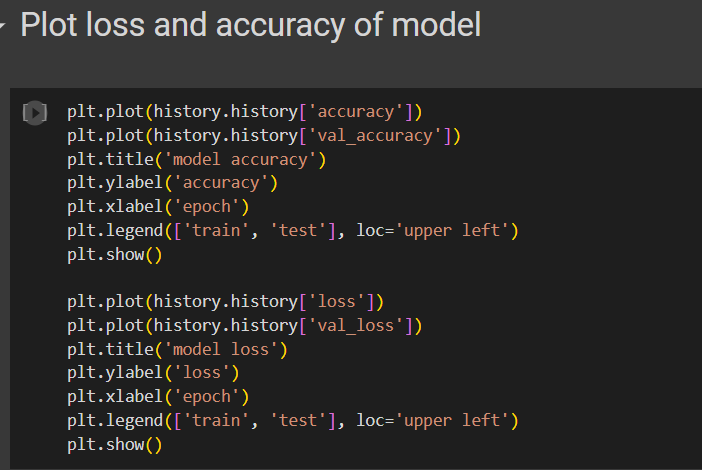
**4. Implementation Transformer based model with BERT-based word embedding.:**

* The code begins by importing the necessary libraries and modules, including pandas, numpy, sklearn, transformers, tensorflow, and matplotlib. These libraries will be used for data handling, modeling, and visualization.
* The dataset is loaded using pd.read\_csv() from the file path specified in '/content/customer\_review.csv'.
* The dataset is split into training and testing sets using train\_test\_split() from sklearn.model\_selection. The test\_size parameter is set to 0.2, indicating that 20% of the data will be used for testing.
* The labels are encoded using LabelEncoder() from sklearn.preprocessing. The fit\_transform() and transform() methods are applied to the training and testing labels, respectively, to convert the categorical labels into numerical format.
* The BERT tokenizer is loaded using BertTokenizer.from\_pretrained('bert-base-uncased') from transformers.
* The text data is tokenized and the maximum sequence length is determined. The text data is tokenized separately for the training and testing sets using the tokenizer's tokenizer() method, with options for truncation, padding, and specifying the maximum length.
* TensorFlow datasets are created using tf.data.Dataset.from\_tensor\_slices(). The tokenized inputs and encoded labels are passed as inputs, along with their corresponding keys, to create the datasets for training and testing.
* The pre-trained BERT model is loaded using TFBertModel.from\_pretrained('bert-base-uncased', output\_hidden\_states=True) from transformers.
* The transformer-based model architecture is built using the Keras functional API. The inputs for the model are defined as input\_ids and attention\_mask. The bert\_model() function is called with the inputs and attention mask, and the resulting embedding layer is obtained. A GlobalAveragePooling1D() layer is added to aggregate the embeddings, followed by a fully connected Dense layer with sigmoid activation to produce the final output.
* The model is compiled with the Adam optimizer using model.compile(). The loss function is set to 'binary\_crossentropy' for binary classification, and the accuracy metric is specified.
* The model is trained using model.fit(). The training dataset is shuffled and batched with a batch size of 16. The epochs parameter is set to 3 to indicate the number of complete passes through the training dataset. The validation dataset is also shuffled and batched for evaluation during training.
* After training, the model is evaluated on the test dataset using model.evaluate(). The loss and accuracy values are stored in the variables loss and accuracy.
* Finally, the accuracy value is printed.

