A PROJECT ON
NATURAL LANGUAGE PROCESSING OF RESTRAUNT REVIEWS

REPORT SUBMITTED BY

NAME OF STUDENTS:

ANKITA PRAKASH AND SWARNAVA MAJUMDER

PRN NUMBER:

23060641003 AND 23060641092

UNDER THE GUIDANCE OF

DR. PRIYA DESHPANDE

SYMBIOSIS STATISTICAL INSTITUTE

COURSE NAME

LINEAR MODELS

Contents

Abstract

In this study, we evaluated the performance of various machine learning models, including Logistic Regression, Kernel Support Vector Machine (SVM), and K-Nearest Neighbors (KNN), for predicting a certain outcome. Our analysis revealed that Logistic Regression yielded the highest accuracy of 80.5%, followed closely by Kernel SVM with an accuracy of 80%. However, the KNN model performed relatively poorly, achieving only a 70% accuracy rate.

We further examined the Logistic Regression model, analyzing its coefficients to understand the impact of different features on the prediction outcome. Our findings indicated that certain features positively influenced the prediction outcome, while others had a negative impact.

To validate the robustness of our Logistic Regression model, we employed k-Fold Cross Validation, which demonstrated consistent performance across different folds with an average accuracy of 79.88% and a standard deviation of 5.82%.

Moreover, we conducted a thorough investigation by splitting the dataset into two equal parts and performing separate analyses on each subset. Our results showed that the Logistic Regression model maintained a similar level of accuracy across both subsets, indicating its stability and reliability.

Overall, our study underscores the effectiveness of Logistic Regression as a predictive modeling technique and highlights its potential applications in real-world scenarios requiring accurate classification tasks.

Acknowledgement

This paper and the research might no longer had been feasible without the great assist of our guide Dr. Priya Deshpande. Her enthusiasm, expertise and exacting interest to detail had been an idea and saved my work on course from my first encounter with the ground level work and with several journals to the final draft of this paper.

She not only simply furnished the entire project with various information, however abruptly shared the worthwhile data on this precise subject matter. The generosity and information and steady motivation have improved this look at in innumerable approaches and stored me from many errors.

Introduction

Natural Language Processing, or NLP for short, is like teaching computers to understand and work with human language, just like we do. Think of it as bridging the gap between how humans communicate and how computers understand things.

1. It breaks down the structure of language to find out what people are saying.
2. NLP algorithms can analyze and manipulate text.
3. It also helps computers to generate human-like text and also helps computers to communicate with people more naturally.

It is a subset of artificial intelligence, computer science and focused on making human communication such as speech and text comprehensible to computers.
Overall, NLP is all about teaching computers to understand, interpret, and generate human language, which opens up a ton of possibilities for making computers more useful and accessible in our everyday lives.

Problem Statement

The objective of this project is to analyze a dataset comprising 1000 customer reviews regarding a restaurant, aiming to discern the overall sentiment expressed by customers. Through the utilization of natural language processing (NLP) techniques, sentiment analysis algorithms, and machine learning models, our goal is to extract valuable insights into customer preferences, pinpoint common themes within positive and negative reviews, and ultimately provide actionable recommendations geared towards enhancing the restaurant's service quality and overall customer experience. Additionally, the study seeks to develop a system capable of identifying and categorizing text messages to ascertain whether individuals express favorable or unfavorable sentiments towards the restaurant based on their feedback.

Methodology

- Importing suitable libraries and understanding the data
- Exploratory Data Analysis
- Bag of words model
- Model fitting and Comparison

Dataset

Dataset: [Ctrl and click on the link at the same time to open the file]
https://drive.google.com/file/d/1Z_a0OasRVZLBK-GWiJ2gB6WsSjj_7mNe/view?usp=sharing

Dataset characteristics	Text
No. of instances	1000
No. of attributes	2
Associated tasks	Natural Language Processing
Missing values	N/A
Feature characteristics	Text and Numeric

In the data named 'Restaurant_Reviwes' which is a tab separated value(tsv) file. The dataset is divided into two columns the first column is the 'Review' column in which thousands of customers have given their review in the form of text messages and the second column is the 'Liked' column if they liked the restaurants or not which is given by '1' meaning they liked it and '0' meaning they didn't like it.

Code

[Ctrl and click on the link at the same time to open the file]
https://drive.google.com/file/d/1-5OcKtSA6IiKFs9qpdTG-tYeZ7bEE39X/view?usp=sharing

Output

A.STEPS INVOLVED IN PREPROCESSING DATA OF NATURAL LANGUAGE PROCESSING.

➢ Importing Libraries:

- import numpy as np: Imports the NumPy library and aliases it as np. NumPy is used for numerical computing in Python.
- import matplotlib.pyplot as plt: Imports the pyplot module from the Matplotlib library and aliases it as plt. Matplotlib is a plotting library for Python.
- import pandas as pd: Imports the pandas library and aliases it as pd. Pandas is a powerful data analysis and manipulation library for Python.

➢ Importing the Dataset:

- dataset = pd.read_csv('Restaurant_Reviews.tsv', delimiter = '\t', quoting = 3): Reads the dataset from a tab-separated values (TSV) file named 'Restaurant_Reviews.tsv'. The delimiter parameter specifies that the values in the file are separated by tabs ('\t'). The quoting parameter is set to 3, which treats all quotes as regular characters, ignoring them as part of the quoting mechanism for fields.

➢ Importing spaCy:

- import spacy: Imports the spaCy library, which is a popular natural language processing (NLP) library in Python.
- SpaCy provides pre-trained models for various languages and tasks, including tokenization, part-of-speech tagging, named entity recognition, and more.

➢ Downloading the English Language Model:

- spacy.cli.download("en_core_web_sm"): This line downloads the English language model named "en_core_web_sm" provided by spaCy. This model is a small-sized model trained on web text data and includes vocabulary, syntax, named entities, and word vectors.

➢ Loading the Downloaded Model:

- nlp = spacy.load("en_core_web_sm"): After downloading the English language model, this line loads the model into memory and initializes it as an NLP pipeline. It creates an instance of the English language class provided by spaCy, configured with the "en_core_web_sm" model.

➢ Importing Libraries:

- import re: Imports the Python module for regular expressions, which is used for pattern matching and manipulation of strings.
- import nltk: Imports the Natural Language Toolkit (NLTK), a library for natural language processing tasks in Python.

➢ Downloading NLTK Resources:

- nltk.download('stopwords'): Downloads the NLTK stopwords corpus, which contains common stopwords for various languages. Stopwords are words that are considered non-informative and are often removed during text preprocessing.

➢ Importing NLTK Resources:

- from nltk.corpus import stopwords: Imports the stopwords corpus from NLTK, which was just downloaded. These stopwords will be used later for filtering out common words from the text.

➢ Initializing Word Lemmatizer:

- lemmatizer = WordNetLemmatizer(): Creates an instance of the WordNet lemmatizer from NLTK. Lemmatization is the process of reducing words to their base or root form.

➢ Text Preprocessing Loop:

- Created a loop which iterates over each review in the dataset DataFrame.
- re.sub('[^a-zA-Z\s]', '', dataset['Review'][i]): Removes non-alphabetic characters and retains spaces in the review text using regular expressions.

- re.sub(r'\b\d+\b', '', review): Removes standalone numbers from the review text.
- review.lower(): Converts the review text to lowercase.
- review.split(): Splits the review text into individual words or tokens based on whitespace.
- all_stopwords = set(stopwords.words('english')): Retrieves all English stopwords from NLTK.
- all_stopwords.remove('not'): Removes all stop words except 'not' from the set of stopwords.
- word for word in review if word.lower() not in all_stopwords: Removes stop words except 'not' from the review text.
- nlp(review): Applies the spaCy NLP pipeline to the preprocessed review text.
- token.lemma_ for token in review: Lemmatizes each token in the review text using spaCy.
- re.sub(r'\s+', ' ', review): Replaces multiple spaces with a single space in the processed review text.
- The preprocessed review is appended to the corpus list.

➢ Printing the Corpus:

- print(corpus): Prints the preprocessed corpus, which contains the cleaned and lemmatized versions of all reviews in the dataset.

➔ **Output:** Corpus Values.pdf

B. STEPS INVOLVED IN BAG OF WORDS MODEL.

➢ Bag of words model:

- The Bag of Words (BoW) model is a way of representing text data in machine learning. It represents text as a matrix where rows correspond to documents (reviews in this case) and columns correspond to unique words in the vocabulary.
- Each cell in the matrix represents the frequency of occurrence of a word in a document.

➢ Importing Libraries:

from sklearn.feature_extraction.text import CountVectorizer: Imports the CountVectorizer class from scikit-learn, which is used to convert text data into a matrix of token counts.

➢ Initializing CountVectorizer:

- cv = CountVectorizer(): Creates an instance of the CountVectorizer class. By default, it will convert text to lowercase and tokenize the text (splits it into individual words or tokens), and builds a vocabulary of all unique words in the corpus.

➢ Fitting and Transforming the Corpus:

- X = cv.fit_transform(corpus).toarray(): This line fits the CountVectorizer to the preprocessed corpus (corpus) and then transforms it into a matrix of token counts.
- fit_transform() method learns the vocabulary dictionary from the corpus and returns a document-term matrix, where rows correspond to documents (reviews) and columns correspond to words in the vocabulary.
- toarray() converts the sparse matrix returned by fit_transform() into a dense matrix representation.

➢ Assigning Labels:

- y = dataset.iloc[:, -1].values: Assigns labels to the y variable. It extracts the last column of the dataset DataFrame, it contains the labels for classification.

➢ Getting Vocabulary Dictionary:

- vocabulary = cv.vocabulary: Retrieves the vocabulary dictionary learned by the CountVectorizer. The vocabulary dictionary maps each word to its index in the Bag of Words representation.

➢ Counting the Number of Unique Words:

- num_unique_words = len(vocabulary): Counts the number of unique words in the vocabulary dictionary, which corresponds to the number of columns in the Bag of Words matrix.

➢ Printing the Result:

- print("Number of unique words in the Bag of Words model:", num_unique_words): Prints the number of unique words in the Bag of Words model.

➔ **Output:** Number of unique words in the Bag of Words model: 1653.

➔ **Interpretation:** There are all total 1653 total unique words in our bag of words model.

➢ Calculating Word Frequencies:

- word_frequencies = np.sum(X, axis=0): Calculates the total frequency of each word in the corpus by summing up the occurrences of each word across all documents. axis=0 specifies that the summation should be done along the rows (i.e., for each word).

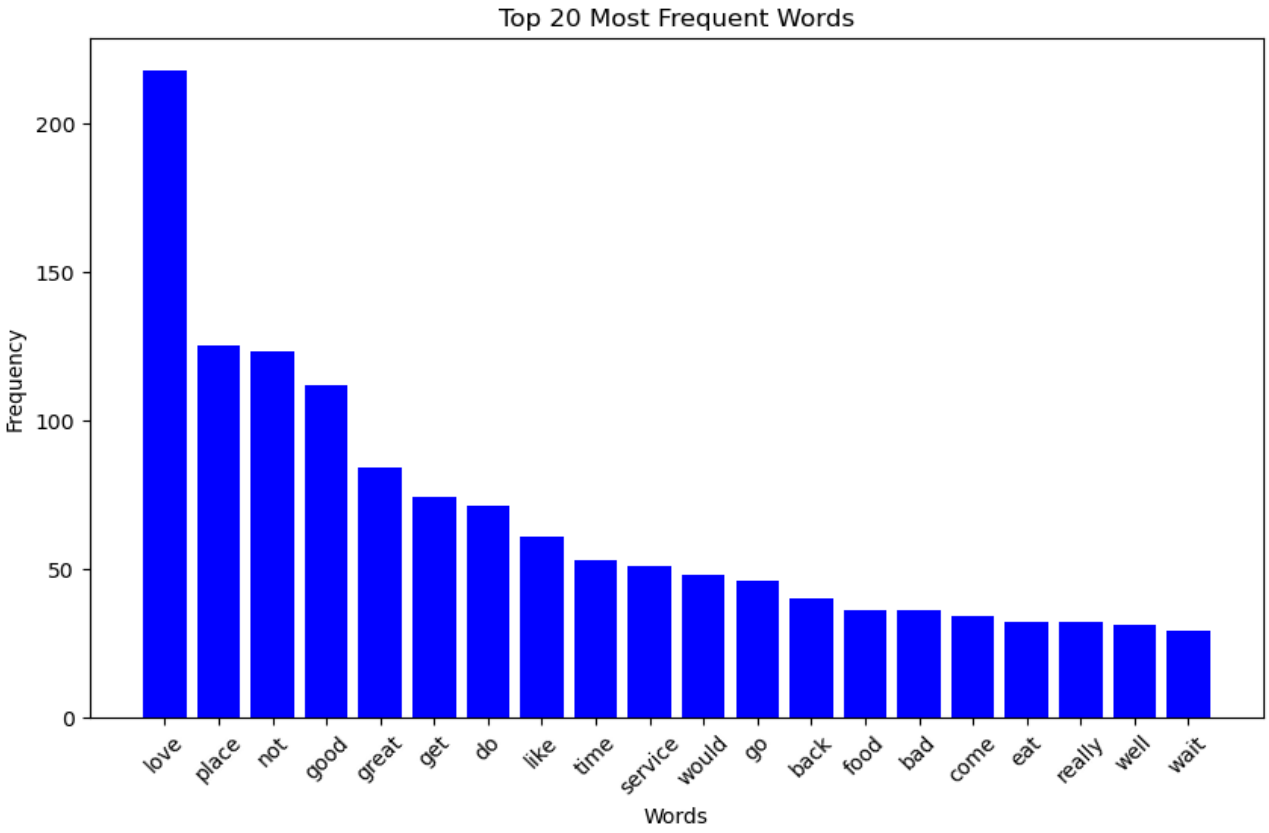
➢ Getting Indices of Top 20 Most Frequent Words:

- top_indices = np.argsort(word_frequencies)[-20:][::-1]: Sorts the word frequencies in descending order and retrieves the indices of the top 20 most frequent words. np.argsort() returns the indices that would sort the array, and [-20:] selects the last 20 indices (i.e., the top 20 most frequent words), and [::-1] reverses the order to get the indices of the most frequent words first.

