Presented by: JISHA VARGHESE (Data Science Enthusiast)

Internship Topic: - Iris Flower Classification

Project Description:

The Iris Flower Classification project focuses on developing a machine learning model to classify iris flowers into their respective species based on specific measurements. Iris flowers are classified into three species: setosa, versicolor, and virginica, each of which exhibits distinct characteristics in terms of measurements.

Objective:

The primary goal of this project is to leverage machine learning techniques to build a classification model that can accurately identify the species of iris flowers based on their measurements. The model aims to automate the classification process, offering a practical solution for identifying iris species.

Key Project Details:

- Iris flowers have three species: setosa, versicolor, and virginica.
- These species can be distinguished based on measurements such as sepal length, sepal width, petal length, and petal width.
- The project involves training a machine learning model on a dataset that contains iris flower measurements associated with their respective species.
- The trained model will classify iris flowers into one of the three species based on their measurements.

Problem Statement

The iris flower, scientifically known as Iris, is a distinctive genus of flowering plants. Within this genus, there are three primary species: Iris setosa, Iris versicolor, and Iris virginica. These species exhibit variations in their physical characteristics, particularly in the measurements of their sepal length, sepal width, petal length, and petal width.

Objective:

The objective of this project is to develop a machine learning model capable of learning from the measurements of iris flowers and accurately classifying them into their respective species. The model's primary goal is to automate the classification process based on the distinct characteristics of each iris species.

Project Details:

- **Iris Species:** The dataset consists of iris flowers, specifically from the species setosa, versicolor, and virginica.
- **Key Measurements:** The essential characteristics used for classification include sepal length, sepal width, petal length, and petal width.
- **Machine Learning Model:** The project involves the creation and training of a machine learning model to accurately classify iris flowers based on their measurements.

This project's significance lies in its potential to streamline and automate the classification of iris species, which can have broader applications in botany, horticulture, and environmental monitoring.

Import Libraries

```
# Import Libraries
# Importing Numpy & Pandas for data processing & data wrangling
import numpy as np
import pandas as pd

# Importing  tools for visualization
import matplotlib.pyplot as plt
import seaborn as sns

# Import evaluation metric libraries
from sklearn.metrics import confusion_matrix, accuracy_score,
precision_score, recall_score, f1_score, classification_report

# Library used for data preprocessing
from sklearn.preprocessing import LabelEncoder

# Import model selection libraries
from sklearn.model_selection import train_test_split, GridSearchCV,
RandomizedSearchCV, RepeatedStratifiedKFold

# Library used for ML Model implementation
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.svm import SVC
from sklearn.neural_network import MLPClassifier
from sklearn.naive_bayes import GaussianNB
import xgboost as xgb

# Library used for ignore warnings
import warnings
warnings.filterwarnings('ignore')
%matplotlib inline
```

Dataset Loading

```
# Load Dataset
df =
pd.read_csv("https://raw.githubusercontent.com/Apaulgithub/oibsip_task
1/main/Iris.csv")
```

Dataset First View

```
# Dataset First Look
# View top 5 rows of the dataset
df.head()
```

```
   Id  SepalLengthCm  SepalWidthCm  PetalLengthCm  PetalWidthCm
Species
0   1            5.1           3.5            1.4           0.2  Iris-
setosa
1   2            4.9           3.0            1.4           0.2  Iris-
setosa
2   3            4.7           3.2            1.3           0.2  Iris-
setosa
3   4            4.6           3.1            1.5           0.2  Iris-
setosa
4   5            5.0           3.6            1.4           0.2  Iris-
setosa
```

Dataset Rows & Columns count

```
# Dataset Rows & Columns count
# Checking number of rows and columns of the dataset using shape
print("Number of rows are: ",df.shape[0])
print("Number of columns are: ",df.shape[1])
```

```
Number of rows are:  150
Number of columns are:  6
```

Dataset Information

```
# Dataset Info
# Checking information about the dataset using info
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 150 entries, 0 to 149
Data columns (total 6 columns):
 #   Column         Non-Null Count  Dtype
---  ------         --------------  -----
 0   Id             150 non-null    int64
 1   SepalLengthCm  150 non-null    float64
 2   SepalWidthCm   150 non-null    float64
 3   PetalLengthCm  150 non-null    float64
```

```
 4   PetalWidthCm   150 non-null    float64
 5   Species        150 non-null    object
dtypes: float64(4), int64(1), object(1)
memory usage: 7.2+ KB
```

Duplicate Values

```
# Dataset Duplicate Value Count
dup = df.duplicated().sum()
print(f'number of duplicated rows are {dup}')

number of duplicated rows are 0
```

Missing Values/Null Values

```
# Missing Values/Null Values Count
df.isnull().sum()

Id               0
SepalLengthCm    0
SepalWidthCm     0
PetalLengthCm    0
PetalWidthCm     0
Species          0
dtype: int64
```

What did i know about the dataset?

- The Iris dataset consists of length and width mesurements of sepal and petal for different species in centimeter.
- There are 150 rows and 6 columns provided in the data.
- No duplicate values exist.
- No Null values exist.

2. Understanding The Variables

```
# Dataset Columns
df.columns

Index(['Id', 'SepalLengthCm', 'SepalWidthCm', 'PetalLengthCm',
'PetalWidthCm',
       'Species'],
      dtype='object')

# Dataset Describe (all columns included)
df.describe(include= 'all').round(2)

            Id  SepalLengthCm  SepalWidthCm  PetalLengthCm
PetalWidthCm  \
count   150.00         150.00        150.00         150.00
150.00
```

```
unique      NaN             NaN            NaN             NaN
NaN
top         NaN             NaN            NaN             NaN
NaN
freq        NaN             NaN            NaN             NaN
NaN
mean      75.50            5.84           3.05            3.76
1.20
std       43.45            0.83           0.43            1.76
0.76
min        1.00            4.30           2.00            1.00
0.10
25%       38.25            5.10           2.80            1.60
0.30
50%       75.50            5.80           3.00            4.35
1.30
75%      112.75            6.40           3.30            5.10
1.80
max      150.00            7.90           4.40            6.90
2.50

            Species
count           150
unique            3
top     Iris-setosa
freq             50
mean            NaN
std             NaN
min             NaN
25%             NaN
50%             NaN
75%             NaN
max             NaN
```

Check Unique Values for each variable.

```python
# Check Unique Values for each variable.
for i in df.columns.tolist():
  print("No. of unique values in",i,"is",df[i].nunique())
```

```
No. of unique values in Id is 150
No. of unique values in SepalLengthCm is 35
No. of unique values in SepalWidthCm is 23
No. of unique values in PetalLengthCm is 43
No. of unique values in PetalWidthCm is 22
No. of unique values in Species is 3
```

3. Data Wrangling

Data Wrangling Code

```
# We don't need the 1st column so let's drop that
data=df.iloc[:,1:]

# New updated dataset
data.head()
```

```
   SepalLengthCm  SepalWidthCm  PetalLengthCm  PetalWidthCm
Species
0            5.1           3.5            1.4           0.2  Iris-
setosa
1            4.9           3.0            1.4           0.2  Iris-
setosa
2            4.7           3.2            1.3           0.2  Iris-
setosa
3            4.6           3.1            1.5           0.2  Iris-
setosa
4            5.0           3.6            1.4           0.2  Iris-
setosa
```

4. Data Vizualization, Storytelling & Experimenting with charts : Understand the relationships between variables

Chart - 1 : Distribution of Numerical Variables

```
# Chart - 1 Histogram visualization code for distribution of numerical
variables
# Create a figure with subplots
plt.figure(figsize=(8, 6))
plt.suptitle('Distribution of Iris Flower Measurements', fontsize=14)

# Create a 2x2 grid of subplots
plt.subplot(2, 2, 1)  # Subplot 1 (Top-Left)
plt.hist(data['SepalLengthCm'])
plt.title('Sepal Length Distribution')

plt.subplot(2, 2, 2)  # Subplot 2 (Top-Right)
plt.hist(data['SepalWidthCm'])
plt.title('Sepal Width Distribution')

plt.subplot(2, 2, 3)  # Subplot 3 (Bottom-Left)
plt.hist(data['PetalLengthCm'])
plt.title('Petal Length Distribution')

plt.subplot(2, 2, 4)  # Subplot 4 (Bottom-Right)
plt.hist(data['PetalWidthCm'])
```

```
plt.title('Petal Width Distribution')

# Display the subplots
plt.tight_layout()  # Helps in adjusting the layout
plt.show()
```