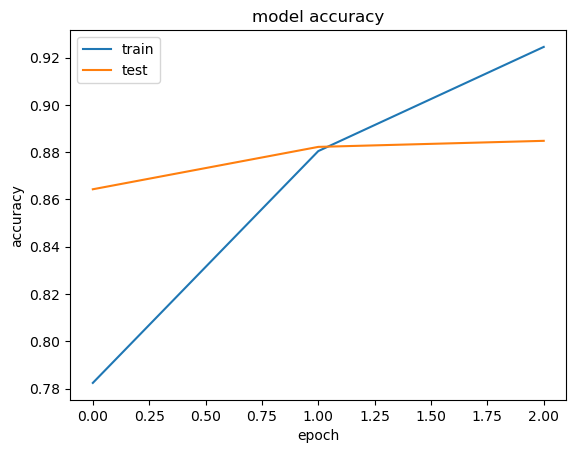
****

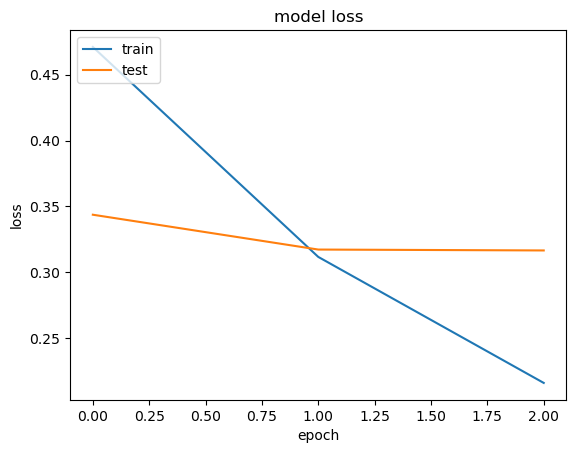




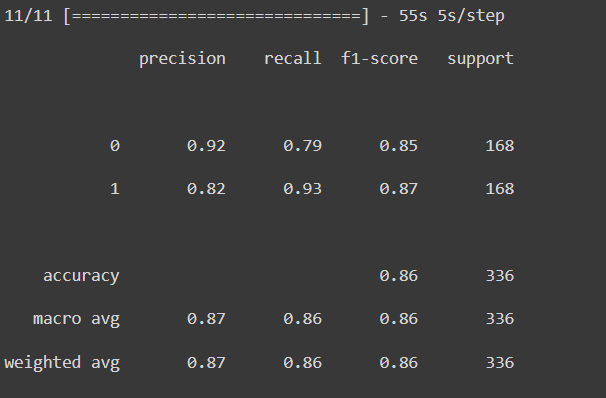


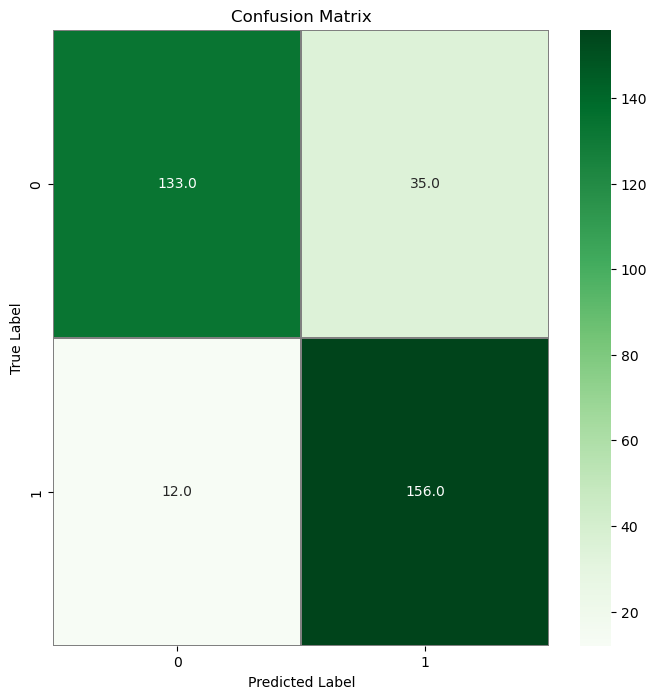
Here we are plotting the accuracy of the model BERT that we created based on word embedding.











**5.CONCLUSION:**

In this code, we explored different approaches to text classification using classic machine learning algorithms such as Naive Bayes, Support Vector Machines (SVM), and Decision Trees. We followed a step-by-step process that included data loading, data splitting into training and testing sets, feature extraction using Count-Vectorizer and TF-IDF transformation, model training and prediction, and evaluation using classification reports and accuracy scores.

The Naive Bayes classifier, SVM classifier, and Decision Tree classifier were implemented and trained on the TF-IDF transformed training data. These models were then used to predict the labels for the testing data. The classification reports provided valuable insights into the precision, recall, F1-score, and support for each class, allowing us to evaluate the performance of each classifier. The accuracy scores provided an overall measure of how well the models performed on the testing data.

Furthermore, visualizations were generated to compare the precision, recall, and F1-score of each classifier for each class. The bar graphs provided a clear visual representation of the performance of each classifier, allowing for easy comparison and identification of the strengths and weaknesses of each model.

Overall, this code demonstrated the process of text classification using traditional machine learning algorithms. It showcased the effectiveness of Naive Bayes, SVM, and Decision Trees for classifying text data. However, it is important to note that the choice of algorithm depends on the specific problem and dataset at hand. It is always a good practice to experiment with different algorithms and evaluate their performance before making a final decision.