➢ Plotting Histogram:

- Showing the top 20 most frequent words and their frequencies.
- plt.figure(figsize=(10, 6)): Sets the figure size to 10 inches wide and 6 inches tall.
- plt.bar(top_words, top_frequencies, color='blue'): Plots a bar chart with words on the x-axis and their frequencies on the y-axis. The bars are colored blue.
- plt.xlabel('Words'): Sets the label for the x-axis as "Words".
- plt.ylabel('Frequency'): Sets the label for the y-axis as "Frequency".
- plt.title('Top 20 Most Frequent Words'): Sets the title of the plot.

➔ **Output:**



➔ **Interpretation:** Here the word ‘love’ is the most used word among the 1653 other words used by the people who visited the restaurants accounting for more than 200 times followed by ‘place’, ‘not’, ‘good’ which are also more than 100 times.

1. STEPS INVOLVED IN FITTING NAIVE BAYES CLASSIFIER.

➢ Naive Bayes:

- A Naive Bayes classifier is a simple probabilistic classifier based on Bayes' theorem with the "naive" assumption of independence between features.
- Types of Naive Bayes Classifiers:
  - ★ Multinomial Naive Bayes: Suitable for features that represent counts or frequencies (e.g., word counts in text classification).
  - ★ Gaussian Naive Bayes: Assumes that features follow a normal distribution. It's suitable for continuous features.

★ Bernoulli Naive Bayes: Used for features that are binary-valued (e.g., presence or absence of a term in text).

- In our data we have proceeded with Gaussian Naive Bayes assuming it follows normal distribution.

➢ Splitting the Dataset:

- train_test_split(X, y, test_size = 0.20, random_state = 0): Splits the dataset into training and testing sets. It takes input features X and labels y as input, with test_size = 0.20 indicating that 20% of the data will be used for testing, and random_state = 0 sets the random seed for reproducibility.

➢ Training the Naive Bayes Model:

- from sklearn.naive_bayes import GaussianNB: Imports the Gaussian Naive Bayes classifier from scikit-learn.
- classifier = GaussianNB(): Creates an instance of the Gaussian Naive Bayes classifier.
- classifier.fit(X_train, y_train): Trains the Gaussian Naive Bayes classifier on the training data (X_train and y_train).

➢ Predicting Test Set Results:

- y_pred = classifier.predict(X_test): Predicts the class labels for the test set using the trained classifier (classifier.predict).
- print(np.concatenate((y_pred.reshape(len(y_pred),1),y_test.reshape(len(y_test),1)),1)): Concatenates the predicted labels (y_pred) and the actual labels (y_test) vertically to compare them.

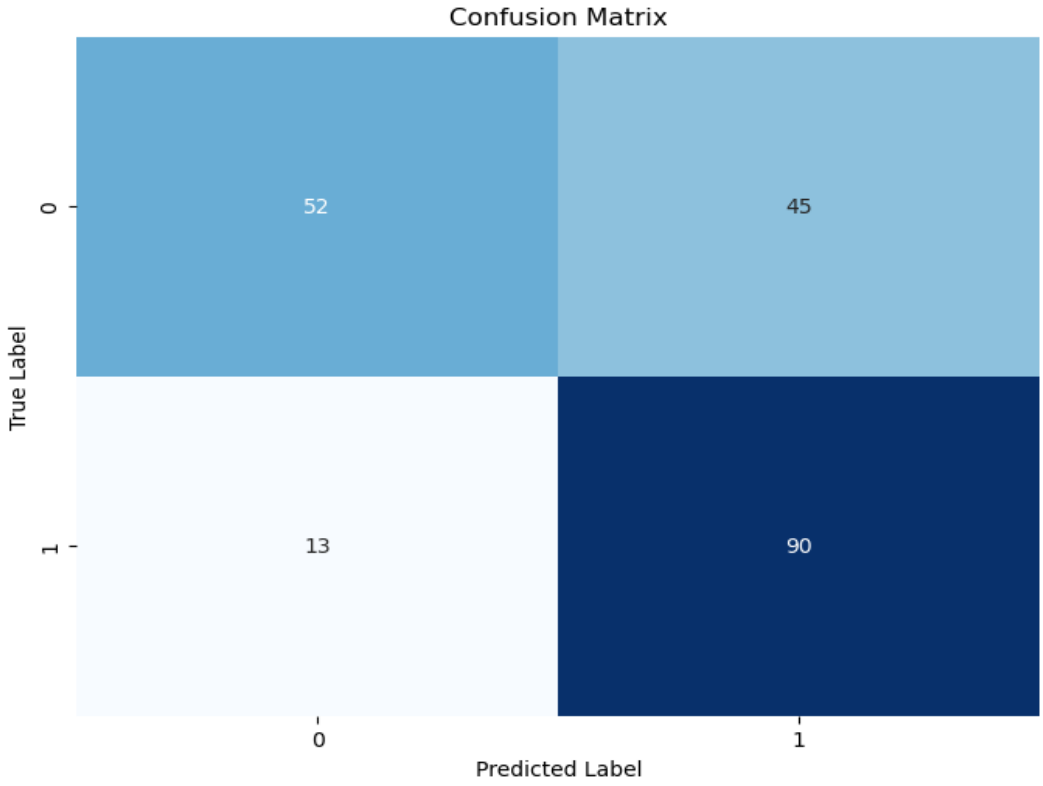
➔ **Output:** Total number of values: 200, Number of matching predicted and actual values: 142, Number of values that did not match: 58.

➔ **Interpretation:** In each row the first value represents the predicted and the second value represents the actual label. Similarly, the pattern goes on for all the rows. Most of our predicted values are matching with actual values. This shows that our model is performing well with less discrepancies.

➢ Confusion Matrix:

- from sklearn.metrics import confusion_matrix, accuracy_score: Imports the functions to calculate the confusion matrix and accuracy score.
- cm = confusion_matrix(y_test, y_pred): Computes the confusion matrix based on the actual and predicted labels.
- print(cm): Prints the confusion matrix.
- accuracy_score(y_test, y_pred): Computes the accuracy score, which is the proportion of correctly classified samples.

➔ **Output:** Accuracy - 0.71



➔

➔ **Interpretation:** Looking at the above table we can infer that:

★ True Positives: The model has correctly predicted 52 values as positive (1) and they were actually positive (1).

★ False Positives: The model predicted 45 values as positive (1) and they were actually negative (0).

★ False Negative: The model predicted 13 values as negative (0) and they were actually positive (1).

★ True Negative: The model has correctly predicted 90 values as negative (0) and they were actually negative (0).

★ The accuracy is calculated by the formula **[(True Positive + True Negative)/Total number of values].**

= (52+90)/200*100

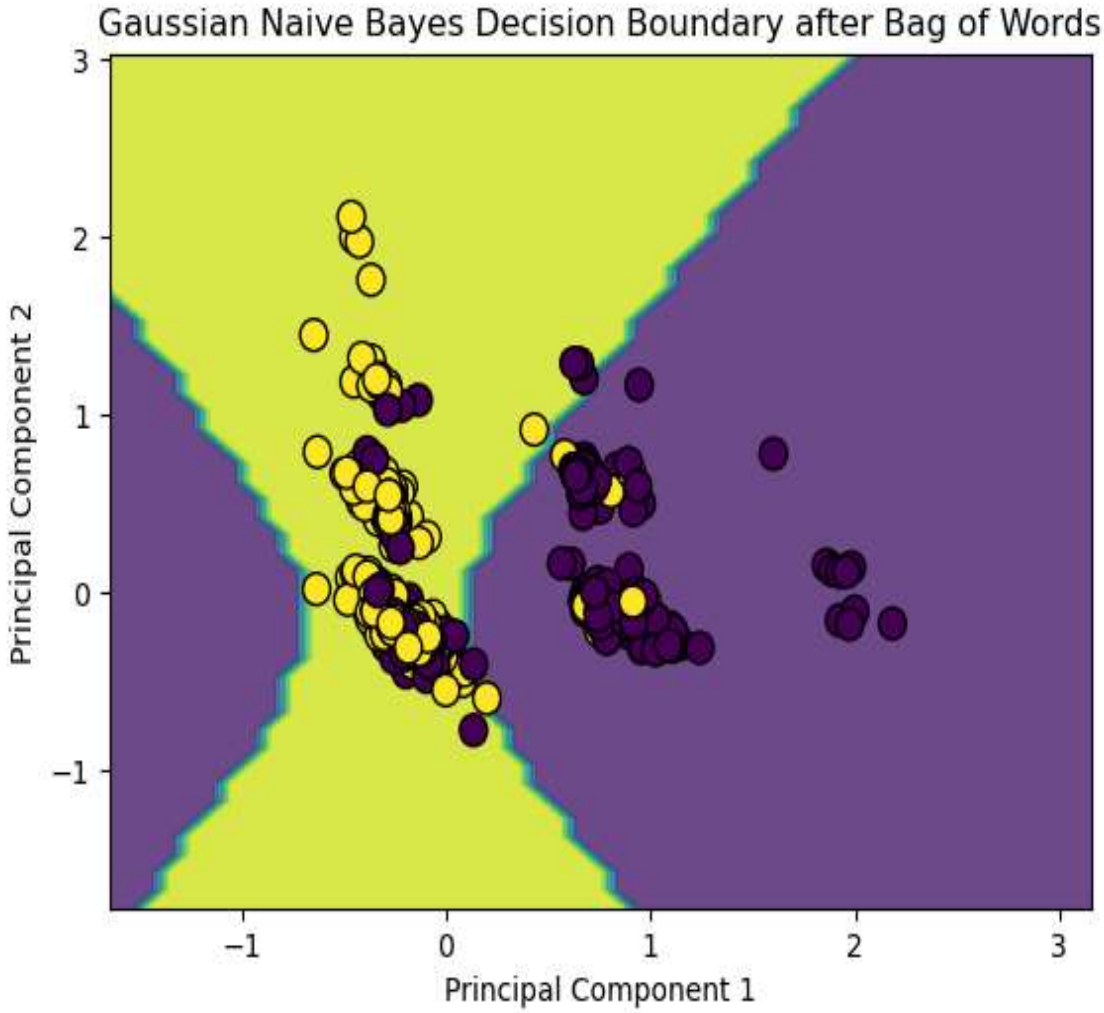
= 0.71*100

= 71.0%

It is coming out to be 71.0 %, which indicates that the model has correctly predicted 71.0% instances in the dataset.

➢ **Graphical Representation:**

➔ **Output:**



➔ **Interpretation:** We have reduced the data into 2 most explaining principal components and have plotted the graph in which the 'purple' areas are

representing that the person liked the restaurant (1) and the 'yellow' area is representing that the person didn't like the restaurant (0). While the 'purple' balls predicted that the person liked the restaurant (1) while the 'yellow' balls predicted that the person who didn't liked the restaurant (0). We could see that most of our predictions are correct leaving a few of them which makes a good model. There is no linear line acting as a differentiator between the level (1) and (0) as we know that naive bayes is a non-linear classifier.

2. STEPS INVOLVED IN FITTING LOGISTIC REGRESSION CLASSIFIER.

➢ Logistic Regression:

- Logistic Regression is a supervised learning algorithm used for binary classification tasks, where the target variable (also known as the dependent variable) is categorical and has only two possible outcomes (e.g., yes/no, true/false, 1/0). Despite its name, logistic regression is primarily used for classification rather than regression.
- The main idea behind logistic regression is to model the probability that a given input belongs to a particular class. It predicts the probability of the input belonging to the positive class (usually denoted as class 1) using a logistic function. The logistic function (also known as the sigmoid function) maps any input value to a value between 0 and 1, representing the probability of the input belonging to the positive class.

➢ Splitting the Dataset:

- train_test_split(X, y, test_size = 0.20, random_state = 0): Splits the dataset into training and testing sets. It takes input features X and labels y as input, with test_size = 0.20 indicating that 20% of the data will be used for testing, and random_state = 0 sets the random seed for reproducibility.

➢ Training the Logistic Regression Model:

- LogisticRegression (random_state = 0): Importing our function Logistic regression from Scikit-learn for training our model.The classifier

random_state ensures that the result will be the same throughout the code. After running this code, we will have our model ready to make predictions.

