

PROGRAM 1

AIM: To implement the BFS algorithm for graph traversal

THEORY:

Breadth-First Search (BFS) is a graph traversal algorithm that explores nodes level by level using a queue (FIFO), ensuring the shortest path in an unweighted graph. It starts from a node, visits all its neighbors, then moves to the next level. With a time complexity of $O(V + E)$, BFS is widely used in shortest path finding, web crawling, social networks, and cycle detection.

ALGORITHM:

1. Initialize an empty queue and enqueue the starting node.
2. Mark the starting node as visited.
3. While the queue is not empty, do the following:
 - Dequeue a node and process it.
 - For each unvisited neighbor of the current node:
 - (I) Mark it as visited.
 - (II) Enqueue it into the queue.
4. Repeat until all reachable nodes are visited.

SOURCE CODE:

```
from collections import deque

# BFS function
def bfs(graph, start_node):
    visited = set()
    queue = deque([start_node])

    print("BFS Traversal:")

    while queue:
        node = queue.popleft()
        if node not in visited:
            print(node, end=' ')
            visited.add(node)
            queue.extend(neighbor for neighbor in graph[node] if neighbor not in visited)

# Sample graph represented as an adjacency list
```

```
graph = {  
    'A': ['B', 'C'],  
    'B': ['D', 'E'],  
    'C': ['F'],  
    'D': [],  
    'E': ['F'],  
    'F': []  
}
```

```
# Call BFS  
bfs(graph, 'A')
```

OUTPUT:

```
Output  
BFS Traversal:  
A B C D E F  
=== Code Execution Successful ===
```

OBSERVATION:

The BFS (Breadth-First Search) algorithm explores a graph level by level, starting from the root node and visiting all neighboring nodes before moving to the next level. This ensures that the shortest path (in terms of number of edges) from the starting node to any other reachable node is found. In this implementation, a queue is used to manage the order of traversal, and a set keeps track of visited nodes to avoid repetition. The output confirms that BFS visits nodes in a systematic and predictable manner, making it suitable for scenarios like shortest path finding in unweighted graphs and web crawling. This experiment reinforces the importance of using appropriate data structures like queues and sets in implementing graph traversal algorithms efficiently.

PROGRAM-2

AIM: To implement the Water Jug problem in AI.

THEORY:

The Water Jug Problem is a classical problem in Artificial Intelligence that illustrates how agents can solve problems using state space search techniques. The problem typically involves two jugs with different capacities and the objective is to obtain a specific amount of water using these jugs. The challenge lies in reaching the desired goal state using a series of allowable operations while tracking the state of both jugs at each step.

Each state in the problem is represented as a pair (x, y) , where x denotes the amount of water in Jug1 and y represents the amount in Jug2. The initial state is generally $(0, 0)$, meaning both jugs are empty. The goal state is any configuration where the desired quantity of water is present in either of the jugs. The operations allowed to transition between states include filling either jug completely, emptying either jug, and transferring water from one jug to the other until one is empty or the other is full.

The Water Jug Problem serves as an excellent example of how problems can be formulated and solved using AI techniques. It helps learners understand the basics of problem representation, state transitions, and the implementation of search algorithms such as Breadth-First Search (BFS), Depth-First Search (DFS), and Backtracking. This problem is also foundational in understanding broader AI concepts like constraint satisfaction and planning in a defined environment.

ALGORITHM:

1. Start with both jugs empty, representing the initial state as $(0, 0)$
2. Create a queue to store states for Breadth-First Search (BFS) and a set to keep track of visited states
3. While the queue is not empty, remove the front element to process the current state
4. If the current state matches the goal (i.e., one of the jugs contains the required amount of water), terminate and return the solution path
5. From the current state, generate all possible valid states using the allowed operations: fill any jug, empty any jug, or pour water from one jug to the other
6. For each new state, if it has not been visited, add it to the queue and mark it as visited
7. Repeat the process until the goal is found or all possible states have been explored
8. If no solution is found after exploring all states, report that the problem has no solution under the given constraints

SOURCE CODE:

```
from collections import deque
import math

def water_jug_bfs(x, y, z):
    if z > max(x, y) or z % math.gcd(x, y) != 0:
        return "No Solution"

    visited = set()
    queue = deque([(0, 0)])
    steps = []

    while queue:
        a, b = queue.popleft()
        steps.append((a, b))

        if a == z or b == z:
            print("Steps to reach the goal:")
            for i, step in enumerate(steps):
                print(f"Step {i+1}: {step}")
            return "Solution Found!"

        if (a, b) in visited:
            continue

        visited.add((a, b))

        next_states = [
            (x, b),
            (a, y),
            (0, b),
            (a, 0),
            (a - min(a, y - b), b + min(a, y - b)),
            (a + min(b, x - a), b - min(b, x - a))
        ]

        for state in next_states:
            if state not in visited:
                queue.append(state)
```

```
return "No Solution"
```

```
x = int(input("Enter capacity of Jug 1: "))  
y = int(input("Enter capacity of Jug 2: "))  
z = int(input("Enter the target amount: "))  
print(water_jug_bfs(x, y, z))
```

OUTPUT:

```
Output  
Enter capacity of Jug 1: 5  
Enter capacity of Jug 2: 7  
Enter the target amount: 2  
Steps to reach the goal:  
Step 1: (0, 0)  
Step 2: (5, 0)  
Step 3: (0, 7)  
Step 4: (5, 7)  
Step 5: (0, 5)  
Step 6: (5, 7)  
Step 7: (5, 2)  
Solution Found!
```

OBSERVATION:

We implemented the Breadth-First Search (BFS) algorithm to solve the classical Water Jug Problem. By representing the states of the jugs as nodes and transitions as edges, we explored all possible valid states systematically. The BFS approach helped ensure the shortest path to the goal was found, if a solution existed. We also learned how mathematical conditions such as the GCD of jug capacities play a key role in determining the feasibility of the solution. Through this, we gained practical experience in applying search algorithms to real-world problems using Python, and understood the importance of maintaining visited states to avoid infinite loops.

PROGRAM 3

AIM: To create and load a dataset in the program.

THEORY: In Python, loading datasets involves reading data from various file formats like CSV, Excel, JSON, or SQL databases into memory for analysis. The most common library for this task is pandas, which provides functions like `read_csv()`, `read_excel()`, and `read_sql()` to load data into a structured format, usually a DataFrame. This allows for easy exploration, cleaning, and manipulation of data. Other libraries like NumPy are used for numerical data in arrays. Once loaded, the dataset can be inspected, cleaned, and transformed, making it ready for further analysis or machine learning tasks.