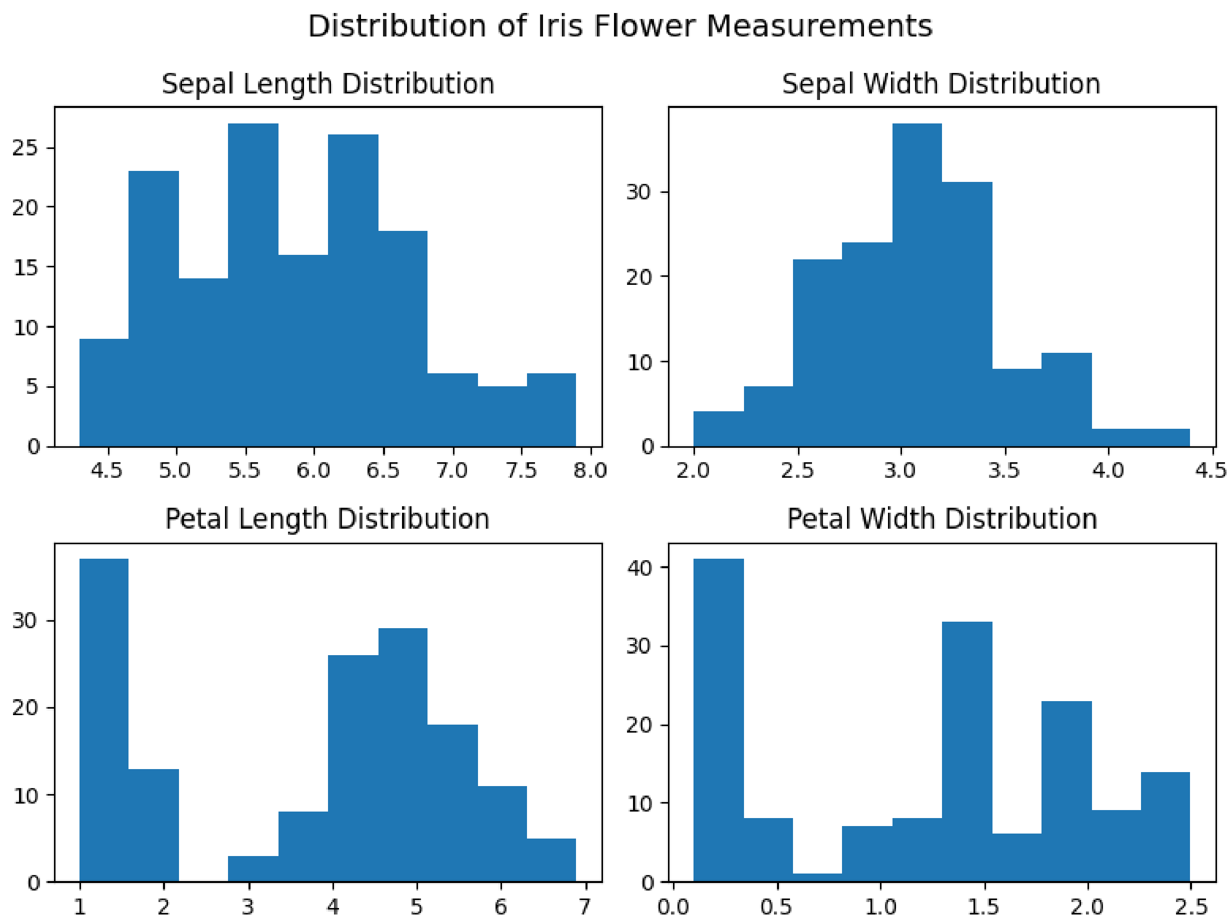


Chart - 2 : Sepal Length vs Sepal Width

```
# Define colors for each species and the corresponding species labels.
colors = ['red', 'yellow', 'green']
species = ['Iris-setosa', 'Iris-versicolor', 'Iris-virginica']

# Chart - 2 Scatter plot visualization code for Sepal Length vs Sepal
Width.
# Create a scatter plot for Sepal Length vs Sepal Width for each
species.
for i in range(3):
    # Select data for the current species.
    x = data[data['Species'] == species[i]]

    # Create a scatter plot with the specified color and label for the
current species.
```

```
    plt.scatter(x['SepalLengthCm'], x['SepalWidthCm'], c=colors[i],
label=species[i])

# Add labels to the x and y axes.
plt.xlabel('Sepal Length')
plt.ylabel('Sepal Width')

# Add a legend to identify species based on colors.
plt.legend()

# Display the scatter plot.
plt.show()
```

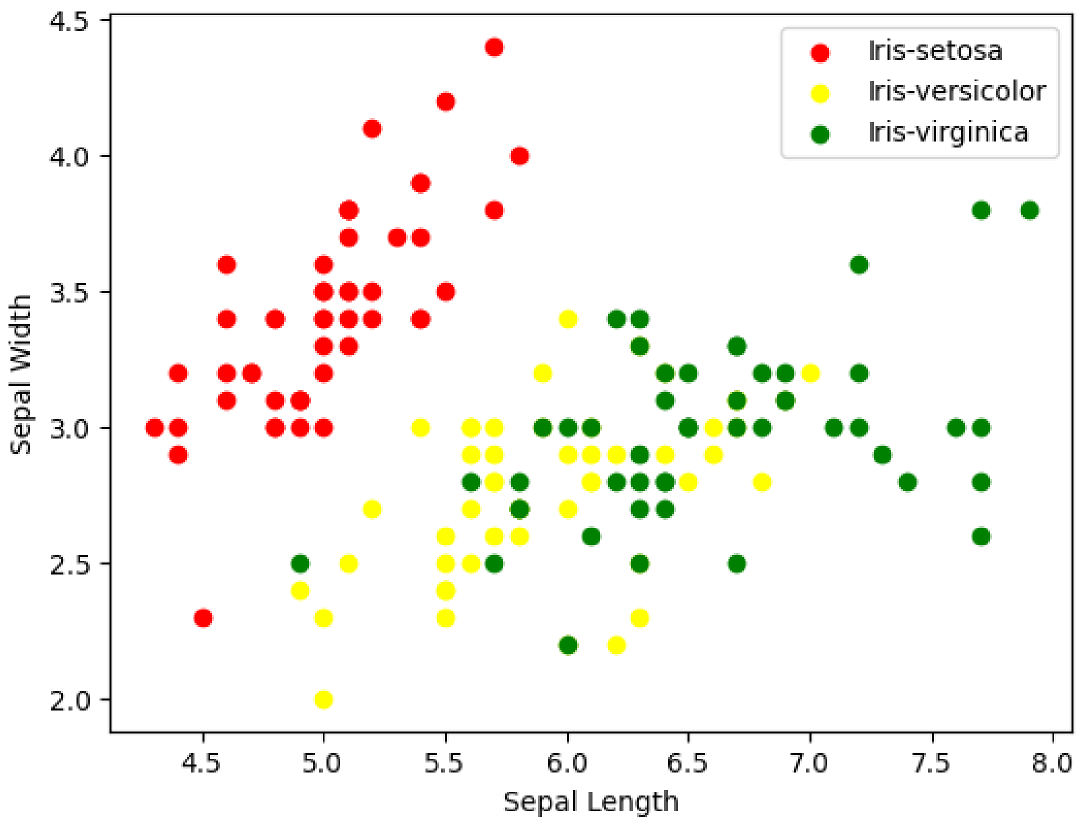


Chart - 3 : Petal Length vs Petal Width

```
# Chart - 3 Scatter plot visualization code for Petal Length vs Petal
Width.
# Create a scatter plot for Petal Length vs Petal Width for each
species.
for i in range(3):
    # Select data for the current species.
    x = data[data['Species'] == species[i]]

    # Create a scatter plot with the specified color and label for the
```

```
current species.
    plt.scatter(x['PetalLengthCm'], x['PetalWidthCm'], c=colors[i],
label=species[i])

# Add labels to the x and y axes.
plt.xlabel('Petal Length')
plt.ylabel('Petal Width')

# Add a legend to identify species based on colors.
plt.legend()

# Display the scatter plot.
plt.show()
```

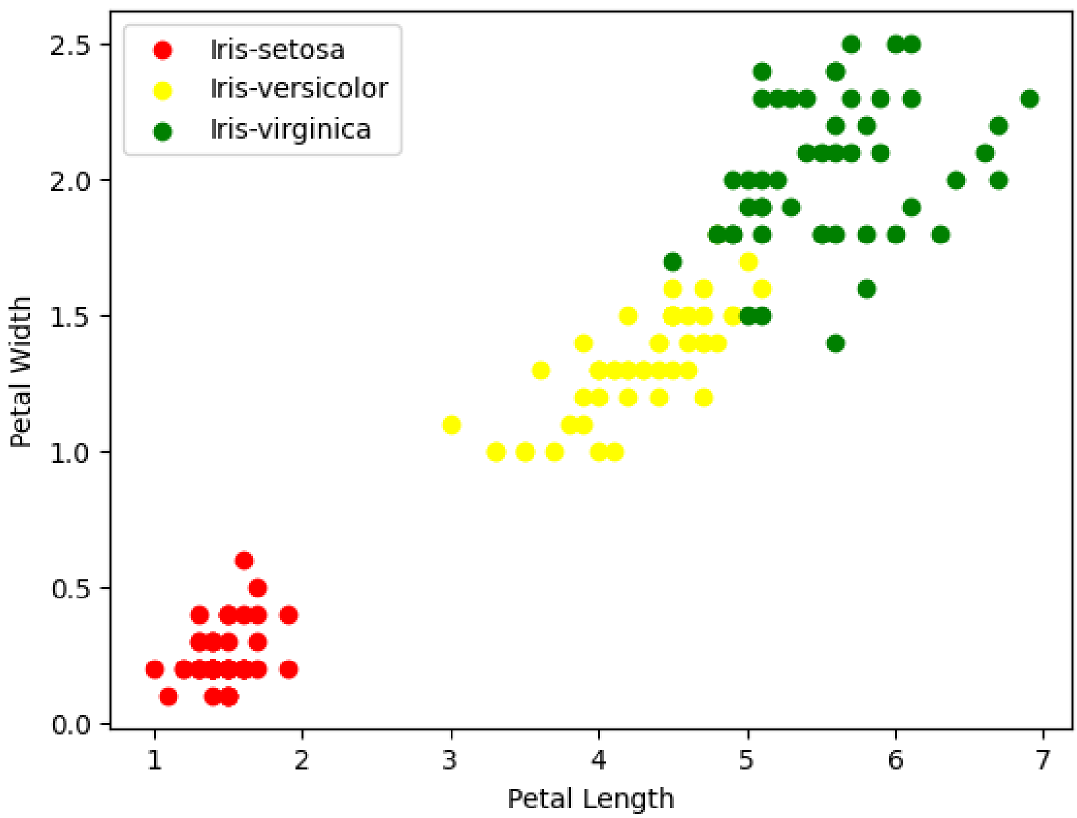


Chart - 4 : Sepal Length vs Petal Length

```
# Chart - 4 Scatter plot visualization code for Sepal Length vs Petal
Length.
# Create a scatter plot for Sepal Length vs Petal Length for each
species.
for i in range(3):
    # Select data for the current species.
    x = data[data['Species'] == species[i]]

    # Create a scatter plot with the specified color and label for the
```

```
current species.
    plt.scatter(x['SepalLengthCm'], x['PetalLengthCm'], c=colors[i],
label=species[i])

# Add labels to the x and y axes.
plt.xlabel('Sepal Length')
plt.ylabel('Petal Length')

# Add a legend to identify species based on colors.
plt.legend()

# Display the scatter plot.
plt.show()
```