➢ Predicting Test Set Results:

- y_pred = classifier.predict(X_test): Predicts the class labels for the test set using the trained classifier (classifier.predict).
- print(np.concatenate((y_pred.reshape(len(y_pred),1),y_test.reshape(len(y_test),1)),1)): Concatenates the predicted labels (y_pred) and the actual labels (y_test) vertically to compare them.

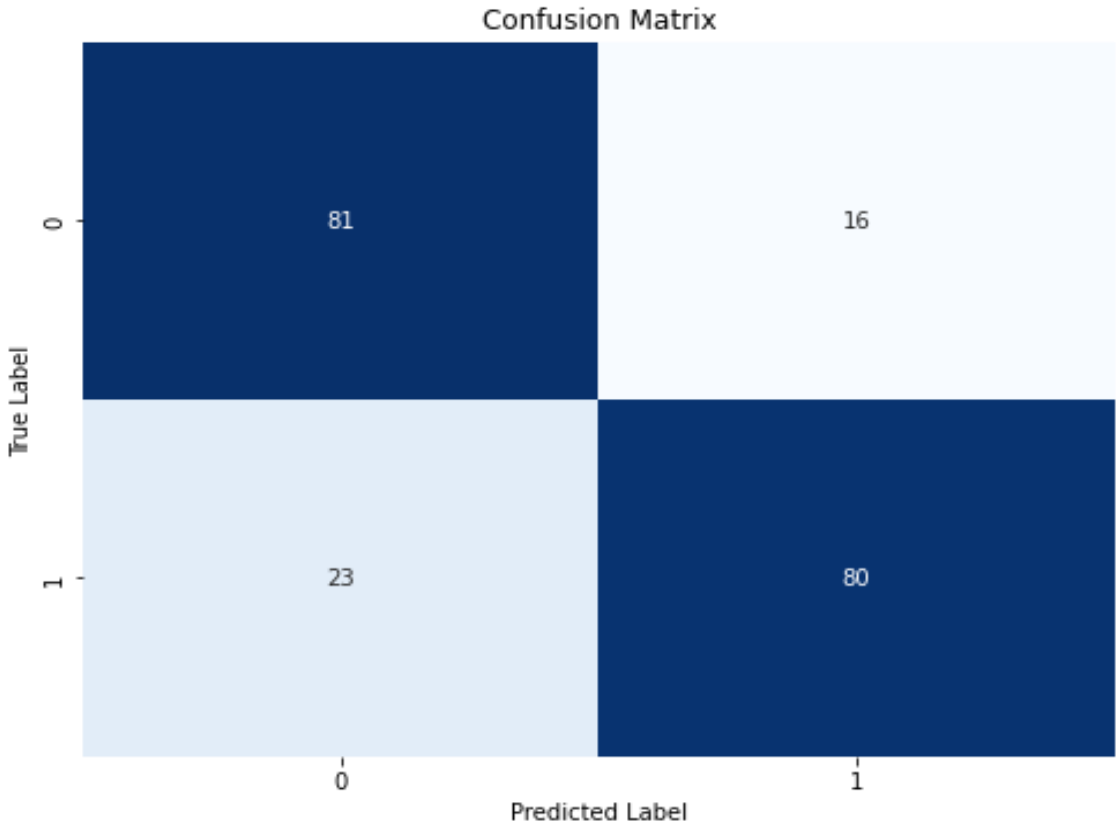
➔ **Output:** Total number of values: 200, Number of matching predicted and actual values: 161, Number of values that did not match: 39.

➔ **Interpretation:** In each row the first value represents the predicted and the second value represents the actual label. Similarly, the pattern goes on for all the rows. Most of our predicted values are matching with actual values. This shows that our model is performing well with less discrepancies.

➢ Confusion Matrix:

- from sklearn.metrics import confusion_matrix, accuracy_score: Imports the functions to calculate the confusion matrix and accuracy score.
- cm = confusion_matrix(y_test, y_pred): Computes the confusion matrix based on the actual and predicted labels.
- print(cm): Prints the confusion matrix.
- accuracy_score(y_test, y_pred): Computes the accuracy score, which is the proportion of correctly classified samples.

➔ **Output:** Accuracy - 0.805



➔ **Interpretation:** Looking at the above table we can infer that:

★ True Positives: The model has correctly predicted 81 values as positive (1) and they were actually positive (1).

★ False Positives: The model predicted 16 values as positive (1) and they were actually negative (0).

★ False Negative: The model predicted 23 values as negative (0) and they were actually positive (1).

★ True Negative: The model has correctly predicted 80 values as negative (0) and they were actually negative (0).

★ The accuracy is calculated by the formula **[(True Positive + True Negative)/Total number of values].**

= (81+80)/200*100

= 0.805*100

= 80.5%

It is coming out to be 80.5 %, which indicates that the model has correctly predicted 80.5% instances in the dataset. And logistic regression predicts better than naive bayes model whose accuracy is 71.0%.

➢ **Graphical Representation:**

➔ **Output:**

➔ **Interpretation:** We have reduced the data into 2 most explaining principal components and have plotted the graph in which the 'purple' area is representing that the person liked the restaurant (1) and the 'yellow' area is representing that the person didn't like the restaurant (0). While the 'purple' balls predicted that the person liked the restaurant (1) while the 'yellow' balls predicted that the person who didn't liked the restaurant (0). We could see that most of our predictions are correct leaving a few of them which makes a good model. There is a straight line acting as a differentiator between the level (1) and (0) as we know that logistic regression is a linear classifier.

3. STEPS INVOLVED IN K-NEAREST NEIGHBORS CLASSIFIER.

➢ K-Nearest Neighbors:

- K-Nearest Neighbors (KNN) is a simple yet powerful supervised machine learning algorithm used for classification and regression tasks. It is a non-parametric and lazy learning algorithm, meaning it does not make any assumptions about the underlying data distribution and does not learn a discriminative function from the training data. Instead, it memorizes the entire training dataset and makes predictions based on the similarity between new input data and previously seen examples.
- When given a new input data point, KNN calculates the distance between the new point and all points in the training dataset using a distance metric such as Euclidean distance, Manhattan distance, or cosine similarity. KNN then identifies the K nearest neighbors (data points with the smallest distances) to the new data point from the training dataset.

➢ Splitting the Dataset:

- train_test_split(X, y, test_size = 0.20, random_state = 0): Splits the dataset into training and testing sets. It takes input features X and labels y as input, with test_size = 0.20 indicating that 20% of the data will be used for testing, and random_state = 0 sets the random seed for reproducibility.

➢ Training the K-Nearest Neighbors Model:

- from sklearn.neighbors import KNeighborsClassifier: Imports the KNeighborsClassifier class from scikit-learn, which is used to implement the K-Nearest Neighbors algorithm for classification tasks.
- classifier = KNeighborsClassifier(n_neighbors = 10, metric = 'minkowski', p = 2): Creates an instance of the KNeighborsClassifier class.
- n_neighbors = 10: Specifies the number of neighbors (K) to consider when making predictions. In this case, it's set to 10.

- metric = 'minkowski': Specifies the distance metric used to measure the similarity between data points. Here, the Minkowski distance metric is used, which is a generalization of both the Euclidean distance (p = 2) and the Manhattan distance (p = 1).
- p = 2: Specifies the parameter for the Minkowski distance metric. When p = 2, it corresponds to the Euclidean distance.
- classifier.fit(X_train, y_train): Trains the K-Nearest Neighbors classifier on the training data (X_train and y_train).

➢ Predicting Test Set Results:

- y_pred = classifier.predict(X_test): Predicts the class labels for the test set using the trained classifier (classifier.predict).
- print(np.concatenate((y_pred.reshape(len(y_pred),1),y_test.reshape(len(y_test),1)),1)): Concatenates the predicted labels (y_pred) and the actual labels (y_test) vertically to compare them.

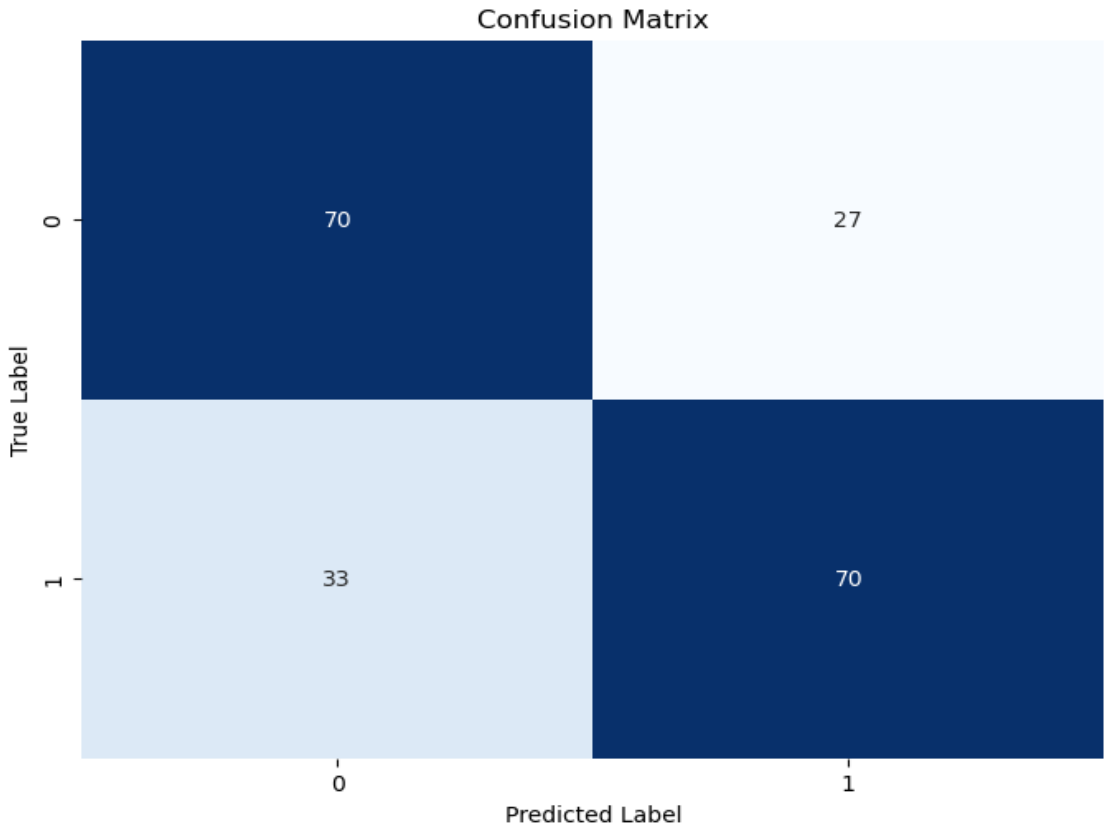
➔ **Output:** Total number of values: 200, Number of matching predicted and actual values: 140, Number of values that did not match: 60.

➔ **Interpretation:** In each row the first value represents the predicted and the second value represents the actual label. Similarly, the pattern goes on for all the rows. Most of our predicted values are matching with actual values. This shows that our model is performing well with less discrepancies.

➢ Confusion Matrix:

- from sklearn.metrics import confusion_matrix, accuracy_score: Imports the functions to calculate the confusion matrix and accuracy score.
- cm = confusion_matrix(y_test, y_pred): Computes the confusion matrix based on the actual and predicted labels.
- print(cm): Prints the confusion matrix.
- accuracy_score(y_test, y_pred): Computes the accuracy score, which is the proportion of correctly classified samples.

➔ **Output:** Accuracy - 0.7



➔ **Interpretation:** Looking at the above table we can infer that:

★ True Positives: The model has correctly predicted 70 values as positive (1) and they were actually positive (1).

★ False Positives: The model predicted 27 values as positive (1) and they were actually negative (0).

★ False Negative: The model predicted 33 values as negative (0) and they were actually positive (1).

★ True Negative: The model has correctly predicted 70 values as negative (0) and they were actually negative (0).