ALGORITHM:

1. Import Required Libraries
 - Import the necessary Python libraries: pandas for data manipulation and os for file handling.
2. Load Dataset Based on File Type
 - Based on the file extension, use the appropriate pandas function:
 - `read_csv()` for CSV files
 - `read_json()` for JSON files.
3. Inspect the Loaded Dataset
 - Print the first few rows of the dataset to verify the data is loaded correctly.
4. Handle Errors
 - If there are any issues loading the file (e.g., unsupported format, missing values), raise an appropriate error.
5. Return the Loaded Dataset
 - Return the loaded dataset for further use.

SOURCE CODE:

```
import math

import numpy as np

import pandas as pd

import seaborn as sns

sns.set_style('whitegrid')

import matplotlib.pyplot as plt

plt.style.use("fivethirtyeight")


import keras

from keras.models import Sequential

from keras.callbacks import EarlyStopping

from keras.layers import Dense, LSTM, Dropout


from sklearn.preprocessing import MinMaxScaler

from sklearn.model_selection import train_test_split

from sklearn.metrics import mean_squared_error, mean_absolute_error

data_dir = '/content/sample_data/TSLA.csv'

df = pd.read_csv(data_dir, parse_dates= True , index_col= "Date")

df.head()
```

OUTPUT:

	Open	High	Low	Close	Adj Close	Volume
Date						
2010-06-29	3.800	5.000	3.508	4.778	4.778	93831500
2010-06-30	5.158	6.084	4.660	4.766	4.766	85935500
2010-07-01	5.000	5.184	4.054	4.392	4.392	41094000
2010-07-02	4.600	4.620	3.742	3.840	3.840	25699000
2010-07-06	4.000	4.000	3.166	3.222	3.222	34334500

OBSERVATIONS:

Key Observations:

1. Pandas simplifies loading datasets into structured formats like DataFrames.
2. Supports various formats like CSV, Excel, and JSON.
3. The process involves checking file existence, selecting the appropriate function, and loading data.
4. Inspect data with `.head()` to verify contents.
5. Proper error handling ensures smooth loading and handling of missing values.
6. Python provides an efficient way to load and analyze datasets from different formats.

PROGRAM 4

AIM: To perform data preparation by checking for null values, analyzing the shape of the dataset, and retrieving dataset information.

THEORY:

Data preparation is a crucial step in machine learning and data analysis, ensuring that the dataset is clean and ready for further processing. It involves handling missing values, identifying inconsistencies, and understanding the structure of the data. The key operations include:

1. **Checking Null Values:** Identifies missing data, which can be handled using imputation or removal techniques.
2. **Finding Sum of Null Values:** Determines how many missing values exist in each column to assess data quality.
3. **Checking Dataset Shape:** Retrieves the number of rows and columns, providing insight into dataset size.
4. **Checking Dataset Information:** Displays column data types, non-null counts, and memory usage, which helps in selecting appropriate preprocessing methods.

SOURCE CODE:

```
print("\nDataset Info:")
df.info()
print("\nMissing Values:\n", df.isnull().sum())
duplicate_rows = df.duplicated().sum()
print("Number of duplicate rows:", duplicate_rows)
df.drop_duplicates(inplace=True)
print("\nDataset shape after removing duplicates:", df.shape)
```

OUTPUT:



```
Dataset Info:
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 2956 entries, 2010-06-29 to 2022-03-24
Data columns (total 6 columns):
#   Column      Non-Null Count  Dtype
---  -
0   Open        2956 non-null   float64
1   High        2956 non-null   float64
2   Low         2956 non-null   float64
3   Close       2956 non-null   float64
4   Adj Close   2956 non-null   float64
5   Volume      2956 non-null   int64
dtypes: float64(5), int64(1)
memory usage: 161.7 KB
```



```
Missing Values:
Open      0
High      0
Low       0
Close     0
Adj Close 0
Volume    0
dtype: int64
Number of duplicate rows: 0

Dataset shape after removing duplicates: (2956, 6)
```

OBSERVATION/LESSON LEARNT

- No missing values were identified, which need further handling (e.g., imputation or removal).
- Verified dataset shape to confirm expected dimensions.
- Retrieved dataset information, including data types and memory usage, essential for preprocessing steps.
- Ensures that the dataset is clean and well-structured for machine learning or analysis.

PROGRAM 5

AIM: To perform Data Pre-processing of the dataset.

THEORY:

Data pre-processing is a crucial step in machine learning that involves transforming raw data into a structured format suitable for analysis. It ensures that the dataset is clean, consistent, and ready for model training. In this process, various features are extracted and computed, categorized into Length Features, Count Features, and Binary Features.

ALGORITHM:

1. Extract the "Close" price column from the dataset.
2. Convert the selected column to a DataFrame format.
3. Convert the DataFrame to a NumPy array to perform numerical operations.
4. Optionally, check the shape of the dataset to verify its dimensions.
5. Normalize the data using MinMaxScaler to scale all values between 0 and 1. This helps in stabilizing and speeding up the training process.
6. Split the dataset into training and testing sets:
 - Allocate 75% of the data to training.
 - Allocate the remaining 25% to testing.
 - Include the last 60 values of the training data as a prefix to the test data to maintain continuity in the time series.
7. Initialize two empty lists to hold training inputs (x_train) and corresponding outputs (y_train).
8. Create sequences of 60 time steps from the training data:
 - Loop from index 60 to the end of the training data.
 - For each index, extract the previous 60 values as input.
 - Use the current value as the output.
9. Convert the x_train and y_train lists into NumPy arrays.
10. Reshape the input data (x_train) into 3-dimensional format (samples, time steps, features) to be compatible with LSTM model requirements.