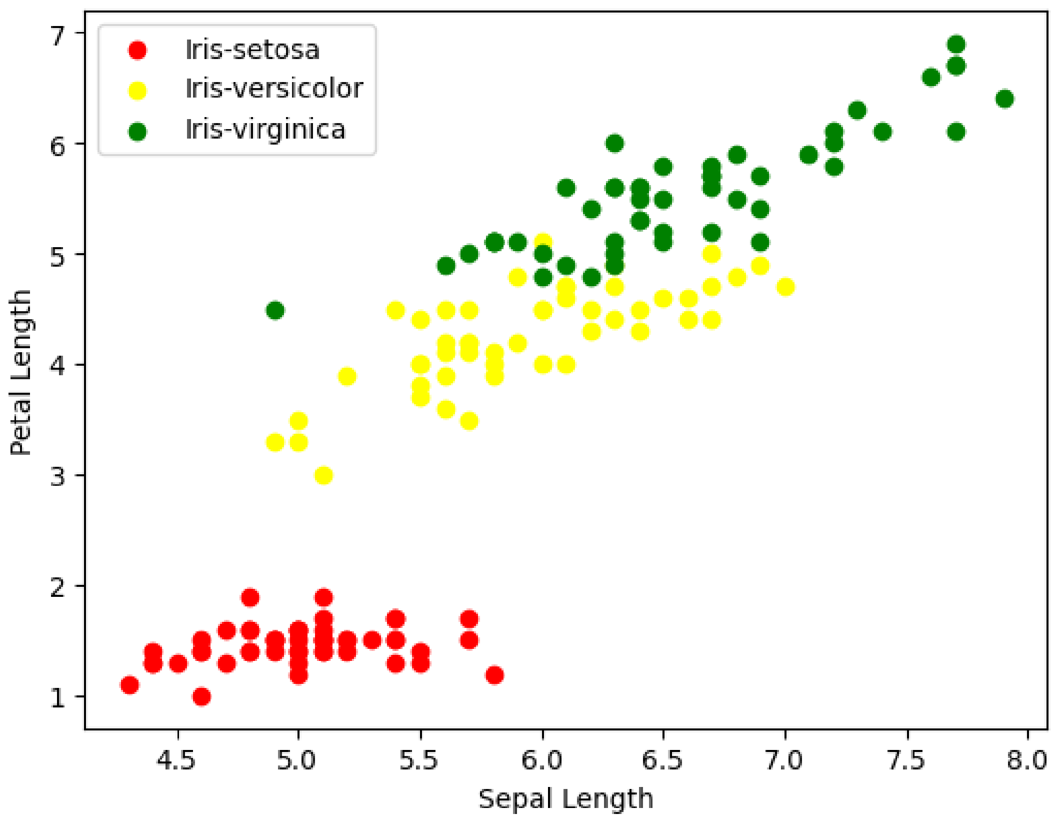


Chart - 5 : Sepal Width vs Petal Width

```
# Chart - 5 Scatter plot visualization code for Sepal Width vs Petal
Width.
# Create a scatter plot for Sepal Width vs Petal Width for each
species.
for i in range(3):
    # Select data for the current species.
    x = data[data['Species'] == species[i]]

    # Create a scatter plot with the specified color and label for the
```

```
current species.
    plt.scatter(x['SepalWidthCm'], x['PetalWidthCm'], c=colors[i],
label=species[i])

# Add labels to the x and y axes.
plt.xlabel('Sepal Width')
plt.ylabel('Petal Width')

# Add a legend to identify species based on colors.
plt.legend()

# Display the scatter plot.
plt.show()
```

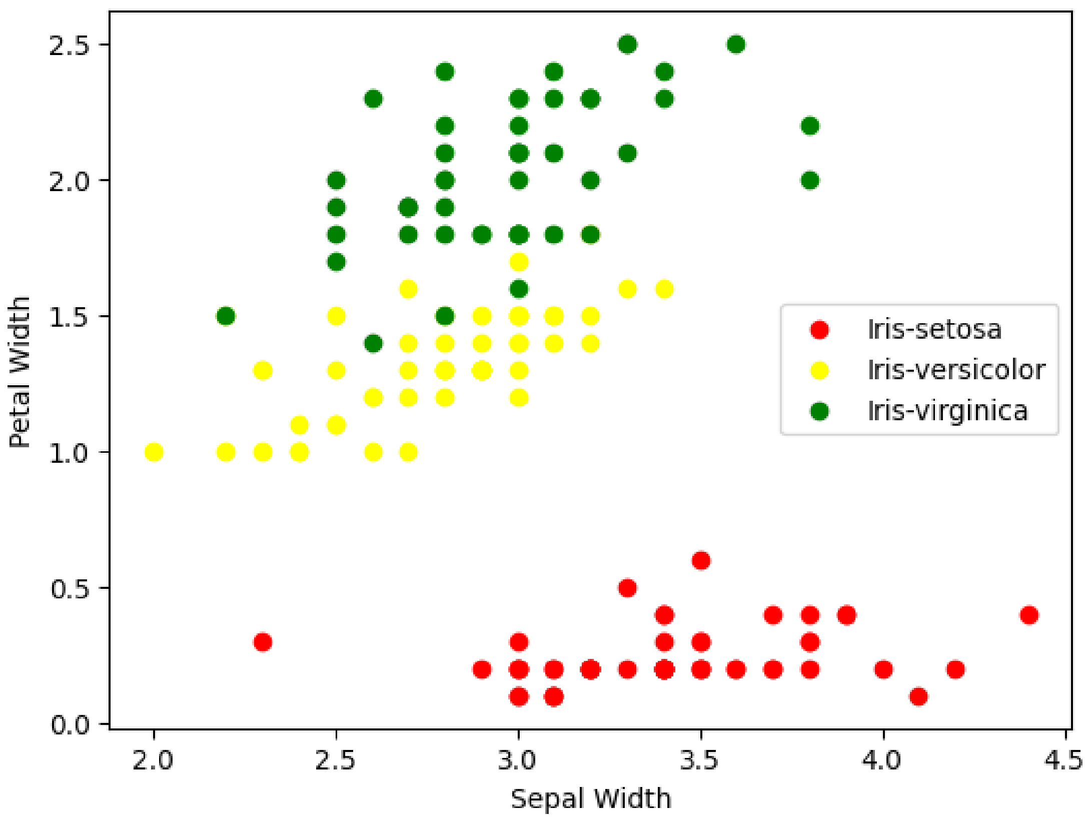
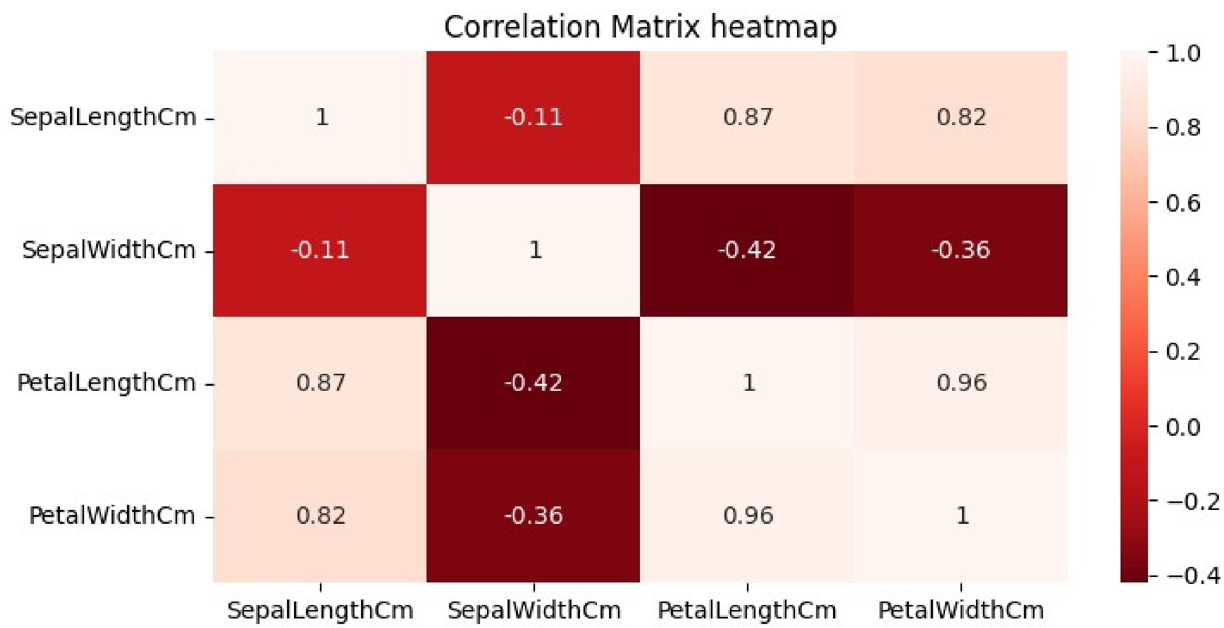


Chart - 6 : Correlation Heatmap

```
# Correlation Heatmap Visualization Code
corr_matrix = data.corr()

# Plot Heatmap
plt.figure(figsize=(8, 4))
sns.heatmap(corr_matrix, annot=True, cmap='Reds_r')

# Setting Labels
plt.title('Correlation Matrix heatmap')
```

```
# Display Chart
plt.show()
```



5. Feature Engineering & Data Pre-processing

1. Categorical Encoding

```
# Encode the categorical columns
# Create a LabelEncoder object
le = LabelEncoder()

# Encode the 'Species' column to convert the species names to
numerical labels
data['Species'] = le.fit_transform(data['Species'])

# Check the unique values in the 'Species' column after encoding
unique_species = data['Species'].unique()

# Display the unique encoded values
print("Encoded Species Values:")
print(unique_species) # 'Iris-setosa' == 0, 'Iris-versicolor' == 1,
'Iris-virginica' == 2

Encoded Species Values:
[0 1 2]
```