★ The accuracy is calculated by the formula **[(True Positive + True Negative)/Total number of values].**

= (70+70)/200*100

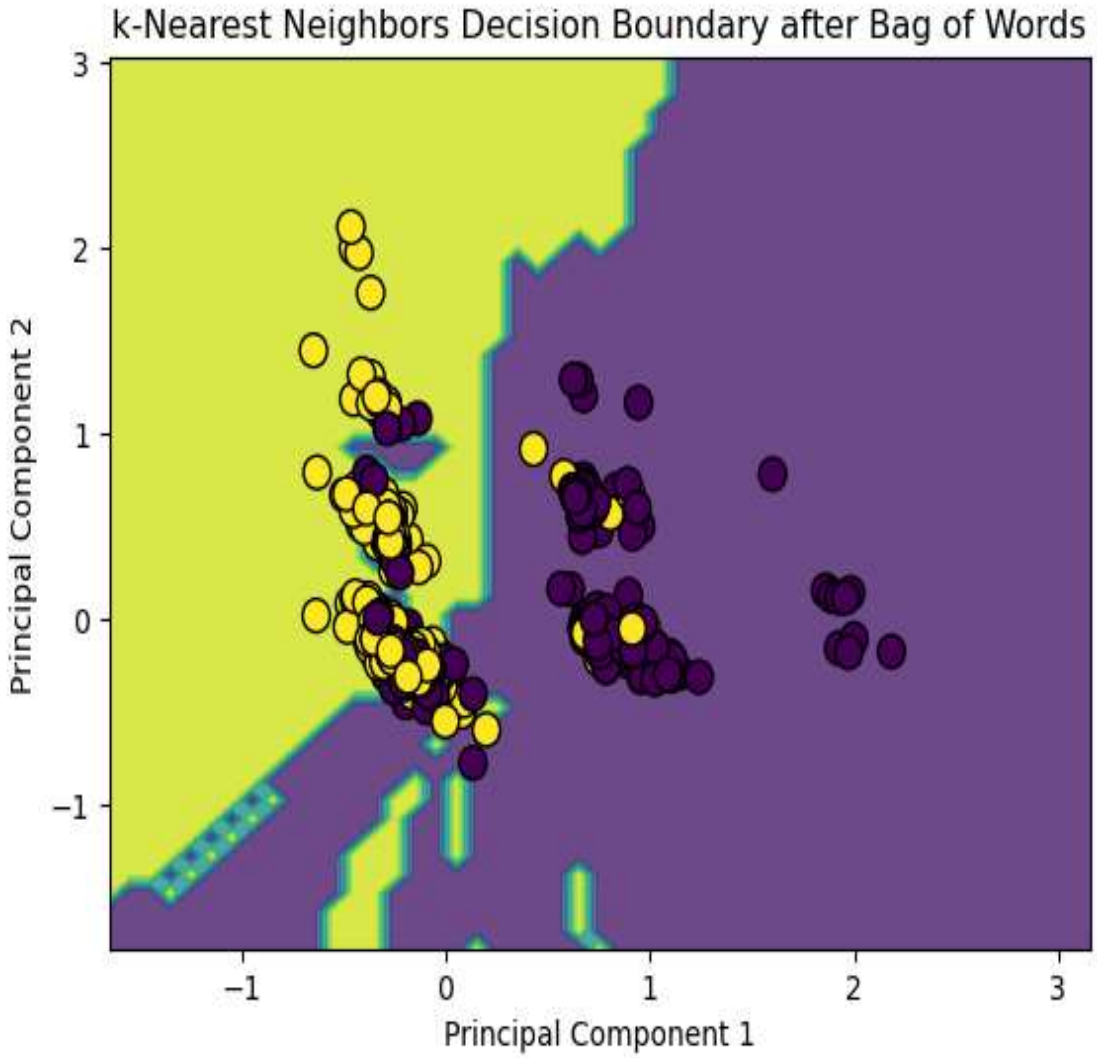
= 0.7*100

= 70.0%

It is coming out to be 70.0 %, which indicates that the model has correctly predicted 70.0% instances in the dataset. But the logistic regression predicts better than the K-nearest neighbor model with an accuracy of 80.5%.

➢ **Graphical Representation:**

➔ **Output:**



➔ **Interpretation:** We have reduced the data into 2 most explaining principal components and have plotted the graph in which the 'purple' areas are representing that the person liked the restaurant (1) and the 'yellow' area is representing that the person didn't like the restaurant (0). While the 'purple' balls predicted that the person liked the restaurant (1) while the 'yellow' balls predicted that the person who didn't liked the restaurant (0). We could see that most of our predictions are correct leaving a few of them which makes a good model. There is no linear line acting as a differentiator between the level (1) and (0) as we know k-nearest neighbor is a non-linear classifier.

4. STEPS INVOLVED IN LINEAR SUPPORT VECTOR MACHINE CLASSIFIER.

➢ Linear Support Vector Machine:

- SVM is a supervised learning algorithm used for classification, regression and outlier detection. It finds the best possible hyperplane that separates data points belonging to different classes. The hyperplane is chosen to maximize the margin, which is the distance between the hyperplane and the nearest data points from each class.
- Linear SVM is a variant of SVM that uses a linear kernel function to find a linear decision boundary that separates data points of different classes. It's efficient, effective for high-dimensional data, and easy to interpret, but it may not handle complex data relationships as well as non-linear SVMs.
- It specifically uses a linear kernel function. A kernel function is a mathematical function that takes two data points as input and computes a measure of similarity between them. In the case of a linear kernel, the similarity is based on the dot product of the feature vectors of the data points.

➢ Splitting the Dataset:

- train_test_split(X, y, test_size = 0.20, random_state = 0): Splits the dataset into training and testing sets. It takes input features X and labels y as input, with test_size = 0.20 indicating that 20% of the data will be used for testing, and random_state = 0 sets the random seed for reproducibility.

➢ Training the Support Vector Machine Classifier:

- from sklearn.svm import SVC: This line imports the Support Vector Classification (SVC) class from the SVM module in the scikit-learn library. SVC is the implementation of SVM for classification tasks.
- classifier = SVC (kernel = 'linear', random_state = 0): Here, we initialize the SVC classifiers.
- kernel = 'linear': This specifies the type of kernel to be used by the SVM algorithm. In this case, we're using a linear kernel, which means the SVM will try to find a linear separator (a straight line or plane) to separate the classes.

- random_state = 0: This parameter sets the random seed for reproducibility. When random_state is set to a specific number (here, 0), the results will be deterministic, meaning the same results will be produced every time the code is run. This helps with reproducibility of results.
- classifier.fit(X_train, y_train): Trains the K-Nearest Neighbors classifier on the training data (X_train and y_train).

➢ Predicting Test Set Results:

- y_pred = classifier.predict(X_test): Predicts the class labels for the test set using the trained classifier (classifier.predict).
- print(np.concatenate((y_pred.reshape(len(y_pred),1),y_test.reshape(len(y_test),1)),1)): Concatenates the predicted labels (y_pred) and the actual labels (y_test) vertically to compare them.

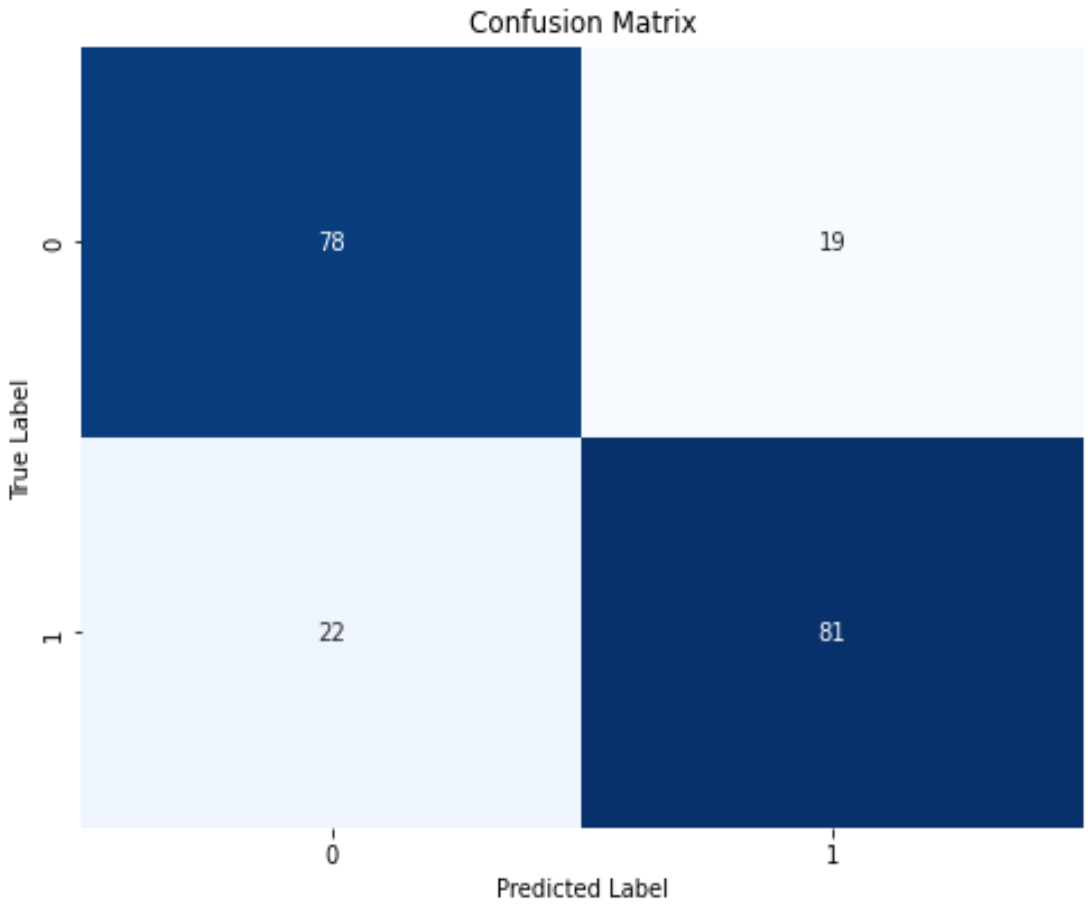
➔ **Output:** Total number of values: 200, Number of matching predicted and actual values: 159, Number of values that did not match: 41.

➔ **Interpretation:** In each row the first value represents the predicted and the second value represents the actual label. Similarly, the pattern goes on for all the rows. Most of our predicted values are matching with actual values. This shows that our model is performing well with less discrepancies.

➢ Confusion Matrix:

- from sklearn.metrics import confusion_matrix, accuracy_score: Imports the functions to calculate the confusion matrix and accuracy score.
- cm = confusion_matrix(y_test, y_pred): Computes the confusion matrix based on the actual and predicted labels.
- print(cm): Prints the confusion matrix.
- accuracy_score(y_test, y_pred): Computes the accuracy score, which is the proportion of correctly classified samples.

➔ **Output:** Accuracy - 0.795



➔ **Interpretation:** Looking at the above table we can infer that:

★ True Positives: The model has correctly predicted 78 values as positive (1) and they were actually positive (1).

★ False Positives: The model predicted 19 values as positive (1) and they were actually negative (0).

★ False Negative: The model predicted 22 values as negative (0) and they were actually positive (1).

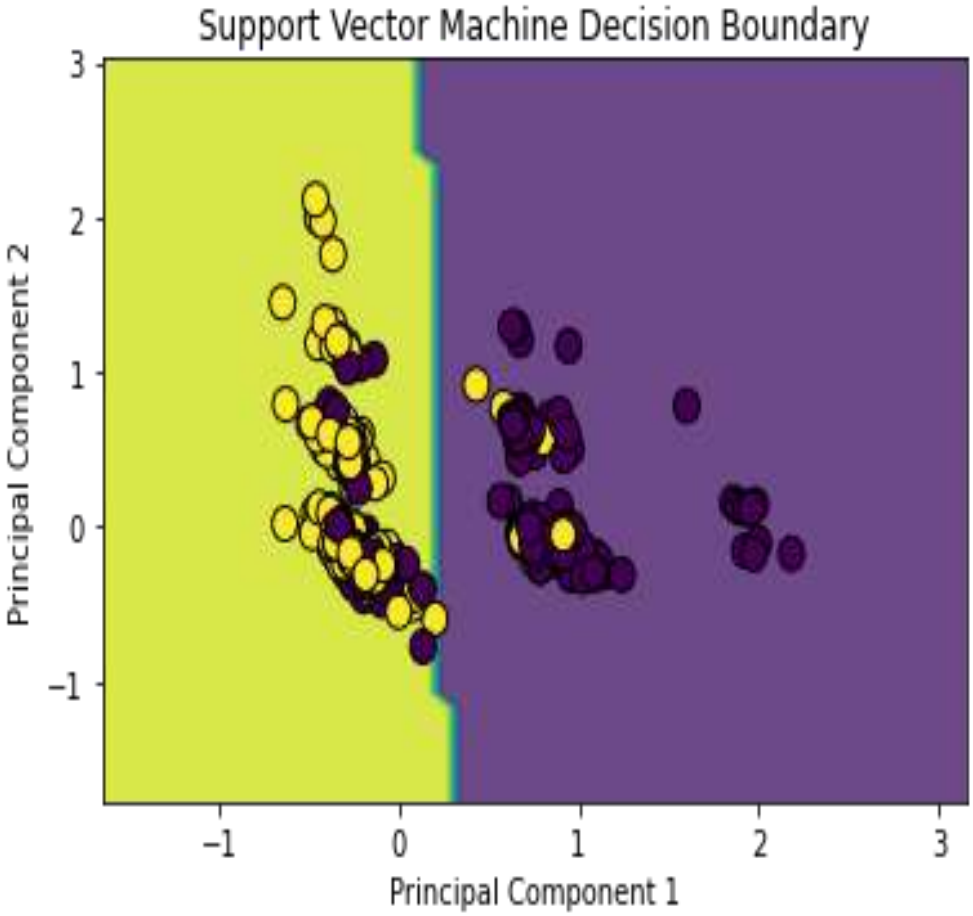
★ True Negative: The model has correctly predicted 81 values as negative (0) and they were actually negative (0).