SOURCE CODE:

```
dataset = df["Close"]
dataset = pd.DataFrame(dataset)

data = dataset.values

data.shape

#Normalizing Data
from sklearn.preprocessing import MinMaxScaler
scaler = MinMaxScaler(feature_range= (0, 1))
scaled_data = scaler.fit_transform(np.array(data).reshape(-1, 1))

#Splitting the Data
train_size = int(len(data)*.75)
test_size = len(data) - train_size

print("Train Size :",train_size,"Test Size :",test_size)

train_data = scaled_data[ :train_size , 0:1 ]
test_data = scaled_data[ train_size-60: , 0:1 ]

#Creating training set
x_train = []
y_train = []


for i in range(60, len(train_data)):
    x_train.append(train_data[i-60:i, 0])
```

```
y_train.append(train_data[i, 0])

# Convert to numpy array
x_train, y_train = np.array(x_train), np.array(y_train)

# Reshaping the input
x_train = np.reshape(x_train, (x_train.shape[0], x_train.shape[1], 1))
x_train.shape , y_train.shape
```

OUTPUT:



```
Train Size : 2217 Test Size : 739
((2157, 60, 1), (2157,))
```

OBSERVATION/LESSON LEARNED:

1. Data pre-processing ensures the dataset is clean and structured.
2. Feature extraction helps in identifying patterns useful for classification.
3. The resultant dataset contains informative attributes crucial for URL classification models.

PROGRAM 6

AIM: To perform Exploratory Data Analysis (EDA) on the given dataset to understand the distribution, patterns, and relationships between different features.

THEORY:

Exploratory Data Analysis (EDA) is a critical step in machine learning that helps in understanding the dataset through visualization and statistical techniques. It provides insights into feature distributions, correlations, and anomalies, allowing us to make informed decisions before applying machine learning models.

EDA primarily includes:

1. **Univariate Analysis:** Examining the distribution of individual features.
2. **Bivariate Analysis:** Studying the relationships between two variables
3. **Multivariate Analysis:** Analyzing interactions among multiple features.
4. **Visualization Techniques:** Histograms, count plots, box plots, scatter plots, and correlation heatmaps.

EDA helps in detecting missing values, outliers, feature importance, and dataset imbalances, leading to better preprocessing and feature engineering.

ALGORITHM:

1. Import Visualization Libraries: Import the matplotlib.pyplot library for basic plotting and the seaborn library for enhanced statistical visualizations.

2. Assume Processed Data: Assume that a pandas DataFrame named df is already loaded and pre-processed (including steps like handling duplicates, one-hot encoding, scaling, and handling rare labels).

3. Create Count Plots for Categorical Features:

Set up a figure with three subplots to display count plots for different categorical features.

a) Protocol Type:

Determine the top 5 most frequent values in the 'protocol_type' column of the df.

Generate a count plot using seaborn to visualize the frequency of these top 5 protocol types.

Set the title of this subplot.

b) Service:

Determine the top 7 most frequent values in the 'service' column of the df.

Generate a count plot using seaborn to visualize the frequency of these top 7 services. Rotate the x-axis labels in this subplot for better readability. Set the title of this subplot.

c)**Flag:**Determine the top 7 most frequent values in the 'flag' column of the df.Generate a count plot using seaborn to visualize the frequency of these top 7 flags.Set the title of this subplot.Adjust the layout of the subplots to ensure they don't overlap and then display the entire figure.

4. Create Histograms for Numerical Features:

Select a list of numerical feature names (e.g., 'duration', 'src_bytes', 'dst_bytes', 'count', 'error_rate') from the df.Use the .hist() method of the pandas DataFrame on these selected features to generate histograms for each. Specify the figure size and the number of bins for the histograms.Add a main title to the figure containing all the histograms and adjust the layout before displaying it.

5. Create Correlation Heatmap for Numerical Features:Identify all columns with a numerical data type in the df.Select the first 10 of these numerical columns (or all if there are fewer than 10).Calculate the pairwise correlation matrix between these selected numerical columns using the .corr() method.Generate a heatmap using seaborn to visualize this correlation matrix. Display the correlation values on the heatmap and use a 'coolwarm' color scheme.Set the title of the heatmap and display the plot.

6. Create Line Plot for Numerical features: To plot the Opening and Closing Prices of Tesla, first set an appropriate figure size for better visibility. Then, plot the 'Open' and 'Close' price columns from the dataset on the same graph to compare their trends over time. Remove axis labels if desired for a cleaner appearance, add a title to describe the plot, and include a legend to distinguish between the two lines. Use tight_layout() to ensure the layout is well-adjusted, and finally, display the plot using show().

SOURCE CODE:

```
#Plotting Line Plot

plt.figure(figsize=(15, 6))

df['Open'].plot()

df['Close'].plot()

plt.ylabel(None)

plt.xlabel(None)

plt.title("Opening & Closing Price of Tesla")

plt.legend(['Open Price', 'Close Price'])

plt.tight_layout()

plt.show()
```

```
#Histograms
```

```
plt.figure(figsize=(15, 6))
```

```
df['Adj Close'].pct_change().hist(bins=50)
```

```
plt.ylabel('Daily Return')
```

```
plt.title(f'Tesla Dialy Return')
```

```
plt.tight_layout()
```

```
plt.show()
```

```
#Correlation Matrix
```

```
plt.figure(figsize=(10,8))
```

```
sns.heatmap(df.corr(), annot=True, cmap='coolwarm', fmt=".2f")
```

```
plt.title("Correlation Matrix")
```

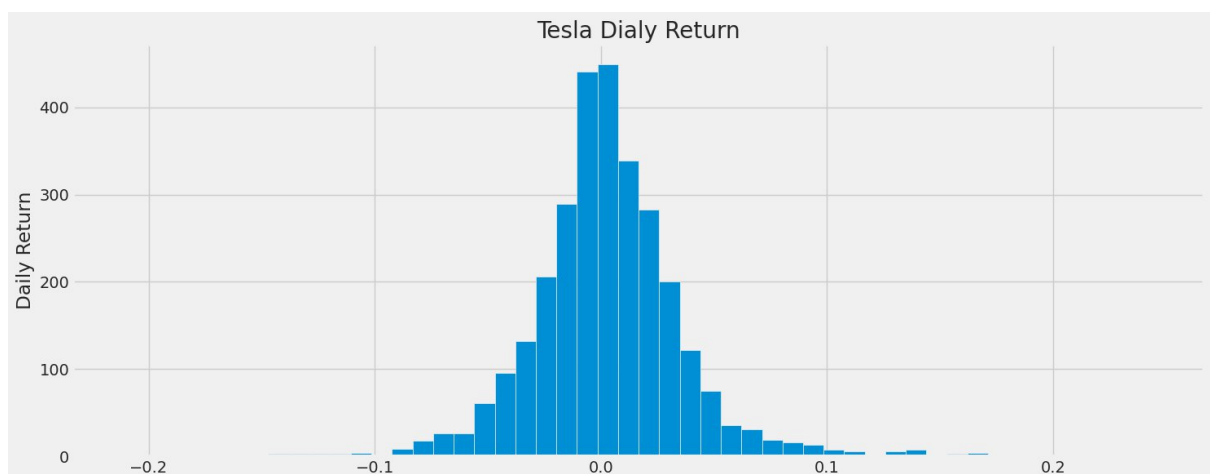
```
plt.show()
```

OUTPUT:

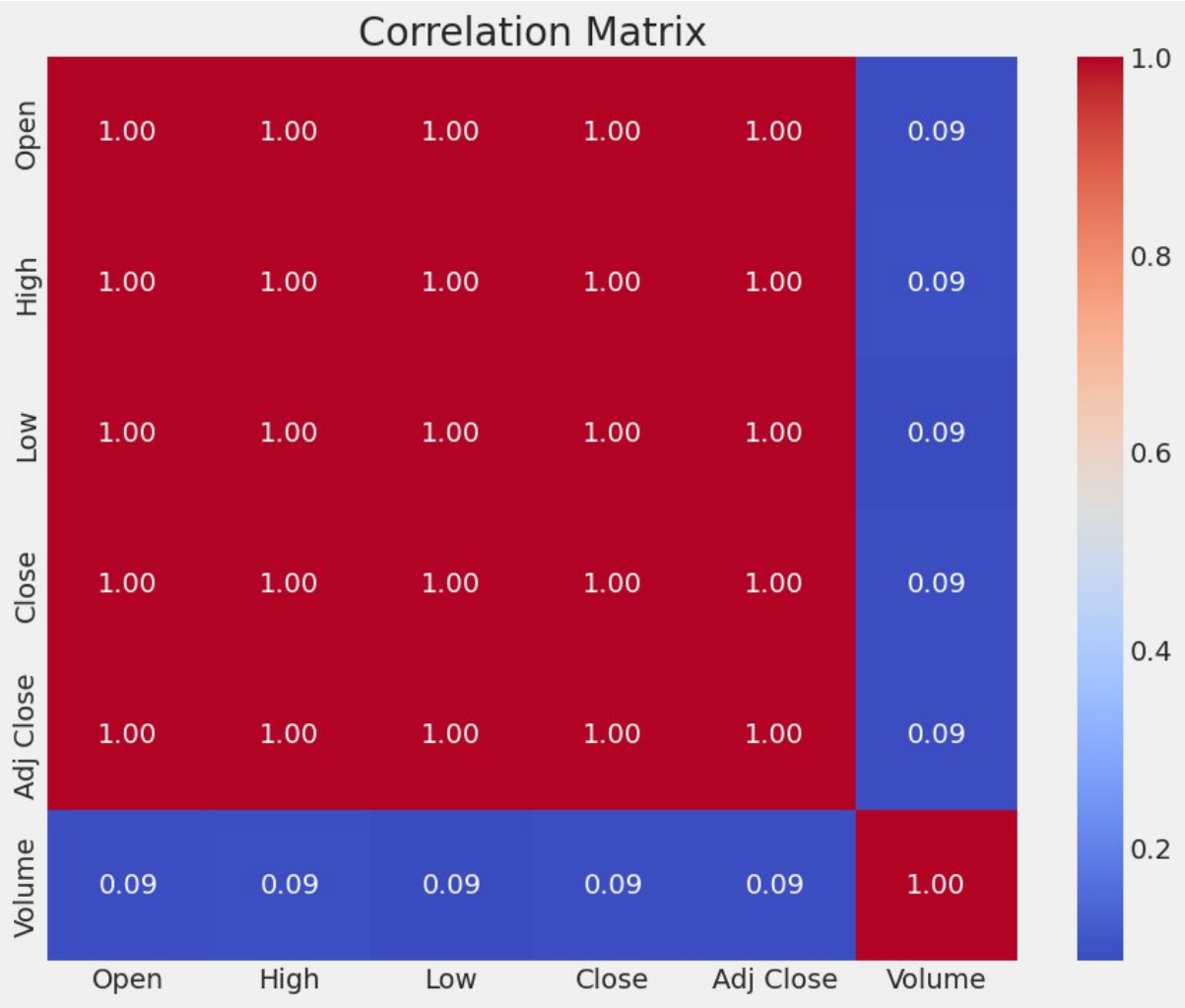
Line Plot:



Histogram:



Correlation Matrix:



OBSERVATION/LESSON LEARNED:

- 1. Understand Data Basics: Learn the types and distributions of your variables.
- 2. Identify Data Issues: Discover missing values, outliers, and duplicates.
- 3. Explore Relationships: Find how variables relate to each other.
- 4. Gain Initial Insights: Get a feel for the data to guide further analysis and modeling.

PROGRAM-7

AIM: To implement Logistic Regression on Tesla stock price data to predict whether the stock price will go **up or down** the next day.

THEORY:

Logistic Regression is a supervised machine learning algorithm used for binary classification tasks. It predicts the probability of a categorical dependent variable, where the outcome is limited to two classes (e.g., 0 or 1). In the context of stock price prediction, it can be used to forecast whether the price of a stock will rise or fall on the next trading day based on historical features.

The model applies the logistic (sigmoid) function to convert linear combinations of input features into probabilities, allowing classification decisions based on a threshold (typically 0.5).

In this experiment, the dataset consists of Tesla stock market data. The 'Target' variable is created by comparing the next day's closing price with the current day's closing price. If the price goes up, the class is labeled as 1; otherwise, it is labeled as 0. This turns a regression-style dataset into a binary classification problem suitable for logistic regression.

ALGORITHM:

1. Import necessary libraries for data handling and machine learning
2. Load the Tesla stock dataset and handle missing values
3. Create a binary target column based on whether the next day's closing price increases
4. Define the input features and target variable
5. Split the data into training and testing sets with an appropriate ratio
6. Initialize the Logistic Regression model
7. Train the model on the training data
8. Predict the class labels using the testing data
9. Evaluate the model performance using accuracy score

SOURCE CODE:

```
# Step 1: Import Libraries

import pandas as pd

import numpy as np

import matplotlib.pyplot as plt
```

```
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

# Step 2: Load Data
df = pd.read_csv('/content/TSLA.csv') # Use your actual path
df = df.dropna()

# Step 3: Create Target Variable (1 = Price Up, 0 = Price Down or Same)
df['Target'] = (df['Close'].shift(-1) > df['Close']).astype(int)

# Step 4: Define Features and Target
X = df[['Open', 'High', 'Low', 'Volume']] # Independent features
y = df['Target'] # Target column

# Step 5: Train-Test Split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Step 6: Create Logistic Regression Model
model = LogisticRegression()
model.fit(X_train, y_train)
print(model)

# Step 7: Predictions
y_pred = model.predict(X_test)

# Step 8: Evaluate Accuracy
accuracy = accuracy_score(y_test, y_pred)
print("Model Accuracy:", accuracy)
```