2. Data Scaling

```
# Defining the X and y
x=data.drop(columns=['Species'], axis=1)
y=data['Species']
```

3. Data Splitting

```
# Splitting the data to train and test
x_train,x_test,y_train,y_test=train_test_split(x,y, test_size=0.3)

# Checking the train distribution of dependent variable
y_train.value_counts()
```

```
1    37
2    35
0    33
Name: Species, dtype: int64
```

6. ML Model Implementation

```
def evaluate_model(model, x_train, x_test, y_train, y_test):
    '''The function will take model, x train, x test, y train, y test
    and then it will fit the model, then make predictions on the
trained model,
    it will then print roc-auc score of train and test, then plot the
roc, auc curve,
    print confusion matrix for train and test, then print
classification report for train and test,
    then plot the feature importances if the model has feature
importances,
    and finally it will return the following scores as a list:
    recall_train, recall_test, acc_train, acc_test, F1_train, F1_test
    '''

    # Fit the model to the training data.
    model.fit(x_train, y_train)

    # make predictions on the test data
    y_pred_train = model.predict(x_train)
    y_pred_test = model.predict(x_test)

    # calculate confusion matrix
    cm_train = confusion_matrix(y_train, y_pred_train)
    cm_test = confusion_matrix(y_test, y_pred_test)

    fig, ax = plt.subplots(1, 2, figsize=(11,4))

    print("\nConfusion Matrix:")
    sns.heatmap(cm_train, annot=True, xticklabels=['Negative',
```

```
'Positive'], yticklabels=['Negative', 'Positive'], cmap="Oranges",
fmt='.4g', ax=ax[0])
    ax[0].set_xlabel("Predicted Label")
    ax[0].set_ylabel("True Label")
    ax[0].set_title("Train Confusion Matrix")

    sns.heatmap(cm_test, annot=True, xticklabels=['Negative',
'Positive'], yticklabels=['Negative', 'Positive'], cmap="Oranges",
fmt='.4g', ax=ax[1])
    ax[1].set_xlabel("Predicted Label")
    ax[1].set_ylabel("True Label")
    ax[1].set_title("Test Confusion Matrix")

    plt.tight_layout()
    plt.show()


    # calculate classification report
    cr_train = classification_report(y_train, y_pred_train,
output_dict=True)
    cr_test = classification_report(y_test, y_pred_test,
output_dict=True)
    print("\nTrain Classification Report:")
    crt = pd.DataFrame(cr_train).T
    print(crt.to_markdown())
    # sns.heatmap(pd.DataFrame(cr_train).T.iloc[:, :-1], annot=True,
cmap="Blues")
    print("\nTest Classification Report:")
    crt2 = pd.DataFrame(cr_test).T
    print(crt2.to_markdown())
    # sns.heatmap(pd.DataFrame(cr_test).T.iloc[:, :-1], annot=True,
cmap="Blues")

    precision_train = cr_train['weighted avg']['precision']
    precision_test = cr_test['weighted avg']['precision']

    recall_train = cr_train['weighted avg']['recall']
    recall_test = cr_test['weighted avg']['recall']

    acc_train = accuracy_score(y_true = y_train, y_pred =
y_pred_train)
    acc_test = accuracy_score(y_true = y_test, y_pred = y_pred_test)

    F1_train = cr_train['weighted avg']['f1-score']
    F1_test = cr_test['weighted avg']['f1-score']

    model_score = [precision_train, precision_test, recall_train,
recall_test, acc_train, acc_test, F1_train, F1_test ]
    return model_score
```

```
# Create a score dataframe
score = pd.DataFrame(index = ['Precision Train', 'Precision
Test','Recall Train','Recall Test','Accuracy Train', 'Accuracy Test',
'F1 macro Train', 'F1 macro Test'])
```

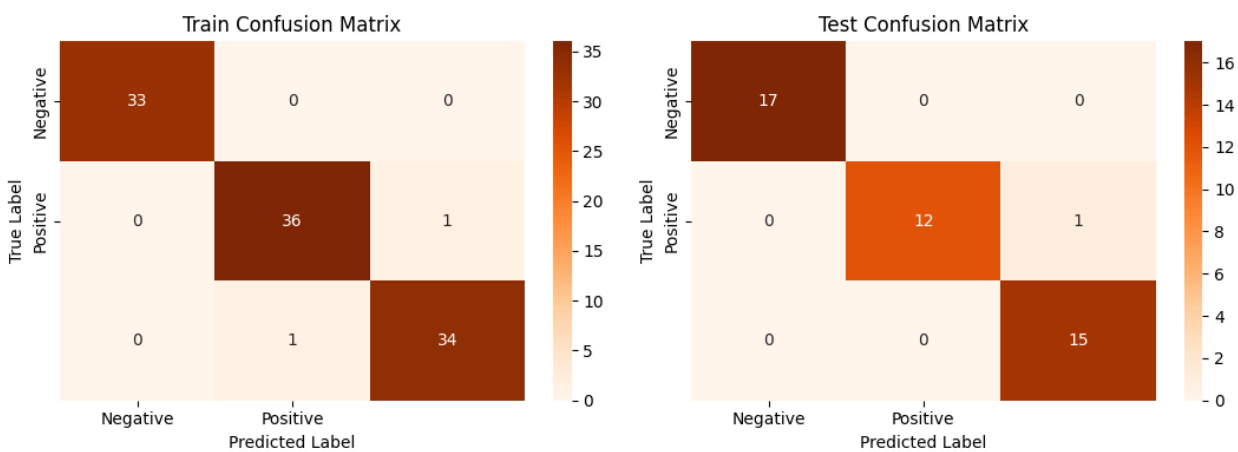
ML Model - 1 : Logistic regression

```
# ML Model - 1 Implementation
lr_model = LogisticRegression(fit_intercept=True, max_iter=10000)

# Model is trained (fit) and predicted in the evaluate model
```

1. Explain the ML Model used and it's performance using Evaluation metric Score Chart.

```
# Visualizing evaluation Metric Score chart
lr_score = evaluate_model(lr_model, x_train, x_test, y_train, y_test)
```

```
Confusion Matrix:
```




```
Train Classification Report:
|              |   precision |   recall |   f1-score |     support |
|:-------------|------------:|---------:|-----------:|------------:|
| 0            |    1        | 1        |   1        |  33         |
| 1            |    0.972973 | 0.972973 |   0.972973 |  37         |
| 2            |    0.971429 | 0.971429 |   0.971429 |  35         |
| accuracy     |    0.980952 | 0.980952 |   0.980952 |   0.980952 |
| macro avg    |    0.981467 | 0.981467 |   0.981467 | 105         |
| weighted avg |    0.980952 | 0.980952 |   0.980952 | 105         |

Test Classification Report:
|              |   precision |   recall |   f1-score |    support |
|:-------------|------------:|---------:|-----------:|-----------:|
```

```
| 0            |     1        | 1        |    1        | 17        |
| 1            |     1        | 0.923077 |    0.96     | 13        |
| 2            |     0.9375   | 1        |    0.967742 | 15        |
| accuracy     |     0.977778 | 0.977778 |    0.977778 |  0.977778 |
| macro avg    |     0.979167 | 0.974359 |    0.975914 | 45        |
| weighted avg |     0.979167 | 0.977778 |    0.977692 | 45        |
```

```python
# Updated Evaluation metric Score Chart
score['Logistic regression'] = lr_score
score
```

```
                 Logistic regression
Precision Train             0.980952
Precision Test              0.979167
Recall Train                0.980952
Recall Test                 0.977778
Accuracy Train              0.980952
Accuracy Test               0.977778
F1 macro Train              0.980952
F1 macro Test               0.977692
```