★ The accuracy is calculated by the formula **[(True Positive + True Negative)/Total number of values].**
= (78+81)/200*100
= 0.795*100

= 79.5%

It is coming out to be 79.5 %, which indicates that the model has correctly predicted 79.50% instances in the dataset. The prediction of the SVM model is better than the K-nearest neighbor, which is just 70 %. But the logistic regression predicts better than the K-nearest neighbor and SVM model with an accuracy of 80.5%.

➢ **Graphical Representation:**



➔ **Interpretation:** We have reduced the data into 2 most explaining principal components and have plotted the graph in which the 'purple' area is representing that the person liked the restaurant (1) and the 'yellow' area is representing that the person didn't like the restaurant (0). While the 'purple' balls predicted that the person liked the restaurant (1) while the 'yellow' balls predicted that the person who didn't liked the restaurant (0). We could see that most of our predictions are correct leaving a few of them which makes a good

model. There is a straight line acting as a differentiator between the level (1) and (0) as we know that a linear Support Vector Machine is a linear classifier.

5. STEPS INVOLVED IN KERNEL SUPPORT VECTOR MACHINE CLASSIFIER.

➢ Kernel Support Vector Machine:

- Kernel Support Vector Machine (SVM) is a supervised machine learning algorithm used for classification and regression tasks. It is particularly effective for classification problems where the data is not linearly separable in the input space. Kernel SVM extends the traditional linear SVM by allowing for nonlinear decision boundaries through the use of kernel functions.

➢ Splitting the Dataset:

- train_test_split(X, y, test_size = 0.20, random_state = 0): Splits the dataset into training and testing sets. It takes input features X and labels y as input, with test_size = 0.20 indicating that 20% of the data will be used for testing, and random_state = 0 sets the random seed for reproducibility.

➢ Training the Kernel Support Vector Machine Classifier:

- from sklearn.svm import SVC: Imports the SVC class from scikit-learn, which is used to implement the Support Vector Classifier (SVC) algorithm.
- classifier = SVC (kernel = 'rbf', random_state = 0): Creates an instance of the SVC class with the following parameters:
- kernel = 'rbf': Specifies the kernel function to be used. In this case, it's the radial basis function (RBF) kernel, which is commonly used for its flexibility in capturing complex nonlinear relationships in the data.
- random_state = 0: Sets the random seed for reproducibility. It ensures that the results are consistent across different runs of the code.
- classifier.fit(X_train, y_train): Trains the Kernel SVM model on the training data (X_train and y_train). The fit () method fits the SVM model to the training data by finding the optimal decision boundary (hyperplane) that separates the classes in the feature space.

➢ Predicting Test Set Results:

- y_pred = classifier.predict(X_test): Predicts the class labels for the test set using the trained classifier (classifier.predict).
- print(np.concatenate((y_pred.reshape(len(y_pred),1),y_test.reshape(len(y_test),1)),1)): Concatenates the predicted labels (y_pred) and the actual labels (y_test) vertically to compare them.

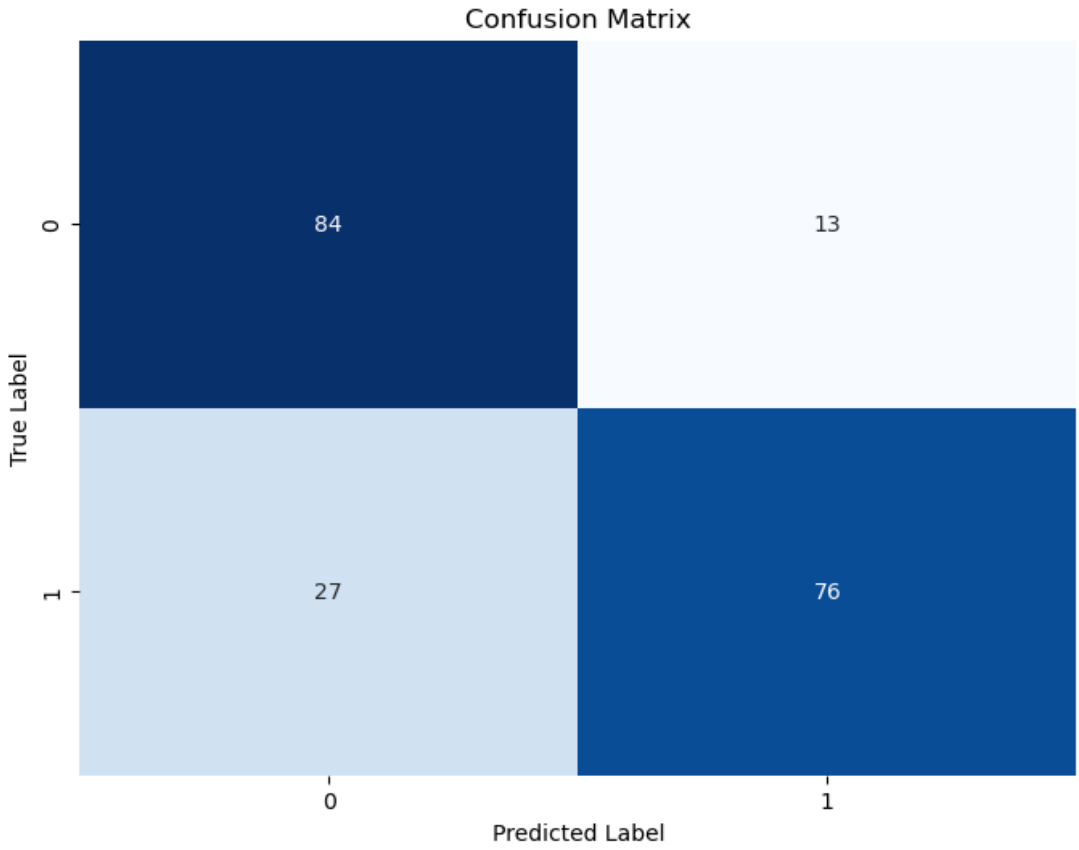
➔ **Output:** Total number of values: 200, Number of matching predicted and actual values: 160, Number of values that did not match: 40.

➔ **Interpretation:** In each row the first value represents the predicted and the second value represents the actual label. Similarly, the pattern goes on for all the rows. Most of our predicted values are matching with actual values. This shows that our model is performing well with less discrepancies.

➢ Confusion Matrix:

- from sklearn.metrics import confusion_matrix, accuracy_score: Imports the functions to calculate the confusion matrix and accuracy score.
- cm = confusion_matrix(y_test, y_pred): Computes the confusion matrix based on the actual and predicted labels.
- print(cm): Prints the confusion matrix.
- accuracy_score(y_test, y_pred): Computes the accuracy score, which is the proportion of correctly classified samples.

➔ **Output:** Accuracy - 0.8



➔ **Interpretation:** Looking at the above table we can infer that:

★ True Positives: The model has correctly predicted 84 values as positive (1) and they were actually positive (1).

★ False Positives: The model predicted 13 values as positive (1) and they were actually negative (0).

★ False Negative: The model predicted 27 values as negative (0) and they were actually positive (1).

★ True Negative: The model has correctly predicted 76 values as negative (0) and they were actually negative (0).

★ The accuracy is calculated by the formula **[(True Positive + True Negative)/Total number of values].**

= (84+76)/200*100

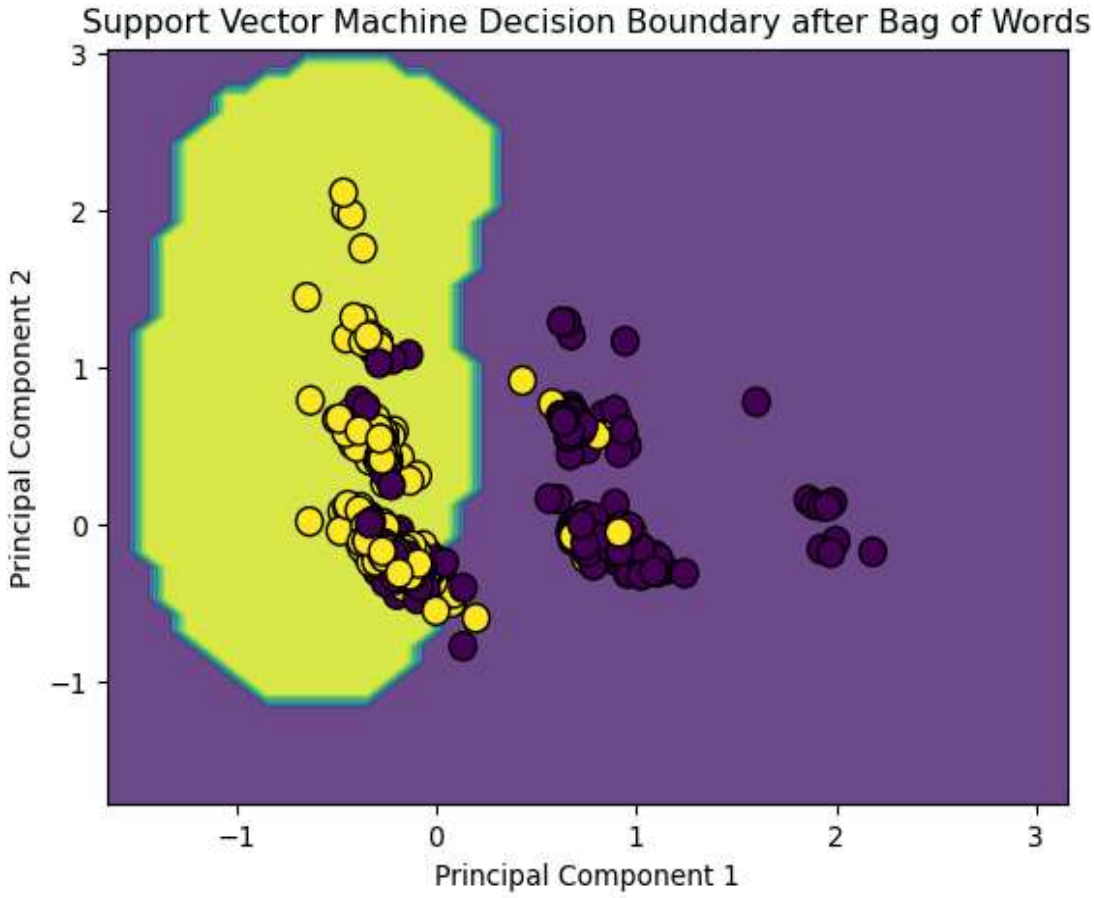
= 0.8*100

= 80.0%

It is coming out to be 80.0 %, which indicates that the model has correctly predicted 80.0% instances in the dataset. Which is slightly less than our accuracy of logistic regression.

➢ **Graphical Representation:**

➔ **Output:**



➔ **Interpretation:** We have reduced the data into 2 most explaining principal components and have plotted the graph in which the 'purple' areas are representing that the person liked the restaurant (1) and the 'yellow' area is representing that the person didn't like the restaurant (0). While the 'purple' balls predicted that the person liked the restaurant (1) while the 'yellow' balls predicted that the person who didn't liked the restaurant (0). We could see that most of our predictions are correct leaving a few of them which makes a good model. There is no linear line acting as a differentiator between the level (1) and (0) as we know that the kernel support vector machine is a non-linear

classifier. But we could see that there is very little difference in the accuracy when introducing a non-linear support vector machine.

6. STEPS INVOLVED IN DECISION TREE CLASSIFIER.

➢ Decision Tree Classifier

- A Decision Tree is a tree-like structure used in machine learning for making decisions. It's a popular algorithm used for both classification and regression tasks. It learns from data by asking the right questions to separate examples into different categories, ultimately helping to make predictions or decisions. The top node is called the "root node," and it represents the first decision or question.
- Each internal node represents a decision based on a feature (attribute), leading to more nodes or branches. To make a decision, you start at the root node and move down the tree, following the branches based on the answers to questions. The leaves of the tree represent the outcome or decision.

➢ Splitting the Dataset:

- train_test_split(X, y, test_size = 0.20, random_state = 0): Splits the dataset into training and testing sets. It takes input features X and labels y as input, with test_size = 0.20 indicating that 20% of the data will be used for testing, and random_state = 0 sets the random seed for reproducibility.

➢ Training the Decision Tree Classifier:

- from sklearn.tree import DecisionTreeClassifier: This line imports the DecisionTreeClassifier class from the tree module in the scikit-learn library.
- classifier: DecisionTreeClassifier(criterion = 'entropy', random_state = 0): Here, we initialize a DecisionTreeClassifier by its parameters.
- criterion: 'entropy': This specifies the criterion used for splitting the data at each node of the decision tree. 'entropy' is a measure of impurity or randomness in the data. The decision tree algorithm uses entropy to decide the best split that maximizes the information gain.

- random_state = 0: This parameter sets the random seed for reproducibility. When random_state is set to a specific number (here, 0), the results will be deterministic, meaning the same results will be produced every time the code is run. This helps with reproducibility of results.
- classifier.fit(X_train, y_train): classifier.fit(X_train, y_train): This line of code fits (trains) the random forest classifier using the training data (X_train and y_train).

➢ Predicting Test Set Results:

- y_pred = classifier.predict(X_test): Predicts the class labels for the test set using the trained classifier (classifier.predict).
- print(np.concatenate((y_pred.reshape(len(y_pred),1),y_test.reshape(len(y_test),1)),1)): Concatenates the predicted labels (y_pred) and the actual labels (y_test) vertically to compare them.