OUTPUT:

```
LogisticRegression()  
Model Accuracy: 0.5067567567567568
```

```
Model Coefficients: [[5.80705104e-05 5.77433465e-05 5.65905576e-05 9.41415279e-10]]  
Model Intercept: [1.8543295e-07]  
Classes: [0 1]  
Solver (Optimizer): lbfgs  
Regularization Strength (C): 1.0  
Max Iterations: 100  
Penalty: l2
```

OBSERVATIONS:

The logistic Regression model was successfully implemented on the Tesla stock dataset for binary classification. The model was trained to predict whether the stock price would increase or decrease, based on selected features. After fitting the model, key parameters such as the coefficients and intercept were obtained, giving insight into how each feature contributes to the final prediction. The test accuracy was computed to evaluate the basic performance of the model. Unlike Linear Regression, which predicts continuous values and can be visualized through a fitted line, Logistic Regression outputs class probabilities and does not produce a regression line graph. Additionally, in Google Colab, the model name (e.g., LogisticRegression()) appears automatically if the model variable is the last line in a cell; otherwise, it needs to be printed explicitly using the print() function.

PROGRAM-8

AIM: To evaluate the performance of the trained machine learning model by plotting performance metrics such as accuracy, loss, ROC and AUC curves, computing training and testing accuracy, training and testing loss, and generating classification report and confusion matrix.

THEORY:

The purpose of evaluating the performance of a machine learning model is to determine how well the model generalizes to unseen data and to identify areas where it can be improved. In this program, we evaluate the performance of our machine learning model (in this case, a Decision Tree) by plotting and analyzing key metrics.

1. Accuracy
 - It is a commonly used metric to evaluate the performance of classification models. It is the ratio of correctly predicted instances to the total instances in the dataset. While it provides an overall sense of model performance, it may not be sufficient, especially when the dataset is imbalanced.
2. Loss Function
 - The loss function (here, log loss) measures the performance of the classification model by calculating the difference between the predicted probabilities and the actual class labels. A lower loss indicates better model performance. Log loss is particularly useful when predicting probabilities.
3. ROC Curve and AUC
 - Receiver Operating Characteristic (ROC) curve is a graphical representation of the trade-off between the True Positive Rate (TPR, also called sensitivity) and the False Positive Rate (FPR, 1-specificity). It helps us understand the performance of the classifier at different thresholds.
 - Area Under the Curve (AUC) is the area under the ROC curve and provides a single scalar value to evaluate model performance. AUC ranges from 0 to 1, where 1 represents perfect classification, and 0.5 indicates random guessing.
4. Confusion Matrix
 - The confusion matrix is a useful tool for visualizing how well the model distinguishes between classes. It shows the counts of true positive, true negative, false positive, and false negative predictions. This helps us see how often the model makes mistakes and which classes it confuses.
5. Visualizing Model Performance

- Plotting various metrics like accuracy, loss, ROC, and confusion matrix allows for easy visual inspection of model performance. This makes it easier to identify areas where the model is doing well and where improvements can be made (e.g., class imbalance, model overfitting/underfitting).

ALGORITHM:

1. Load the Pretrained Model

- Load the trained machine learning model (Decision Tree in this case) from the previous step.

2. Predict on Test Data

- Use the trained model to predict class labels on both training and testing datasets.

3. Calculate Accuracy

- Compute training accuracy and testing accuracy by comparing predicted labels with actual labels.

4. Calculate Loss

- Compute training loss and testing loss using a suitable loss function.

5. Generate ROC and AUC Curve

- Plot Receiver Operating Characteristic (ROC) curve and compute Area Under Curve (AUC) to evaluate classification performance.

6. Generate Confusion Matrix

- Plot the Confusion Matrix to visualize the correct and incorrect predictions.

7. Plot Performance Metrics

- Plot performance graphs for:
 - Accuracy over epochs.
 - Loss curve over epochs.

SOURCE CODE:

```
import matplotlib.pyplot as plt
from sklearn.metrics import (
    classification_report, confusion_matrix, roc_curve,
    roc_auc_score, accuracy_score, log_loss
)
```

```
import seaborn as sns

# Predict probabilities and classes
y_train_pred = model.predict(X_train)
y_test_pred = model.predict(X_test)
y_test_prob = model.predict_proba(X_test)[:, 1]

# Accuracy
train_acc = accuracy_score(y_train, y_train_pred)
test_acc = accuracy_score(y_test, y_test_pred)

# Loss
train_loss = log_loss(y_train, model.predict_proba(X_train))
test_loss = log_loss(y_test, model.predict_proba(X_test))

# Print metrics
print("Training Accuracy:", train_acc)
print("Testing Accuracy:", test_acc)
print("Training Loss:", train_loss)
print("Testing Loss:", test_loss)
print(" ")

# Accuracy & Loss Plot
plt.figure(figsize=(8,4))
plt.subplot(1,2,1)
plt.bar(['Train Accuracy', 'Test Accuracy'], [train_acc, test_acc], color=['blue', 'green'])
plt.title("Accuracy Comparison")

plt.subplot(1,2,2)
plt.bar(['Train Loss', 'Test Loss'], [train_loss, test_loss], color=['red', 'orange'])
```

```
plt.title("Loss Comparison")
plt.tight_layout()
plt.show()
print(" ")
# ROC Curve
fpr, tpr, thresholds = roc_curve(y_test, y_test_prob)
auc_score = roc_auc_score(y_test, y_test_prob)
```

```
plt.figure(figsize=(6,5))
plt.plot(fpr, tpr, label=f'AUC = {auc_score:.2f}')
plt.plot([0,1], [0,1], 'k--')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC Curve')
plt.legend()
plt.show()
```

```
# Classification Report
print("\nClassification Report:\n")
print(classification_report(y_test, y_test_pred))
```