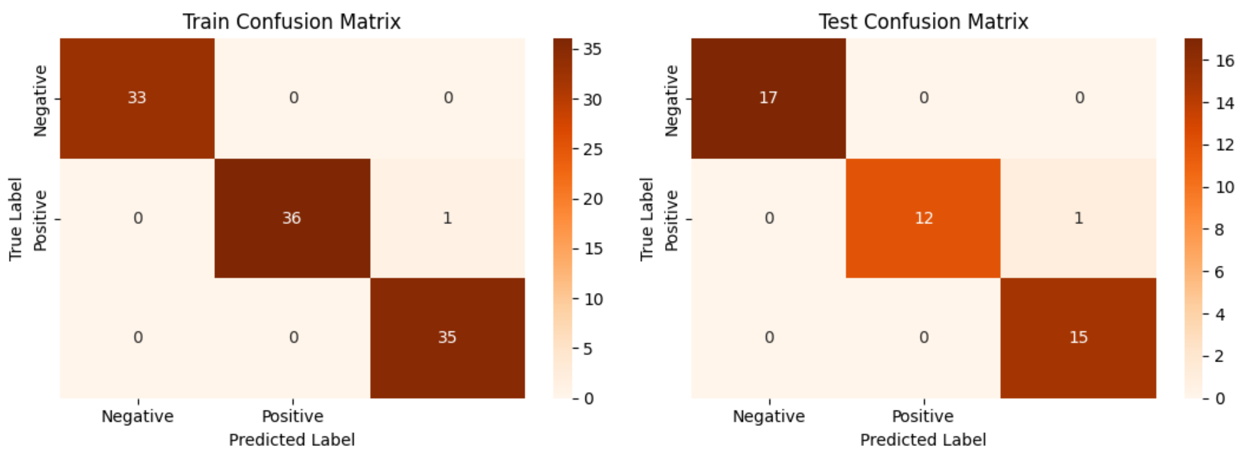
2. Cross- Validation & Hyperparameter Tuning

```python
# ML Model - 1 Implementation with hyperparameter optimization
techniques (i.e., GridSearch CV, RandomSearch CV, Bayesian
Optimization etc.)
# Define the hyperparameter grid
param_grid = {'C': [100,10,1,0.1,0.01,0.001,0.0001],
              'penalty': ['l1', 'l2'],
              'solver':['newton-cg', 'lbfgs', 'liblinear', 'sag',
'saga']}

# Initializing the logistic regression model
logreg = LogisticRegression(fit_intercept=True, max_iter=10000,
random_state=0)

# Repeated stratified kfold
rskf = RepeatedStratifiedKFold(n_splits=3, n_repeats=4,
random_state=0)

# Using GridSearchCV to tune the hyperparameters using cross-
validation
grid = GridSearchCV(logreg, param_grid, cv=rskf)
grid.fit(x_train, y_train)

# Select the best hyperparameters found by GridSearchCV
best_params = grid.best_params_
print("Best hyperparameters: ", best_params)
```

```
Best hyperparameters:  {'C': 10, 'penalty': 'l2', 'solver': 'saga'}
```

```python
# Initiate model with best parameters
lr_model2 = LogisticRegression(C=best_params['C'],
                                  penalty=best_params['penalty'],
                                  solver=best_params['solver'],
                                  max_iter=10000, random_state=0)

# Visualizing evaluation Metric Score chart
lr_score2 = evaluate_model(lr_model2, x_train, x_test, y_train,
y_test)
```

```
Confusion Matrix:
```




```
Train Classification Report:
|              |   precision |   recall |   f1-score |     support |
|:-------------|------------:|---------:|-----------:|------------:|
| 0            |    1        | 1        |   1        |  33         |
| 1            |    1        | 0.972973 |   0.986301 |  37         |
| 2            |    0.972222 | 1        |   0.985915 |  35         |
| accuracy     |    0.990476 | 0.990476 |   0.990476 |   0.990476  |
| macro avg    |    0.990741 | 0.990991 |   0.990739 | 105         |
| weighted avg |    0.990741 | 0.990476 |   0.990478 | 105         |

Test Classification Report:
|              |   precision |   recall |   f1-score |    support |
|:-------------|------------:|---------:|-----------:|-----------:|
| 0            |    1        | 1        |   1        | 17         |
| 1            |    1        | 0.923077 |   0.96     | 13         |
| 2            |    0.9375   | 1        |   0.967742 | 15         |
| accuracy     |    0.977778 | 0.977778 |   0.977778 |  0.977778  |
| macro avg    |    0.979167 | 0.974359 |   0.975914 | 45         |
| weighted avg |    0.979167 | 0.977778 |   0.977692 | 45         |
```

```python
score['Logistic regression tuned'] = lr_score2
```

The hyperparameter optimization technique used is GridSearchCV. GridSearchCV is a method that performs an exhaustive search over a specified parameter grid to find the best hyperparameters for a model. It is a popular method for hyperparameter tuning because it is simple to implement and can be effective in finding good hyperparameters for a model.

The choice of hyperparameter optimization technique depends on various factors such as the size of the parameter space, the computational resources available, and the time constraints. GridSearchCV can be a good choice when the parameter space is relatively small and computational resources are not a major concern.

```
# Updated Evaluation metric Score Chart
score
```

	Logistic regression	Logistic regression tuned
Precision Train	0.980952	0.990741
Precision Test	0.979167	0.979167
Recall Train	0.980952	0.990476
Recall Test	0.977778	0.977778
Accuracy Train	0.980952	0.990476
Accuracy Test	0.977778	0.977778
F1 macro Train	0.980952	0.990478
F1 macro Test	0.977692	0.977692

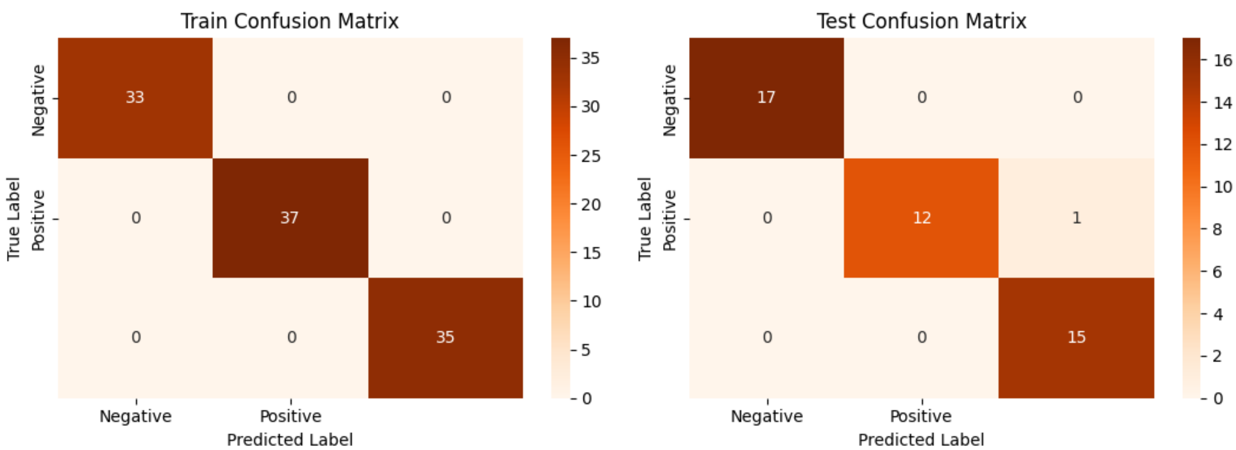
It appears that hyperparameter tuning did not improve the performance of the Logistic Regression model on the test set. The precision, recall, accuracy and F1 scores on the test set are same for both tuned and untuned Logistic Regression models.

ML Model - 2 : Decision Tree

```
# ML Model - 2 Implementation
dt_model = DecisionTreeClassifier(random_state=20)

# Model is trained (fit) and predicted in the evaluate model
```

1. Explain the ML Model used and it's performance using Evaluation metric Score Chart.

```
# Visualizing evaluation Metric Score chart
dt_score = evaluate_model(dt_model, x_train, x_test, y_train, y_test)


Confusion Matrix:
```




```
Train Classification Report:
|              |   precision |   recall |   f1-score |   support |
|:-------------|------------:|---------:|-----------:|----------:|
| 0            |           1 |        1 |          1 |        33 |
| 1            |           1 |        1 |          1 |        37 |
| 2            |           1 |        1 |          1 |        35 |
| accuracy     |           1 |        1 |          1 |         1 |
| macro avg    |           1 |        1 |          1 |       105 |
| weighted avg |           1 |        1 |          1 |       105 |

Test Classification Report:
|              |   precision |   recall |   f1-score |   support |
|:-------------|------------:|---------:|-----------:|----------:|
| 0            |    1        | 1        |   1        | 17        |
| 1            |    1        | 0.923077 |   0.96     | 13        |
| 2            |    0.9375   | 1        |   0.967742 | 15        |
| accuracy     |    0.977778 | 0.977778 |   0.977778 |  0.977778 |
| macro avg    |    0.979167 | 0.974359 |   0.975914 | 45        |
| weighted avg |    0.979167 | 0.977778 |   0.977692 | 45        |
```

```python
# Updated Evaluation metric Score Chart
score['Decision Tree'] = dt_score
score
```

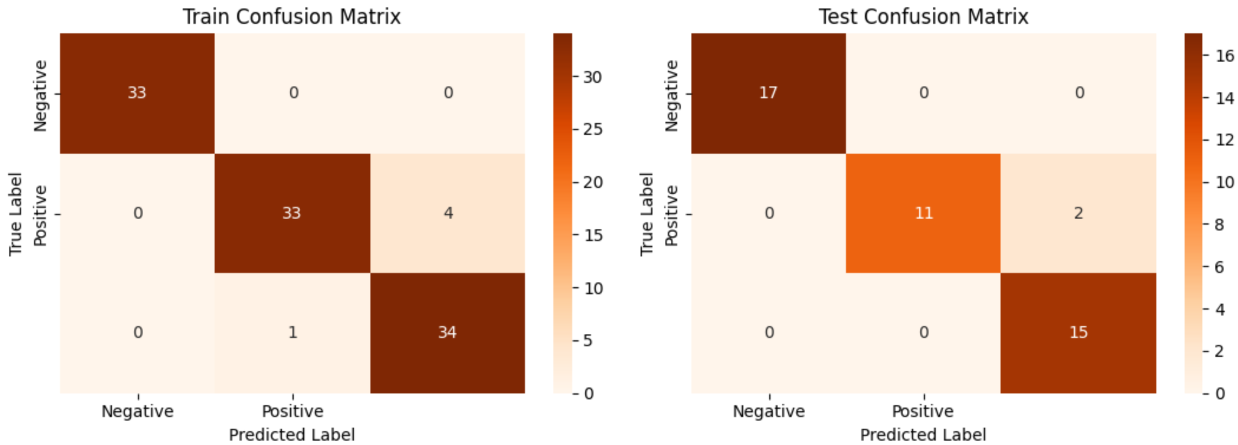
```
                 Logistic regression  Logistic regression tuned  
Decision Tree
Precision Train             0.980952                   0.990741   
1.000000
Precision Test              0.979167                   0.979167   
0.979167
Recall Train                0.980952                   0.990476   
1.000000
Recall Test                 0.977778                   0.977778   
0.977778
Accuracy Train              0.980952                   0.990476
```

```
1.000000
Accuracy Test              0.977778                  0.977778
0.977778
F1 macro Train             0.980952                  0.990478
1.000000
F1 macro Test              0.977692                  0.977692
0.977692
```