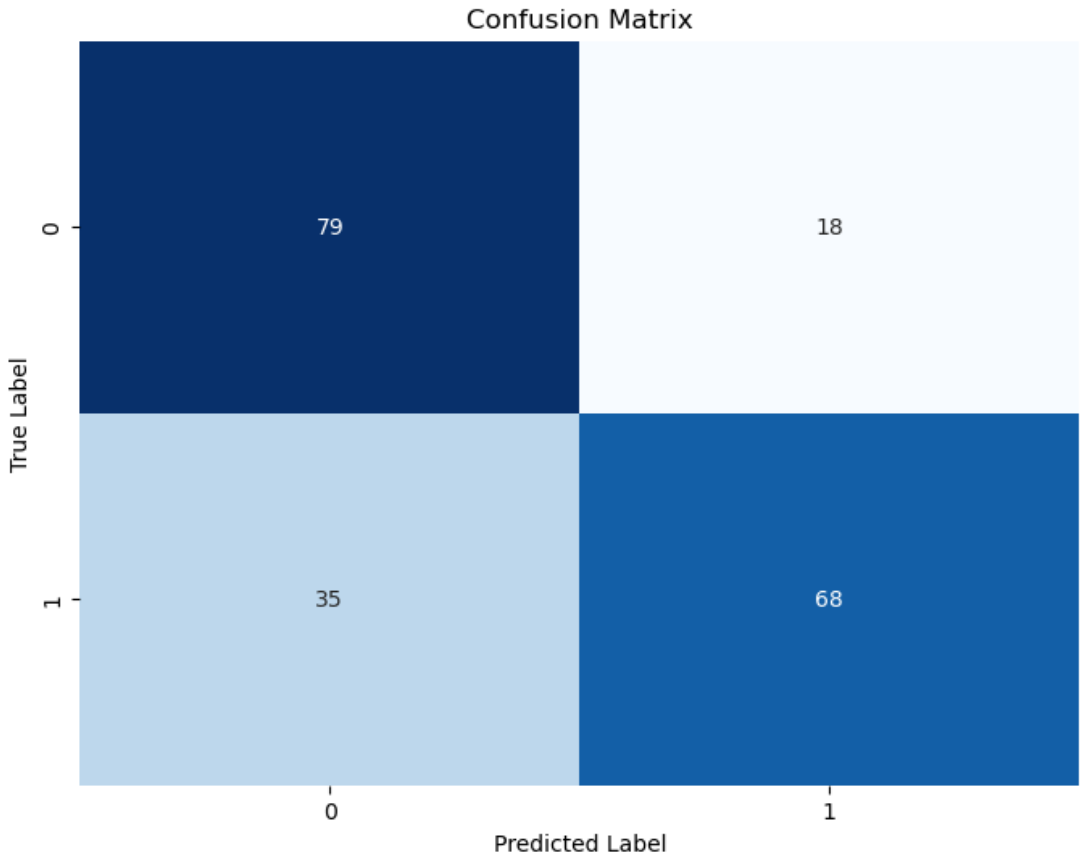
➔ **Output:** Total number of values: 200, Number of matching predicted and actual values: 147, Number of values that did not match: 53.

➔ **Interpretation:** In each row the first value represents the predicted and the second value represents the actual label. Similarly, the pattern goes on for all the rows. Most of our predicted values are matching with actual values. This shows that our model is performing well with less discrepancies.

➢ Confusion Matrix:

- from sklearn.metrics import confusion_matrix, accuracy_score: Imports the functions to calculate the confusion matrix and accuracy score.
- cm = confusion_matrix(y_test, y_pred): Computes the confusion matrix based on the actual and predicted labels.
- print(cm): Prints the confusion matrix.
- accuracy_score(y_test, y_pred): Computes the accuracy score, which is the proportion of correctly classified samples.

➔ **Output:** Accuracy - 0.735



➔ **Interpretation:** Looking at the above table we can infer that:

★ True Positives: The model has correctly predicted 79 values as positive (1) and they were actually positive (1).

★ False Positives: The model predicted 18 values as positive (1) and they were actually negative (0).

★ False Negative: The model predicted 35 values as negative (0) and they were actually positive (1).

★ True Negative: The model has correctly predicted 68 values as negative (0) and they were actually negative (0).

★ The accuracy is calculated by the formula **[(True Positive + True Negative)/Total number of values].**

= (79+68)/200*100

= 0.735*100

= 73.5%

It is coming out to be 73.5 %, which indicates that the model has correctly predicted 73.5 % instances in the dataset. But the prediction made by the logistic regression is the best with the highest accuracy 80.5%.

➢ **Graphical Representation:**

➔ **Output:**

➔ **Interpretation:** We have reduced the data into 2 most explaining principal components and have plotted the graph in which the 'purple' areas are representing that the person liked the restaurant (1) and the 'yellow' area is representing that the person didn't like the restaurant (0). While the 'purple' balls predicted that the person liked the restaurant (1) while the 'yellow' balls predicted that the person who didn't liked the restaurant (0). We could see that most of our predictions are correct leaving a few of them which makes a good model. There is no linear line acting as a differentiator between the level (1) and (0) , the Decision tree is a non-linear classifier.

7. STEPS INVOLVED IN RANDOM FOREST CLASSIFIER.

➢ Random Forest Classifier :

- Random Forests are versatile and can be used for both classification and regression tasks. It is like a team of decision trees working together to make predictions. By combining the votes of many trees and introducing randomness. Random Forests are robust, versatile, and effective at making accurate predictions for a wide range of tasks.
- By introducing randomness, each tree becomes a bit different from the others. This diversity is key to the strength of the Random Forest because it helps prevent overfitting (making predictions too specific to the training data) and improves generalization to new, unseen data. When it's time to make a prediction, each tree in the forest gets a vote, and the most popular prediction wins.

➢ Splitting the Dataset:

- train_test_split(X, y, test_size = 0.20, random_state = 0): Splits the dataset into training and testing sets. It takes input features X and labels y as input, with test_size = 0.20 indicating that 20% of the data will be used for testing, and random_state = 0 sets the random seed for reproducibility.

➢ Training the Decision Tree Classifier:

- from sklearn.ensemble import RandomForestClassifier: This line imports the RandomForestClassifier class from the ensemble module in the scikit-learn library.
- classifier = RandomForestClassifier(n_estimators = 10, criterion = 'entropy', random_state = 0): Here, we initialize a RandomForestClassifier object named classifier with the following parameters:
- n_estimators = 10: This parameter specifies the number of decision trees in the random forest. In this case, we're using 10 decision trees.
- criterion = 'entropy': This specifies the criterion used for splitting the data at each node of each decision tree in the forest. 'entropy' is a measure of

impurity or randomness in the data. The decision trees in the random forest use entropy to decide the best split that maximizes the information gain.

- random_state = 0: This parameter sets the random seed for reproducibility. When random_state is set to a specific number (here, 0), the results will be deterministic, meaning the same results will be produced every time the code is run. This helps with reproducibility of results.
- classifier.fit(X_train, y_train): classifier.fit(X_train, y_train): This line of code fits (trains) the random forest classifier using the training data (X_train and y_train).

➢ Predicting Test Set Results:

- y_pred = classifier.predict(X_test): Predicts the class labels for the test set using the trained classifier (classifier.predict).
- print(np.concatenate((y_pred.reshape(len(y_pred),1),y_test.reshape(len(y_test),1)),1)): Concatenates the predicted labels (y_pred) and the actual labels (y_test) vertically to compare them.

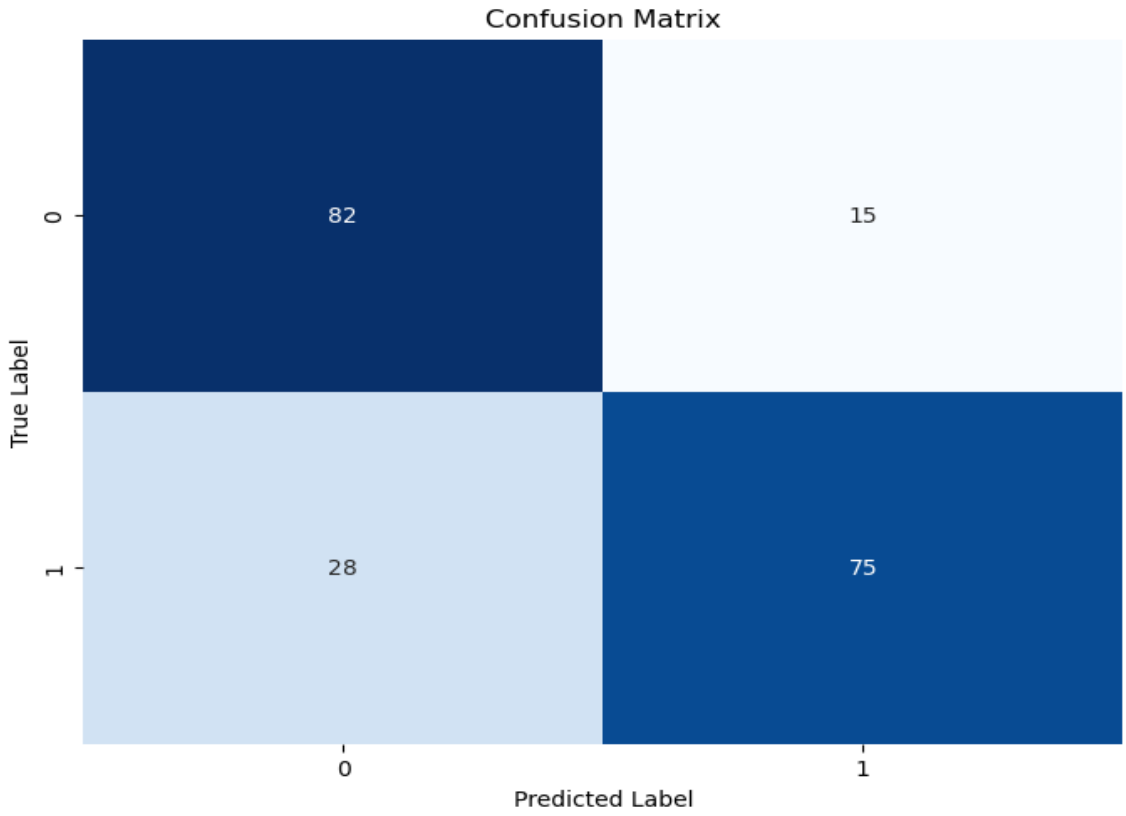
➔ **Output:** Total number of values: 200, Number of matching predicted and actual values: 157, Number of values that did not match: 43.

➔ **Interpretation:** In each row the first value represents the predicted and the second value represents the actual label. Similarly, the pattern goes on for all the rows. Most of our predicted values are matching with actual values. This shows that our model is performing well with less discrepancies.

➢ Confusion Matrix:

- from sklearn.metrics import confusion_matrix, accuracy_score: Imports the functions to calculate the confusion matrix and accuracy score.
- cm = confusion_matrix(y_test, y_pred): Computes the confusion matrix based on the actual and predicted labels.
- print(cm): Prints the confusion matrix.
- accuracy_score(y_test, y_pred): Computes the accuracy score, which is the proportion of correctly classified samples.

➔ **Output:** Accuracy - 0.785



➔ **Interpretation:** Looking at the above table we can infer that:

★ True Positives: The model has correctly predicted 82 values as positive (1) and they were actually positive (1).

★ False Positives: The model predicted 15 values as positive (1) and they were actually negative (0).

★ False Negative: The model predicted 28 values as negative (0) and they were actually positive (1).

★ True Negative: The model has correctly predicted 75 values as negative (0) and they were actually negative (0).