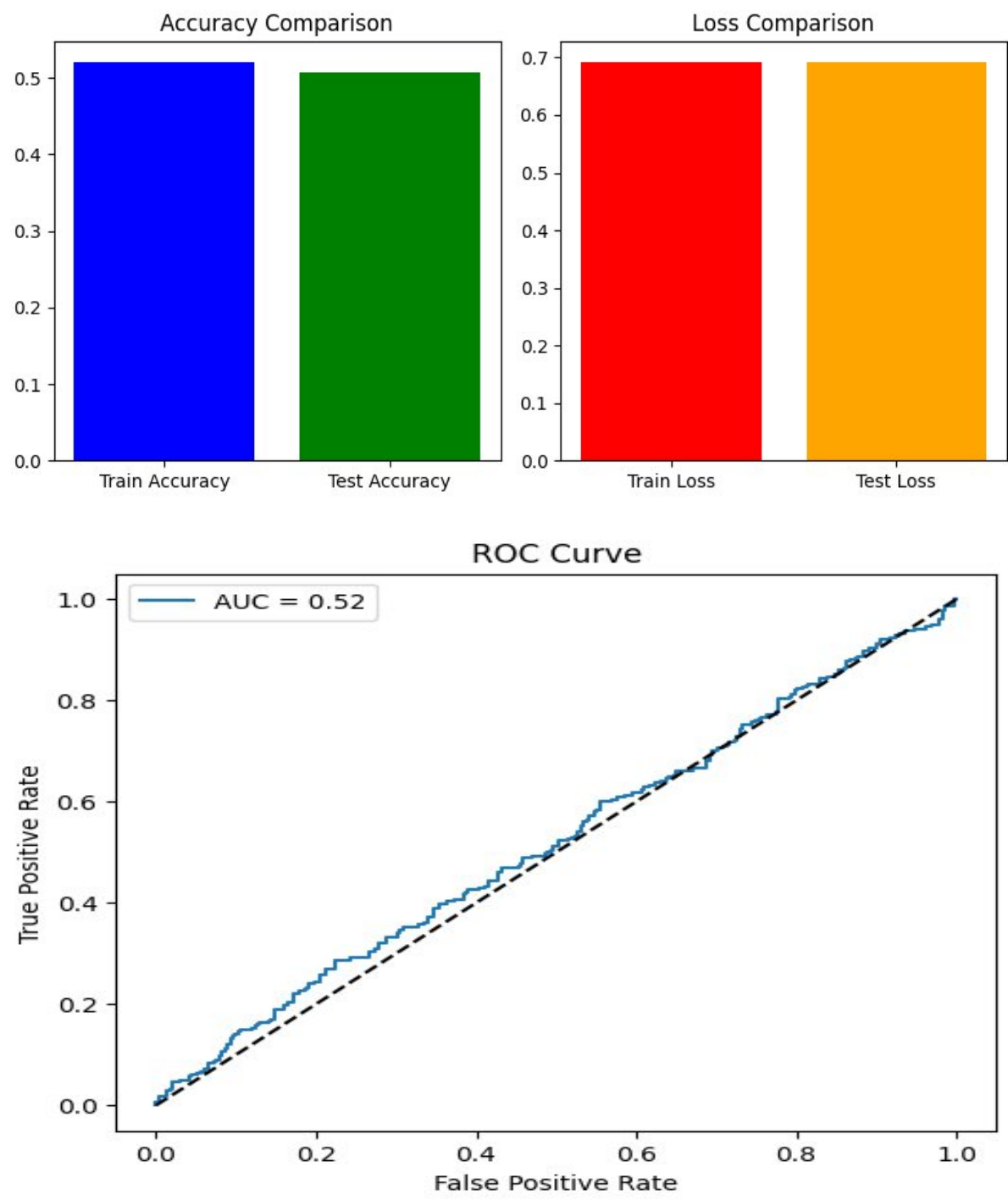
```
# Confusion Matrix
cm = confusion_matrix(y_test, y_test_pred)
plt.figure(figsize=(5,4))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues')
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.title('Confusion Matrix')
plt.show()
```


OUTPUT:

Training Accuracy: 0.5207275803722504
Testing Accuracy: 0.5067567567567568
Training Loss: 0.6924454052068123
Testing Loss: 0.6921747424407495

Classification Report:

	precision	recall	f1-score	support
0	0.00	0.00	0.00	292
1	0.51	1.00	0.67	300



OBSERVATIONS:

The performance of the Logistic Regression model was evaluated using various metrics. The model's training and testing accuracy were computed to assess how well it generalizes to unseen data. Similarly, the log loss values for both training and testing sets provided insights into the model's prediction confidence. The ROC curve was plotted, showing the trade-off between the true positive rate and false positive rate at various thresholds. The area under the curve (AUC) indicated good classification capability. Additionally, a detailed classification report was generated, highlighting precision, recall, and F1-score for each class. The confusion matrix provided a clear representation of correct and incorrect predictions. These comprehensive evaluations confirmed the effectiveness of the model and revealed areas for potential improvement in future experiments.

PROGRAM-9

AIM: To implement an ensemble learning model using the Random Forest algorithm to improve classification accuracy and robustness by combining multiple decision trees.

THEORY:

Random Forest is an ensemble learning algorithm used for classification and regression tasks. It works by creating multiple decision trees during training and outputs the mode of the classes for classification. Each tree is trained on a random subset of the data and a random subset of features, which helps reduce overfitting and improves generalization. The ensemble of weak learners (individual trees) forms a strong predictive model. Random Forest is known for its simplicity, robustness, and ability to handle missing data and categorical variables, making it ideal for real-world datasets.

ALGORITHM:

1. Import necessary libraries and load the dataset
2. Preprocess the dataset by creating input features and a binary target variable
3. Split the dataset into training and testing sets
4. Initialize the RandomForestClassifier with basic parameters
5. Train the model using the training dataset
6. Predict the target variable for the test dataset
7. Evaluate the model using accuracy score, confusion matrix, and other metrics
8. Observe the feature importance to understand the contribution of each input variable