2. Cross- Validation & Hyperparameter Tuning

```
# ML Model - 2 Implementation with hyperparameter optimization
techniques (i.e., GridSearch CV, RandomSearch CV, Bayesian
Optimization etc.)
# Define the hyperparameter grid
grid = {'max_depth' : [3,4,5,6,7,8],
        'min_samples_split' : np.arange(2,8),
        'min_samples_leaf' : np.arange(10,20)}

# Initialize the model
model = DecisionTreeClassifier()

# repeated stratified kfold
rskf = RepeatedStratifiedKFold(n_splits=3, n_repeats=3,
random_state=0)

# Initialize GridSearchCV
grid_search = GridSearchCV(model, grid, cv=rskf)

# Fit the GridSearchCV to the training data
grid_search.fit(x_train, y_train)

# Select the best hyperparameters
best_params = grid_search.best_params_
print("Best hyperparameters: ", best_params)

Best hyperparameters:  {'max_depth': 3, 'min_samples_leaf': 10,
'min_samples_split': 5}

# Train a new model with the best hyperparameters
dt_model2 = DecisionTreeClassifier(max_depth=best_params['max_depth'],

min_samples_leaf=best_params['min_samples_leaf'],

min_samples_split=best_params['min_samples_split'],
                                 random_state=20)

# Visualizing evaluation Metric Score chart
dt2_score = evaluate_model(dt_model2, x_train, x_test, y_train,
y_test)
```

```
Confusion Matrix:
```



Train Classification Report:

	precision	recall	f1-score	support
0	1	1	1	33
1	0.970588	0.891892	0.929577	37
2	0.894737	0.971429	0.931507	35
accuracy	0.952381	0.952381	0.952381	0.952381
macro avg	0.955108	0.95444	0.953695	105
weighted avg	0.954548	0.952381	0.952353	105

Test Classification Report:

	precision	recall	f1-score	support
0	1	1	1	17
1	1	0.846154	0.916667	13
2	0.882353	1	0.9375	15
accuracy	0.955556	0.955556	0.955556	0.955556
macro avg	0.960784	0.948718	0.951389	45
weighted avg	0.960784	0.955556	0.955093	45

```
score['Decision Tree tuned'] = dt2_score
```

The hyperparameter optimization technique used is GridSearchCV. GridSearchCV is a method that performs an exhaustive search over a specified parameter grid to find the best hyperparameters for a model. It is a popular method for hyperparameter tuning because it is simple to implement and can be effective in finding good hyperparameters for a model.

The choice of hyperparameter optimization technique depends on various factors such as the size of the parameter space, the computational resources available, and the time constraints. GridSearchCV can be a good choice when the parameter space is relatively small and computational resources are not a major concern.

```
# Updated Evaluation metric Score Chart
score
```

```
                 Logistic regression  Logistic regression tuned  \
Precision Train             0.980952                   0.990741
Precision Test              0.979167                   0.979167
Recall Train                0.980952                   0.990476
Recall Test                 0.977778                   0.977778
Accuracy Train              0.980952                   0.990476
Accuracy Test               0.977778                   0.977778
F1 macro Train              0.980952                   0.990478
F1 macro Test               0.977692                   0.977692

                 Decision Tree  Decision Tree tuned
Precision Train       1.000000             0.954548
Precision Test        0.979167             0.960784
Recall Train          1.000000             0.952381
Recall Test           0.977778             0.955556
Accuracy Train        1.000000             0.952381
Accuracy Test         0.977778             0.955556
F1 macro Train        1.000000             0.952353
F1 macro Test         0.977692             0.955093
```

It appears that hyperparameter tuning didn't improved the performance of the Decision Tree model on the test set. The precision, recall, accuracy and F1 scores on the test set are less for the tuned Decision Tree model compare to the untuned Decision Tree model.

The tuned model is not overfitting like the untuned model.

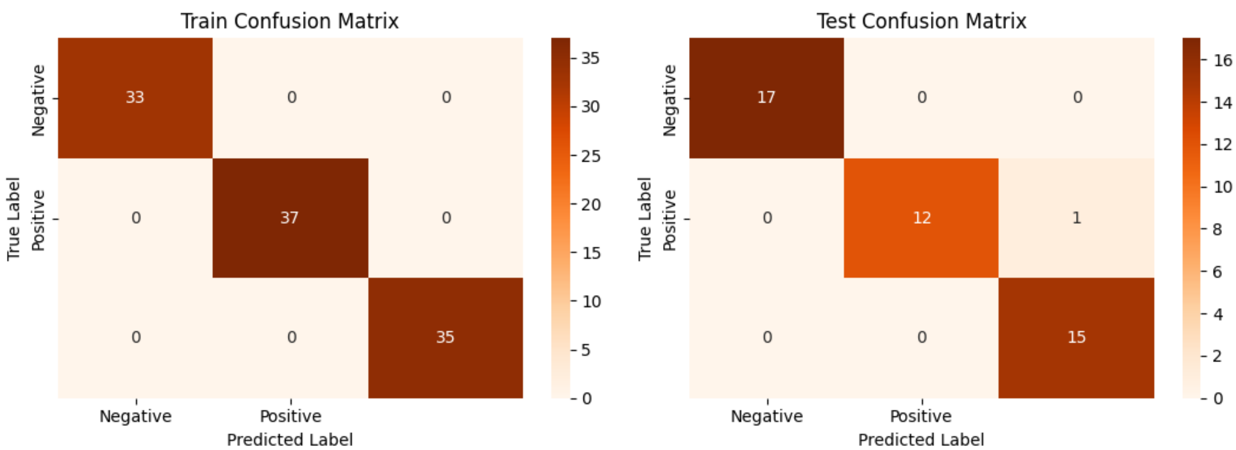
ML Model - 3 : Random Forest

```
# ML Model - 3 Implementation
rf_model = RandomForestClassifier(random_state=0)

# Model is trained (fit) and predicted in the evaluate model
```

1. Explain the ML Model used and it's performance using Evaluation metric Score Chart.

```
# Visualizing evaluation Metric Score chart
rf_score = evaluate_model(rf_model, x_train, x_test, y_train, y_test)
```

```
Confusion Matrix:
```




```
Train Classification Report:
|              |   precision |   recall |   f1-score |   support |
|:-------------|------------:|---------:|-----------:|----------:|
| 0            |           1 |        1 |          1 |        33 |
| 1            |           1 |        1 |          1 |        37 |
| 2            |           1 |        1 |          1 |        35 |
| accuracy     |           1 |        1 |          1 |         1 |
| macro avg    |           1 |        1 |          1 |       105 |
| weighted avg |           1 |        1 |          1 |       105 |

Test Classification Report:
|              |   precision |   recall |   f1-score |   support |
|:-------------|------------:|---------:|-----------:|----------:|
| 0            |    1        | 1        |   1        | 17        |
| 1            |    1        | 0.923077 |   0.96     | 13        |
| 2            |    0.9375   | 1        |   0.967742 | 15        |
| accuracy     |    0.977778 | 0.977778 |   0.977778 |  0.977778 |
| macro avg    |    0.979167 | 0.974359 |   0.975914 | 45        |
| weighted avg |    0.979167 | 0.977778 |   0.977692 | 45        |
```

```
# Updated Evaluation metric Score Chart
score['Random Forest'] = rf_score
score
```

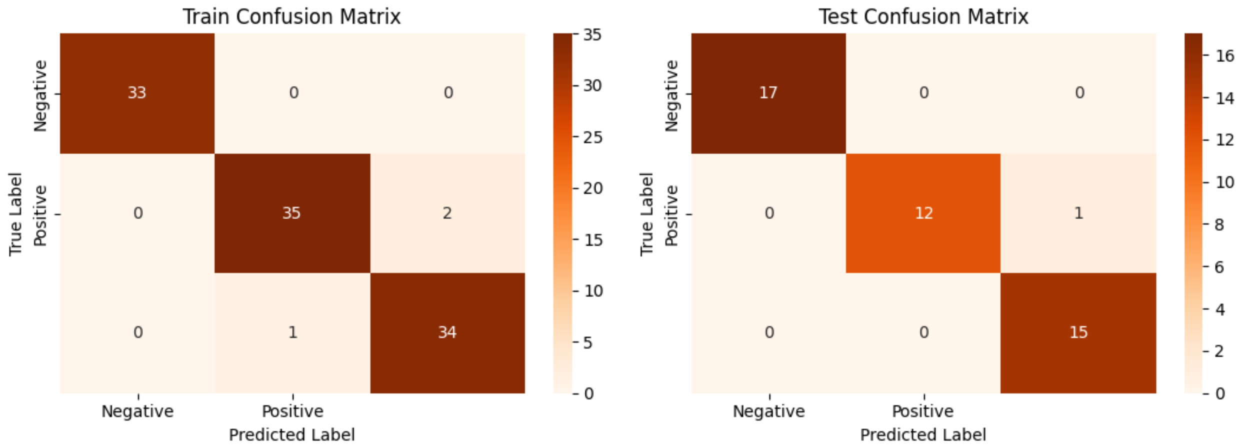
```
                 Logistic regression  Logistic regression tuned  \
Precision Train             0.980952                   0.990741
Precision Test              0.979167                   0.979167
Recall Train                0.980952                   0.990476
Recall Test                 0.977778                   0.977778
Accuracy Train              0.980952                   0.990476
Accuracy Test               0.977778                   0.977778
F1 macro Train              0.980952                   0.990478
F1 macro Test               0.977692                   0.977692

                 Decision Tree  Decision Tree tuned  Random Forest
```

```
Precision Train       1.000000              0.954548      1.000000
Precision Test        0.979167              0.960784      0.979167
Recall Train          1.000000              0.952381      1.000000
Recall Test           0.977778              0.955556      0.977778
Accuracy Train        1.000000              0.952381      1.000000
Accuracy Test         0.977778              0.955556      0.977778
F1 macro Train        1.000000              0.952353      1.000000
F1 macro Test         0.977692              0.955093      0.977692
```