★ The accuracy is calculated by the formula **[(True Positive + True Negative)/Total number of values].**

= (82+75)/200*100

= 0.785*100

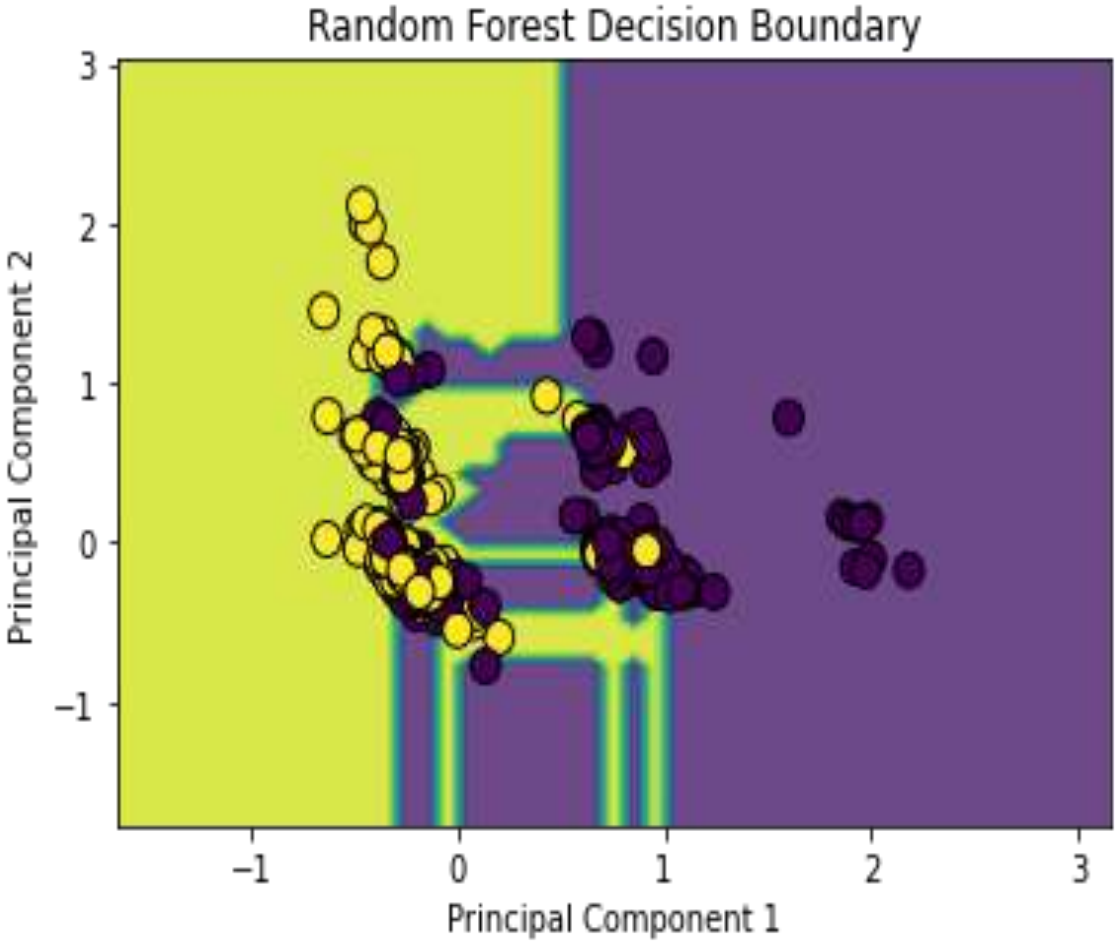
= 78.5%

It is coming out to be 78.5 %, which indicates that the model has correctly predicted 78.5 % instances in the dataset. The accuracy is much better than the decision tree

(73.5 %). But the prediction made by the logistic regression is still the best with the highest accuracy 80.5%.

➢ **Graphical Representation:**

➔ **Output:**



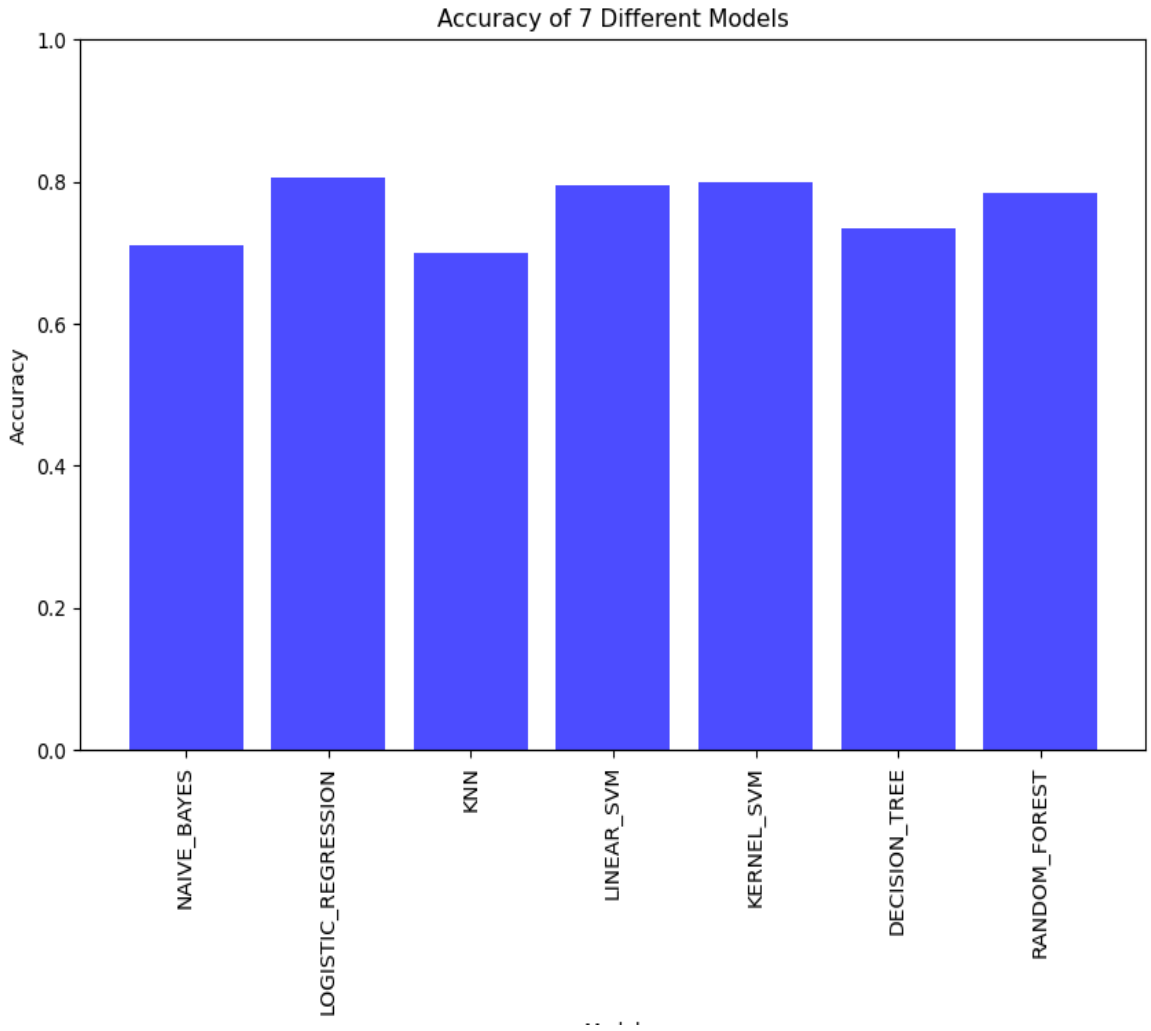
➔ **Interpretation:** We have reduced the data into 2 most explaining principal components and have plotted the graph in which the 'purple' areas are representing that the person liked the restaurant (1) and the 'yellow' area is representing that the person didn't like the restaurant (0). While the 'purple' balls predicted that the person liked the restaurant (1) while the 'yellow' balls predicted that the person who didn't liked the restaurant (0). We could see that most of our predictions are correct leaving a few of them which makes a good model. There is no linear line acting as a differentiator between the level (1) and (0) , the Random Forest is a non-linear classifier.

Conclusion and Discussion

➢ **We fitted the seven different model which are as follows:**

- Naive Bayes
- Logistic Regression
- K-Nearest Neighbors
- Linear Support Vector Machine
- Kernel Support Vector Machine
- Decision Trees
- Random Forest

➔ **Output:**



➔ **Interpretation:** We found that the logistic regression model has the maximum accuracy of 80.5% followed by the kernel support vector machine having an

accuracy of 80%. The K-nearest neighbor model didn't fit well compared to the rest of the model having accuracy of just 70%.

➔ **Discussion:** Logistic regression gave the best accuracy so we want to create a new training set and test set so as to find if our model is really good or it just performed well in that training and test set.

➢ Summary of Logistic Regression

➔ **Output:** Coefficients: [[0. -0.027878 0.18472357 ... 0. 0.29843889 -0.68406941]], Intercept: [0.00428888]

➔ **Interpretation:**

- ★ Each coefficient represents the change in the log-odds of the dependent variable for a one-unit change in the corresponding independent variable, holding other variables constant.
- ★ Positive coefficients indicate that as the corresponding independent variable increases, the log-odds of the dependent variable also increase.
- ★ Negative coefficients indicate that as the corresponding independent variable increases, the log-odds of the dependent variable decrease.
- ★ The intercept represents the log-odds of the dependent variable when all independent variables are zero.

➢ K-Fold Cross Validation:

- k-Fold Cross Validation is a technique used to evaluate the performance of a machine learning model by dividing the dataset into k subsets (or folds) of equal size. The evaluation process is then performed k times, with each subset being used once as the testing data while the remaining k-1 subsets are used as training data.

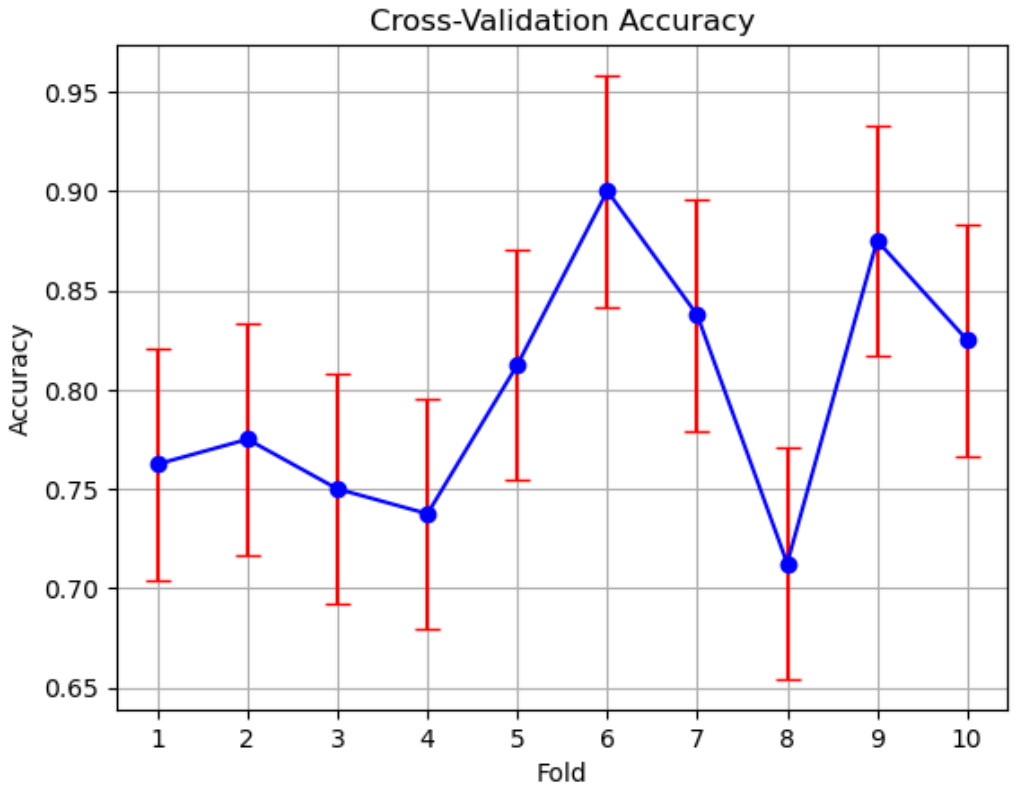
➢ Performing K-Fold Cross Validation:

- Imports the cross_val_score function from sklearn.model_selection, which is used for performing k-Fold Cross Validation.
- Cross Validation:
- The cross_val_score function is then used to evaluate the performance of the classifier model (classifier) using k-Fold Cross Validation.
- X: The feature matrix.

- y: The target variable.
- cv: The number of folds in k-Fold Cross Validation. Here it is 10-fold.
- accuracies.mean(): Calculates the mean of accuracies obtained from k-Fold Cross Validation.
- accuracies.std(): Calculates the standard deviation of accuracies obtained from k-Fold Cross Validation.

➔ **Output:** Accuracy: 79.88 %, Standard Deviation: 5.82 %.