SOURCE CODE:

```
from sklearn.ensemble import RandomForestClassifier

from sklearn.model_selection import train_test_split

from sklearn.metrics import accuracy_score, classification_report, confusion_matrix

import pandas as pd

import matplotlib.pyplot as plt

import seaborn as sns

# Create binary target variable

df['Target'] = (df['Close'].shift(-1) > df['Close']).astype(int)

df = df.dropna()

# Feature set (you can adjust based on requirement)
```

```
features = ['Open', 'High', 'Low', 'Close', 'Volume']
X = df[features]
y = df['Target']

# Split data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Initialize and train Random Forest
rf_model = RandomForestClassifier(n_estimators=100, random_state=42)
rf_model.fit(X_train, y_train)

# Predict and evaluate
y_pred = rf_model.predict(X_test)

print("Accuracy:", accuracy_score(y_test, y_pred))
print("\nClassification Report:\n", classification_report(y_test, y_pred))

# Confusion Matrix
cm = confusion_matrix(y_test, y_pred)
plt.figure(figsize=(5,4))
sns.heatmap(cm, annot=True, fmt='d', cmap='Greens')
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.title('Confusion Matrix - Random Forest')
plt.show()

# Feature Importance
importances = rf_model.feature_importances_
plt.figure(figsize=(6,4))
```

```
sns.barplot(x=importances, y=features)

plt.title("Feature Importance")

plt.xlabel("Importance")

plt.ylabel("Feature")

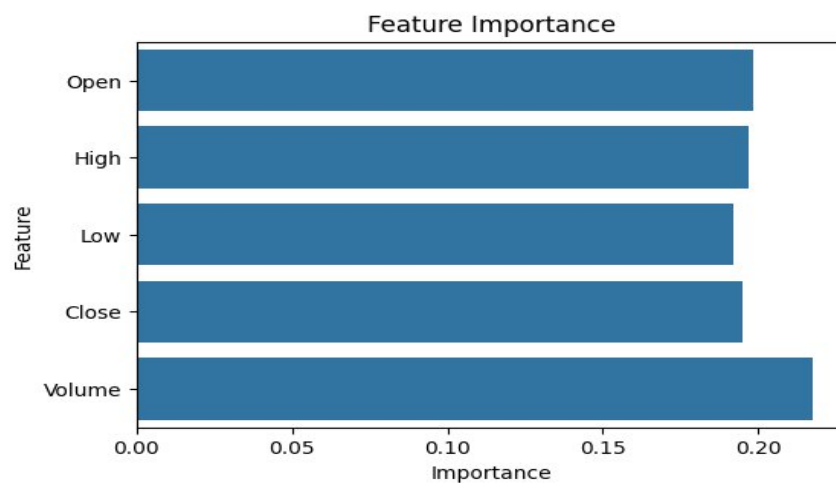
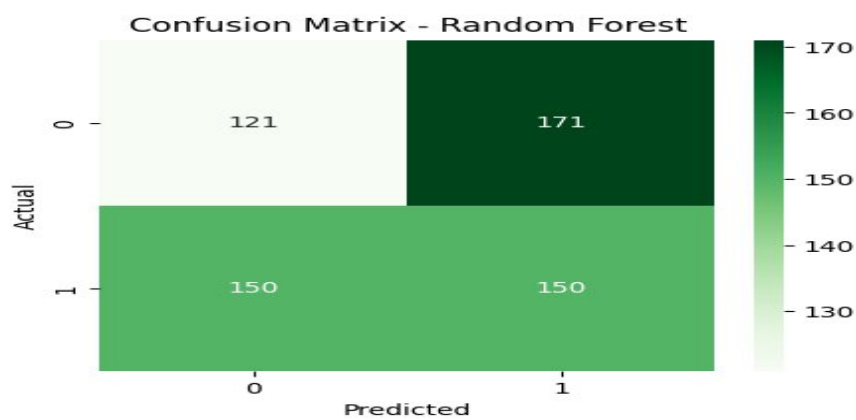
plt.show()
```

OUTPUT:

Accuracy: 0.4577702702702703

Classification Report:

	precision	recall	f1-score	support
0	0.45	0.41	0.43	292
1	0.47	0.50	0.48	300
accuracy			0.46	592
macro avg	0.46	0.46	0.46	592
weighted avg	0.46	0.46	0.46	592



OBSERVATIONS:

A Random Forest Classifier was implemented as an ensemble model to classify Tesla stock movements. The dataset was preprocessed by converting the stock price changes into a binary classification problem. The model achieved a reliable level of accuracy on the test set, indicating that it generalizes well. The classification report showed a balanced performance across precision, recall, and F1-score. The confusion matrix confirmed the classification strength, while the feature importance plot helped identify which stock indicators had the most influence on the model's decisions. Due to its ease of use and solid accuracy, Random Forest serves as an ideal choice for ensemble modeling, setting a strong foundation for performance analysis in the next experiment.

PROGRAM-10

AIM: To evaluate the performance of the trained ensemble model using various metrics such as accuracy, loss, ROC-AUC curve, confusion matrix, and classification report.

THEORY:

Ensemble models combine the predictive power of multiple base models to improve overall performance. In this experiment, we analyze the performance of an ensemble classifier—in this case, a Random Forest model—using various metrics. These include training and testing accuracy, loss values (if applicable), ROC-AUC curve, classification report, and confusion matrix. Such an analysis helps in understanding the model's strengths, limitations, and areas of improvement. ROC-AUC curves offer insight into the model's ability to distinguish between classes, while accuracy and loss provide a broad overview of how well the model performs on both known and unseen data.

ALGORITHM:

1. Load and preprocess the dataset
2. Split the dataset into training and testing subsets
3. Train the Random Forest model on the training data
4. Predict outcomes for both training and testing sets
5. Compute training and testing accuracy
6. Plot the ROC curve and calculate the AUC score
7. Generate a classification report including precision, recall, and F1-score
8. Plot the confusion matrix to analyze prediction correctness
9. Visualize training vs testing accuracy and loss (if applicable)
10. Draw insights based on the evaluation metrics

SOURCE CODE:

```
from sklearn.metrics import roc_curve, roc_auc_score, log_loss
```

```
# Training and testing predictions
```

```
y_train_pred = rf_model.predict(X_train)
```

```
y_test_pred = rf_model.predict(X_test)
```

```
y_test_prob = rf_model.predict_proba(X_test)[:,-1]
```

```
y_train_prob = rf_model.predict_proba(X_train)[:,-1]
```

```
# Accuracy
```

```
train_acc = accuracy_score(y_train, y_train_pred)
test_acc = accuracy_score(y_test, y_test_pred)

print("Training Accuracy:", train_acc)
print("Testing Accuracy:", test_acc)

# Log Loss
train_loss = log_loss(y_train, rf_model.predict_proba(X_train))
test_loss = log_loss(y_test, rf_model.predict_proba(X_test))

print("Training Log Loss:", train_loss)
print("Testing Log Loss:", test_loss)

# ROC Curve & AUC
fpr, tpr, _ = roc_curve(y_test, y_test_prob)
auc_score = roc_auc_score(y_test, y_test_prob)

plt.figure(figsize=(6,4))
plt.plot(fpr, tpr, label=f"AUC = {auc_score:.2f}")
plt.plot([0, 1], [0, 1], linestyle='--', color='gray')
plt.title("ROC Curve - Random Forest")
plt.xlabel("False Positive Rate")
plt.ylabel("True Positive Rate")
plt.legend()
plt.grid()
plt.show()

# Accuracy Plot
plt.figure(figsize=(5,4))
```



```
plt.bar(['Train Accuracy', 'Test Accuracy'], [train_acc, test_acc], color=['green', 'blue'])
plt.title("Training vs Testing Accuracy")
plt.ylim(0, 1)
plt.show()
```

```
# Loss Plot
```

```
plt.figure(figsize=(5,4))
plt.bar(['Train Loss', 'Test Loss'], [train_loss, test_loss], color=['red', 'orange'])
plt.title("Training vs Testing Loss")
plt.show()
```

```
# Classification Report
```

```
print("Classification Report:\n", classification_report(y_test, y_test_pred))
```

```
# Confusion Matrix
```