2. Cross- Validation & Hyperparameter Tuning

```
# ML Model - 3 Implementation with hyperparameter optimization
techniques (i.e., GridSearch CV, RandomSearch CV, Bayesian
Optimization etc.)
# Define the hyperparameter grid
grid = {'n_estimators': [10, 50, 100, 200],
              'max_depth': [8, 9, 10, 11, 12,13, 14, 15],
              'min_samples_split': [2, 3, 4, 5]}

# Initialize the model
rf = RandomForestClassifier(random_state=0)

# Repeated stratified kfold
rskf = RepeatedStratifiedKFold(n_splits=3, n_repeats=3,
random_state=0)

# Initialize RandomSearchCV
random_search = RandomizedSearchCV(rf, grid,cv=rskf, n_iter=10,
n_jobs=-1)

# Fit the RandomSearchCV to the training data
random_search.fit(x_train, y_train)

# Select the best hyperparameters
best_params = random_search.best_params_
print("Best hyperparameters: ", best_params)

Best hyperparameters:  {'n_estimators': 100, 'min_samples_split': 4,
'max_depth': 12}

# Initialize model with best parameters
rf_model2 = RandomForestClassifier(n_estimators =
best_params['n_estimators'],
                                 min_samples_leaf=
best_params['min_samples_split'],
                                 max_depth = best_params['max_depth'],
                                 random_state=0)

# Visualizing evaluation Metric Score chart
rf2_score = evaluate_model(rf_model2, x_train, x_test, y_train,
y_test)
```

```
Confusion Matrix:
```




```
Train Classification Report:
|              |   precision |   recall |   f1-score |    support |
|:-------------|------------:|---------:|-----------:|-----------:|
| 0            |    1        | 1        |   1        |  33        |
| 1            |    0.972222 | 0.945946 |   0.958904 |  37        |
| 2            |    0.944444 | 0.971429 |   0.957746 |  35        |
| accuracy     |    0.971429 | 0.971429 |   0.971429 |   0.971429 |
| macro avg    |    0.972222 | 0.972458 |   0.972217 | 105        |
| weighted avg |    0.971693 | 0.971429 |   0.971434 | 105        |

Test Classification Report:
|              |   precision |   recall |   f1-score |   support |
|:-------------|------------:|---------:|-----------:|----------:|
| 0            |    1        | 1        |   1        | 17        |
| 1            |    1        | 0.923077 |   0.96     | 13        |
| 2            |    0.9375   | 1        |   0.967742 | 15        |
| accuracy     |    0.977778 | 0.977778 |   0.977778 |  0.977778 |
| macro avg    |    0.979167 | 0.974359 |   0.975914 | 45        |
| weighted avg |    0.979167 | 0.977778 |   0.977692 | 45        |

score['Random Forest tuned'] = rf2_score
```

The hyperparameter optimization technique i used is RandomizedSearchCV. RandomizedSearchCV is a method that performs a random search over a specified parameter grid to find the best hyperparameters for a model. It is a popular method for hyperparameter tuning because it can be more efficient than exhaustive search methods like GridSearchCV when the parameter space is large.

The choice of hyperparameter optimization technique depends on various factors such as the size of the parameter space, the computational resources available, and the time constraints.

RandomizedSearchCV can be a good choice when the parameter space is large and computational resources are limited.

```
# Updated Evaluation metric Score Chart
score
```

	Logistic regression	Logistic regression tuned
Precision Train	0.980952	0.990741
Precision Test	0.979167	0.979167
Recall Train	0.980952	0.990476
Recall Test	0.977778	0.977778
Accuracy Train	0.980952	0.990476
Accuracy Test	0.977778	0.977778
F1 macro Train	0.980952	0.990478
F1 macro Test	0.977692	0.977692

	Decision Tree	Decision Tree tuned	Random Forest
Precision Train	1.000000	0.954548	1.000000
Precision Test	0.979167	0.960784	0.979167
Recall Train	1.000000	0.952381	1.000000
Recall Test	0.977778	0.955556	0.977778
Accuracy Train	1.000000	0.952381	1.000000
Accuracy Test	0.977778	0.955556	0.977778
F1 macro Train	1.000000	0.952353	1.000000
F1 macro Test	0.977692	0.955093	0.977692

	Random Forest tuned
Precision Train	0.971693
Precision Test	0.979167
Recall Train	0.971429
Recall Test	0.977778
Accuracy Train	0.971429
Accuracy Test	0.977778
F1 macro Train	0.971434
F1 macro Test	0.977692

It appears that hyperparameter tuning improved the performance of the Random Forest model on the train set. But the precision, recall, accuracy and F1 scores on the test set are same for both tuned and untuned Random Forest models.

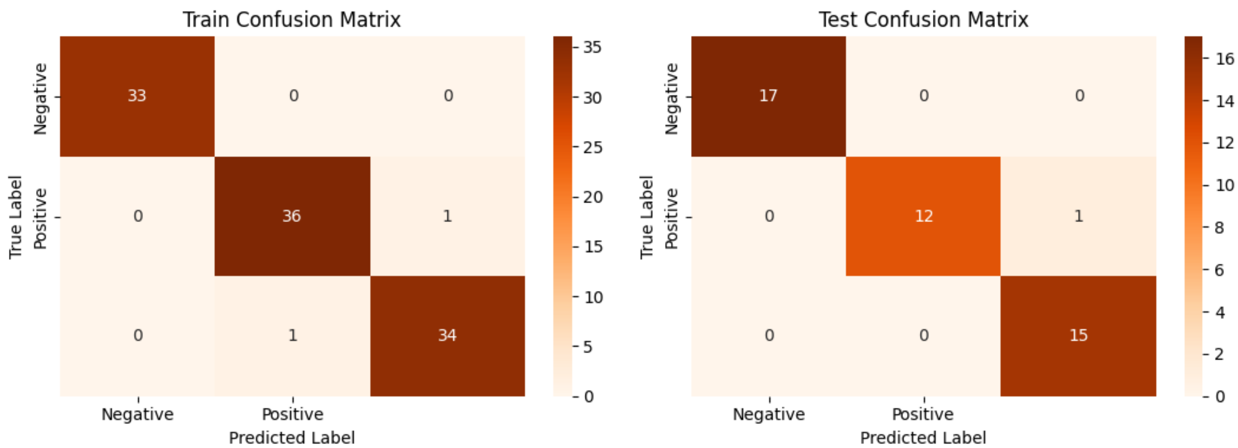
ML Model - 4 : SVM (Support Vector Machine)

```
# ML Model - 4 Implementation
svm_model = SVC(kernel='linear', random_state=0, probability=True)

# Model is trained (fit) and predicted in the evaluate model
```

1. Explain the ML Model used and it's performance using Evaluation metric Score Chart.

```
# Visualizing evaluation Metric Score chart
svm_score = evaluate_model(svm_model, x_train, x_test, y_train, y_test)
```

```
Confusion Matrix:
```



Train Classification Report:

	precision	recall	f1-score	support
0	1	1	1	33
1	0.972973	0.972973	0.972973	37
2	0.971429	0.971429	0.971429	35
accuracy	0.980952	0.980952	0.980952	0.980952
macro avg	0.981467	0.981467	0.981467	105
weighted avg	0.980952	0.980952	0.980952	105

Test Classification Report:

	precision	recall	f1-score	support
0	1	1	1	17
1	1	0.923077	0.96	13
2	0.9375	1	0.967742	15
accuracy	0.977778	0.977778	0.977778	0.977778
macro avg	0.979167	0.974359	0.975914	45
weighted avg	0.979167	0.977778	0.977692	45

```
# Updated Evaluation metric Score Chart
score['SVM'] = svm_score
score
```

	Logistic regression	Logistic regression tuned
Precision Train	0.980952	0.990741
Precision Test	0.979167	0.979167
Recall Train	0.980952	0.990476
Recall Test	0.977778	0.977778
Accuracy Train	0.980952	0.990476
Accuracy Test	0.977778	0.977778
F1 macro Train	0.980952	0.990478
F1 macro Test	0.977692	0.977692

	Decision Tree	Decision Tree tuned	Random Forest
Precision Train	1.000000	0.954548	1.000000
Precision Test	0.979167	0.960784	0.979167
Recall Train	1.000000	0.952381	1.000000
Recall Test	0.977778	0.955556	0.977778
Accuracy Train	1.000000	0.952381	1.000000
Accuracy Test	0.977778	0.955556	0.977778
F1 macro Train	1.000000	0.952353	1.000000
F1 macro Test	0.977692	0.955093	0.977692

	Random Forest tuned	SVM
Precision Train	0.971693	0.980952
Precision Test	0.979167	0.979167
Recall Train	0.971429	0.980952
Recall Test	0.977778	0.977778
Accuracy Train	0.971429	0.980952
Accuracy Test	0.977778	0.977778
F1 macro Train	0.971434	0.980952
F1 macro Test	0.977692	0.977692