➔ **Graph:**



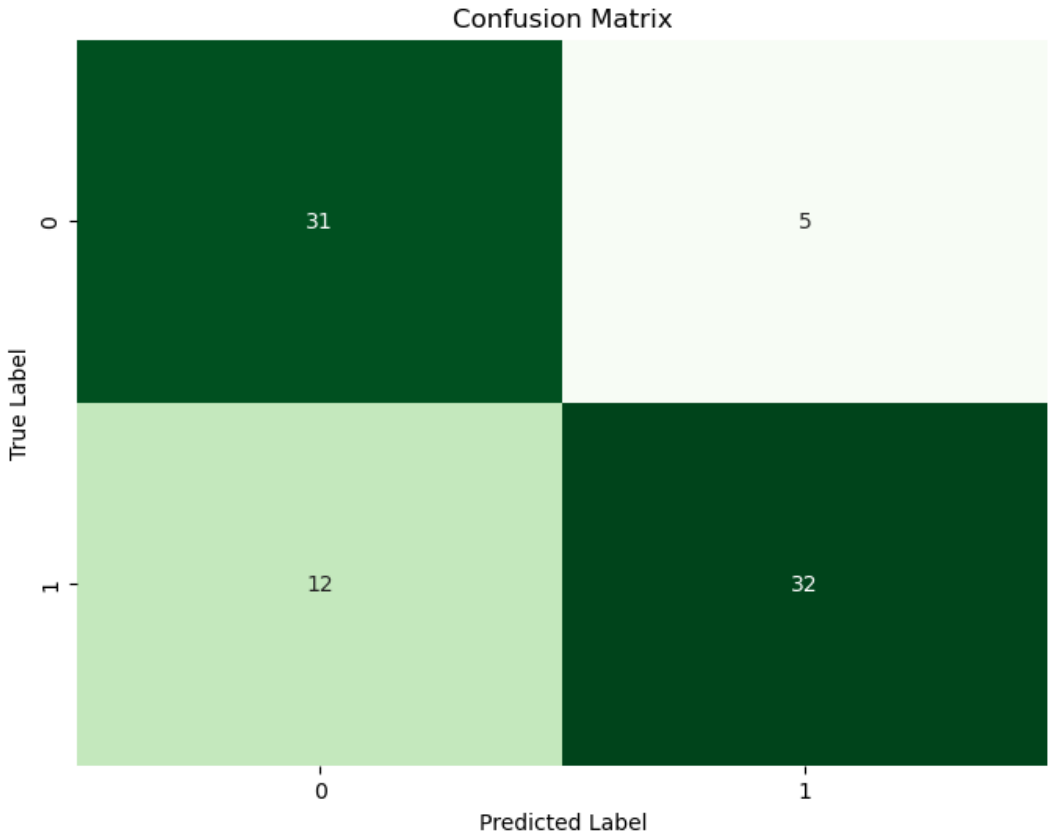
➔ **Interpretation:** We have found out that the average accuracy of the 10 folds is nearly 80% which is very similar to that of the accuracy obtained from a single fold in logistic regression as well as we can see that the accuracies deviate by a small margin of 5.82% which shows that our model performs equal for all the cases. We can see from the graph the different accuracies at each step of the 10 folds.

➔ **Discussion:** It was clear that the logistic regression performs the same in all the scenarios. To get more confirmation we divided the whole training set into two randomly equal parts and divided each part into 80% training and 20% testing further performing K-fold cross validation on each one of them again.

➢ Logistic Regression on First Part:

- We find out the accuracy and confusion matrix on the first part

➔ **Graph:**



➔ **Interpretation:** Looking at the above table we can infer that:

★ True Positives: The model has correctly predicted 31 values as positive (1) and they were actually positive (1).

★ False Positives: The model predicted 5 values as positive (1) and they were actually negative (0).

★ False Negative: The model predicted 12 values as negative (0) and they were actually positive (1).

★ True Negative: The model has correctly predicted 32 values as negative (0) and they were actually negative (0).

★ The accuracy is calculated by the formula **[(True Positive + True Negative)/Total number of values].**
= (31+32)/80*100

= 0.7875*100

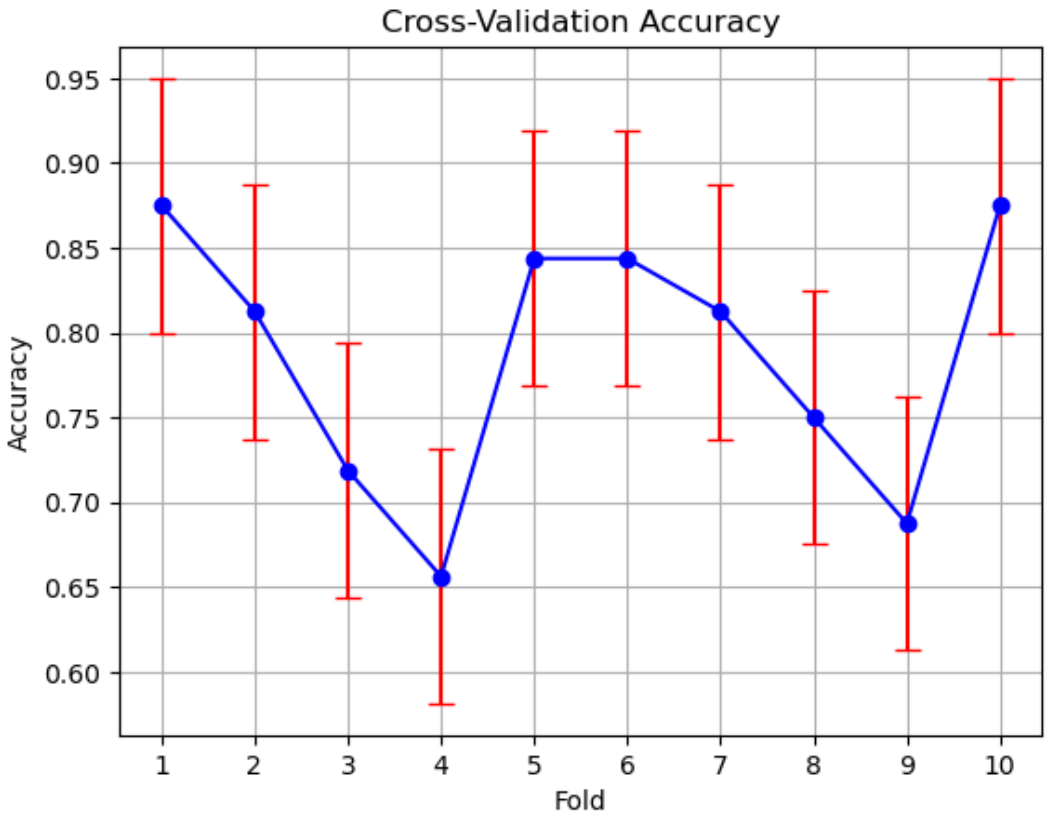
= 78.75%

It is coming out to be 78.75 %, which indicates that the model has correctly predicted 78.75 % instances in the dataset.

➢ Performing K-fold Cross Validation on the First Part:

➔ **Output:** Accuracy: 78.75 %, Standard Deviation: 7.50 %

➔ **Graph:**

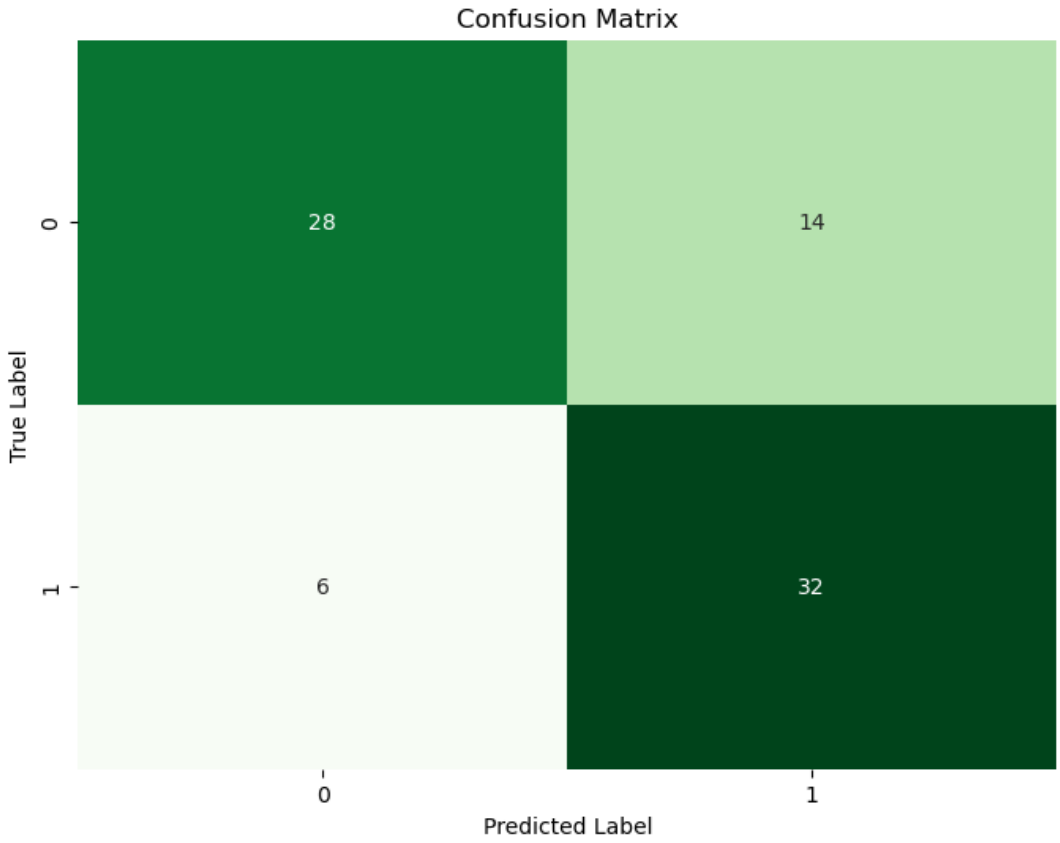


➔ **Interpretation:** We have found out that the average accuracy of the 10 folds is 78.75% which is similar to that of the accuracy obtained from a single fold in the first part as well as we can see that the accuracies deviate by a small margin of 7.50% which shows that our model performs equal for all the cases. We can see from the graph the different accuracies at each step of the 10 folds.

➢ Logistic Regression on Second Part:

- We find out the accuracy and confusion matrix on the first part

➔ **Graph:**



➔ **Interpretation:** Looking at the above table we can infer that:

★ True Positives: The model has correctly predicted 28 values as positive (1) and they were actually positive (1).

★ False Positives: The model predicted 14 values as positive (1) and they were actually negative (0).

★ False Negative: The model predicted 6 values as negative (0) and they were actually positive (1).

★ True Negative: The model has correctly predicted 32 values as negative (0) and they were actually negative (0).

★ The accuracy is calculated by the formula **[(True Positive + True Negative)/Total number of values].**

= (28+32)/80*100

= 0.75*100

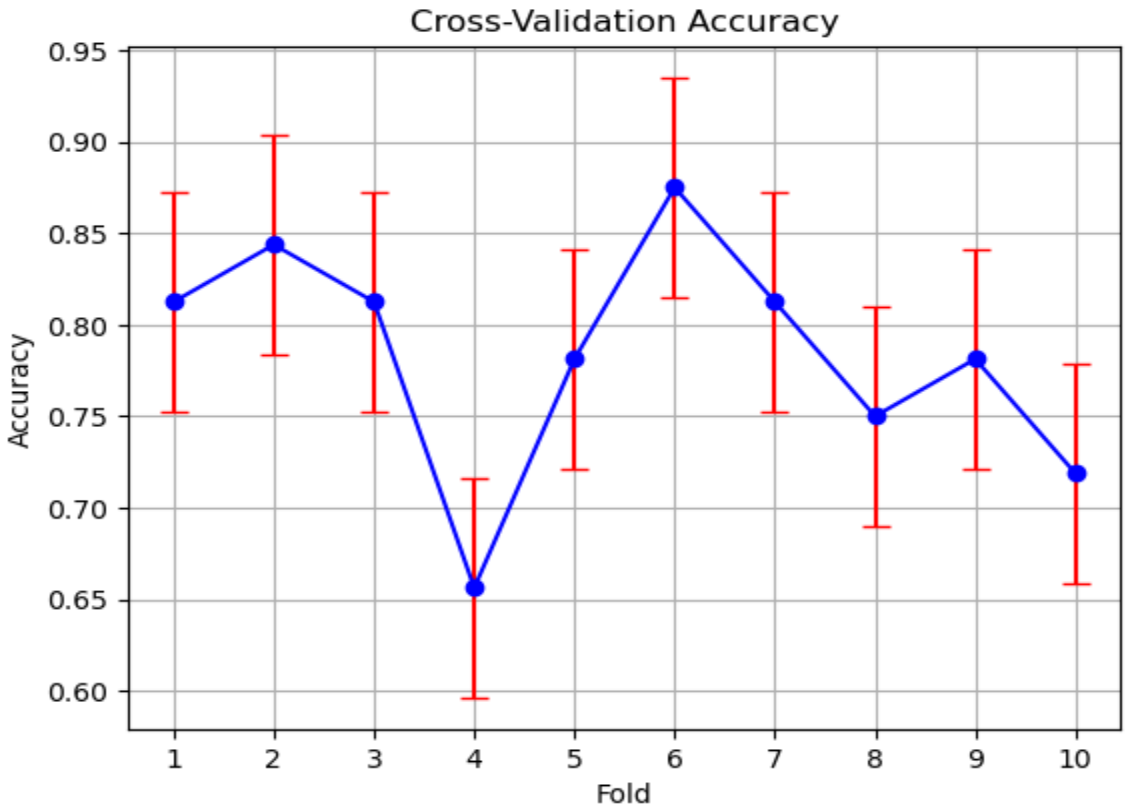
= 75.0%

It is coming out to be 75.0 %, which indicates that the model has correctly predicted 75.0 % instances in the dataset.

➢ Performing K-fold Cross Validation on the First Part:

➔ **Output:** Accuracy: 78.44 %, Standard Deviation: 6.00 %

➔ **Graph:**



➔ **Interpretation:** We have found out that the average accuracy of the 10 folds is 78.44% which is better than that of the accuracy obtained from a single fold in the second part as well as we can see that the accuracies deviate by a small margin of 6.00% which shows that our model performs equal for all the cases. We can see from the graph the different accuracies at each step of the 10 folds.

References

- https://www.w3schools.com/
- https://www.kaggle.com/
- https://scikit-learn.org/stable/
- https://github.com/
- https://en.wikipedia.org/wiki/Natural_language_processing