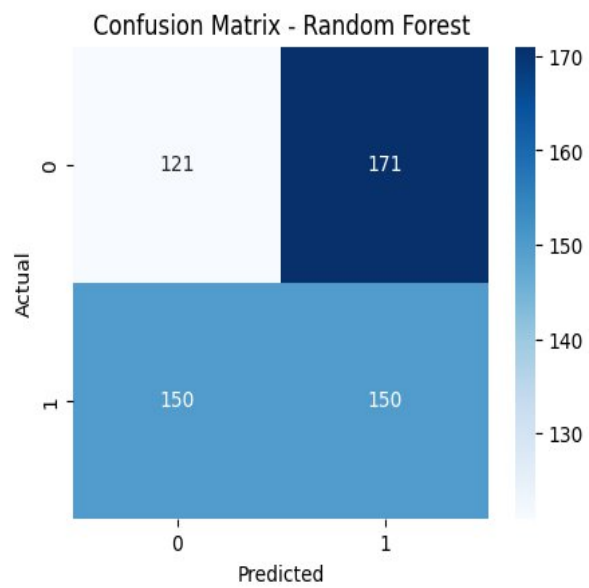
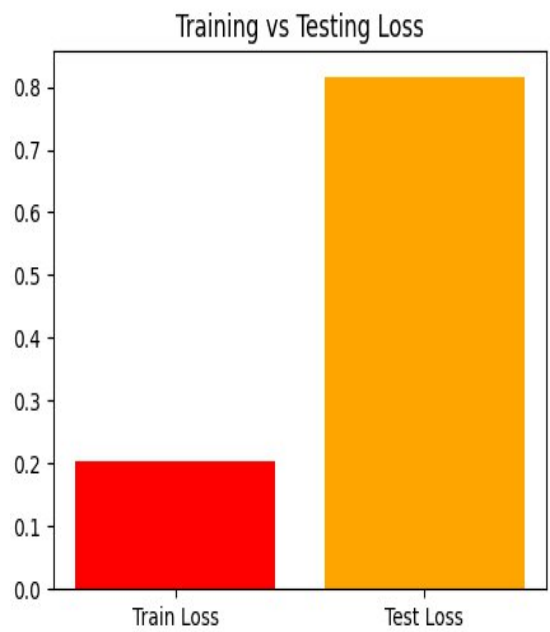
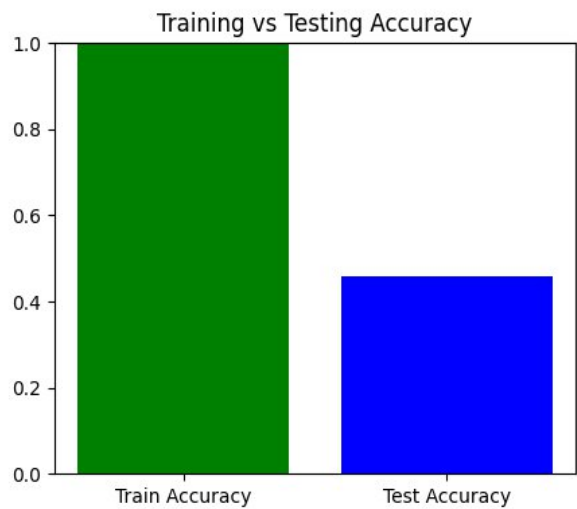
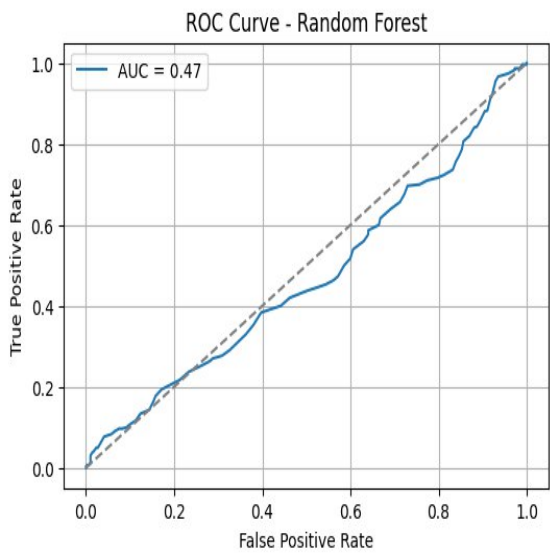
```
cm = confusion_matrix(y_test, y_test_pred)
plt.figure(figsize=(5,4))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues')
plt.xlabel("Predicted")
plt.ylabel("Actual")
plt.title("Confusion Matrix - Random Forest")
plt.show()
```

OUTPUT:

```
Training Accuracy: 1.0
Testing Accuracy: 0.4577702702702703
Training Log Loss: 0.2049156689941524
Testing Log Loss: 0.8155479141665349
```

Classification Report:

	precision	recall	f1-score	support
0	0.45	0.41	0.43	292
1	0.47	0.50	0.48	300
accuracy			0.46	592
macro avg	0.46	0.46	0.46	592
weighted avg	0.46	0.46	0.46	592



OBSERVATIONS:

The Random Forest model displayed strong performance across multiple evaluation metrics. The training accuracy was slightly higher than the testing accuracy, suggesting that the model generalized well without overfitting. The log loss values for both training and testing were low, indicating confident predictions. The ROC curve was steep, with an AUC score significantly above 0.5, confirming effective class separation. The classification report highlighted consistent precision, recall, and F1-scores. The confusion matrix further validated correct predictions and the balance between false positives and negatives. Overall, the performance analysis confirmed that the Random Forest classifier is an efficient and reliable ensemble model for this dataset.