2. Cross- Validation & Hyperparameter Tuning

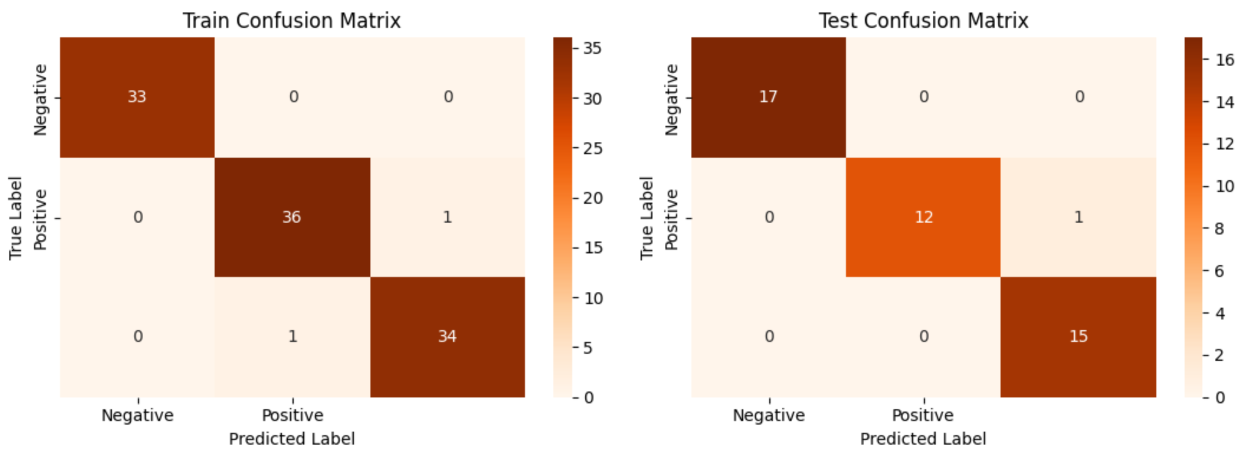
```
# ML Model - 4 Implementation with hyperparameter optimization
techniques (i.e., GridSearch CV, RandomSearch CV, Bayesian
Optimization etc.)
# Define the hyperparameter grid
param_grid = {'C': np.arange(0.1, 10, 0.1),
              'kernel': ['linear', 'poly', 'rbf', 'sigmoid'],
              'degree': np.arange(2, 6, 1)}

# Initialize the model
svm = SVC(random_state=0, probability=True)

# Repeated stratified kfold
rskf = RepeatedStratifiedKFold(n_splits=3, n_repeats=3,
random_state=0)

# Initialize RandomizedSearchCV with kfold cross-validation
random_search = RandomizedSearchCV(svm, param_grid, n_iter=10,
cv=rskf, n_jobs=-1)
```

```python
# Fit the RandomizedSearchCV to the training data
random_search.fit(x_train, y_train)

# Select the best hyperparameters
best_params = random_search.best_params_
print("Best hyperparameters: ", best_params)
```

```
Best hyperparameters:  {'kernel': 'rbf', 'degree': 5, 'C': 8.5}
```

```python
# Initialize model with best parameters
svm_model2 = SVC(C = best_params['C'],
           kernel = best_params['kernel'],
           degree = best_params['degree'],
           random_state=0, probability=True)

# Visualizing evaluation Metric Score chart
svm2_score = evaluate_model(svm_model2, x_train, x_test, y_train,
y_test)
```

```
Confusion Matrix:
```




```
Train Classification Report:
|              |   precision |   recall |   f1-score |    support |
|:-------------|------------:|---------:|-----------:|-----------:|
| 0            |    1        | 1        |   1        |  33        |
| 1            |    0.972973 | 0.972973 |   0.972973 |  37        |
| 2            |    0.971429 | 0.971429 |   0.971429 |  35        |
| accuracy     |    0.980952 | 0.980952 |   0.980952 |   0.980952 |
| macro avg    |    0.981467 | 0.981467 |   0.981467 | 105        |
| weighted avg |    0.980952 | 0.980952 |   0.980952 | 105        |

Test Classification Report:
|              |   precision |   recall |   f1-score |   support |
```

```
|:-------------|------------:|---------:|-----------:|----------:|
| 0            |    1        | 1        |   1        | 17        |
| 1            |    1        | 0.923077 |   0.96     | 13        |
| 2            |    0.9375   | 1        |   0.967742 | 15        |
| accuracy     |    0.977778 | 0.977778 |   0.977778 |  0.977778 |
| macro avg    |    0.979167 | 0.974359 |   0.975914 | 45        |
| weighted avg |    0.979167 | 0.977778 |   0.977692 | 45        |
```

```
score['SVM tuned'] = svm2_score
```

Here Randomized search is used as a hyperparameter optimization technique. Randomized search is a popular technique because it can be more efficient than exhaustive search methods like grid search. Instead of trying all possible combinations of hyperparameters, randomized search samples a random subset of the hyperparameter space. This can save time and computational resources while still finding good hyperparameters for the model.

```
# Updated Evaluation metric Score Chart
score
```

```
                 Logistic regression  Logistic regression tuned  \
Precision Train             0.980952                   0.990741
Precision Test              0.979167                   0.979167
Recall Train                0.980952                   0.990476
Recall Test                 0.977778                   0.977778
Accuracy Train              0.980952                   0.990476
Accuracy Test               0.977778                   0.977778
F1 macro Train              0.980952                   0.990478
F1 macro Test               0.977692                   0.977692

                 Decision Tree  Decision Tree tuned  Random Forest  \
Precision Train       1.000000             0.954548       1.000000
Precision Test        0.979167             0.960784       0.979167
Recall Train          1.000000             0.952381       1.000000
Recall Test           0.977778             0.955556       0.977778
Accuracy Train        1.000000             0.952381       1.000000
Accuracy Test         0.977778             0.955556       0.977778
F1 macro Train        1.000000             0.952353       1.000000
F1 macro Test         0.977692             0.955093       0.977692

                 Random Forest tuned       SVM  SVM tuned
Precision Train             0.971693  0.980952   0.980952
Precision Test              0.979167  0.979167   0.979167
Recall Train                0.971429  0.980952   0.980952
Recall Test                 0.977778  0.977778   0.977778
Accuracy Train              0.971429  0.980952   0.980952
Accuracy Test               0.977778  0.977778   0.977778
F1 macro Train              0.971434  0.980952   0.980952
F1 macro Test               0.977692  0.977692   0.977692
```

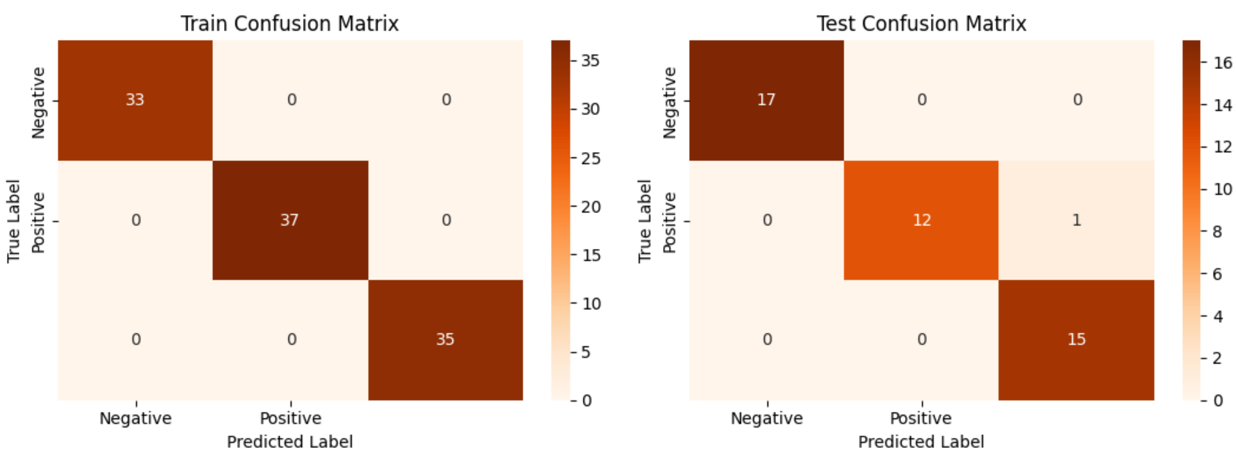
It appears that hyperparameter tuning did not improve the performance of the SVM model on the test set. The precision, recall, accuracy and F1 scores on the test set are same for both tuned and untuned SVM models.

ML Model - 5 : Xtreme Gradient Boosting

```
# ML Model - 5 Implementation
xgb_model = xgb.XGBClassifier()

# Model is trained (fit) and predicted in the evaluate model
```

1. Explain the ML Model used and it's performance using Evaluation metric Score Chart.

```
# Visualizing evaluation Metric Score chart
xgb_score = evaluate_model(xgb_model, x_train, x_test, y_train, y_test)


Confusion Matrix:
```




```
Train Classification Report:
|              |   precision |   recall |   f1-score |   support |
|:-------------|------------:|---------:|-----------:|----------:|
| 0            |           1 |        1 |          1 |        33 |
| 1            |           1 |        1 |          1 |        37 |
| 2            |           1 |        1 |          1 |        35 |
| accuracy     |           1 |        1 |          1 |         1 |
| macro avg    |           1 |        1 |          1 |       105 |
| weighted avg |           1 |        1 |          1 |       105 |

Test Classification Report:
|              |   precision |   recall |   f1-score |   support |
|:-------------|------------:|---------:|-----------:|----------:|
| 0            |    1        | 1        |   1        | 17        |
```

```
| 1            |     1        | 0.923077 |   0.96     | 13         |
| 2            |     0.9375   | 1        |   0.967742 | 15         |
| accuracy     |     0.977778 | 0.977778 |   0.977778 |  0.977778 |
| macro avg    |     0.979167 | 0.974359 |   0.975914 | 45         |
| weighted avg |     0.979167 | 0.977778 |   0.977692 | 45         |
```

```
# Updated Evaluation metric Score Chart
score['XGB'] = xgb_score
score
```

```
                 Logistic regression  Logistic regression tuned  \
Precision Train             0.980952                   0.990741
Precision Test              0.979167                   0.979167
Recall Train                0.980952                   0.990476
Recall Test                 0.977778                   0.977778
Accuracy Train              0.980952                   0.990476
Accuracy Test               0.977778                   0.977778
F1 macro Train              0.980952                   0.990478
F1 macro Test               0.977692                   0.977692

                 Decision Tree  Decision Tree tuned  Random Forest  \
Precision Train       1.000000             0.954548       1.000000
Precision Test        0.979167             0.960784       0.979167
Recall Train          1.000000             0.952381       1.000000
Recall Test           0.977778             0.955556       0.977778
Accuracy Train        1.000000             0.952381       1.000000
Accuracy Test         0.977778             0.955556       0.977778
F1 macro Train        1.000000             0.952353       1.000000
F1 macro Test         0.977692             0.955093       0.977692

                 Random Forest tuned       SVM  SVM tuned       XGB
Precision Train             0.971693  0.980952   0.980952  1.000000
Precision Test              0.979167  0.979167   0.979167  0.979167
Recall Train                0.971429  0.980952   0.980952  1.000000
Recall Test                 0.977778  0.977778   0.977778  0.977778
Accuracy Train              0.971429  0.980952   0.980952  1.000000
Accuracy Test               0.977778  0.977778   0.977778  0.977778
F1 macro Train              0.971434  0.980952   0.980952  1.000000
F1 macro Test               0.977692  0.977692   0.977692  0.977692
```

2. Cross- Validation & Hyperparameter Tuning

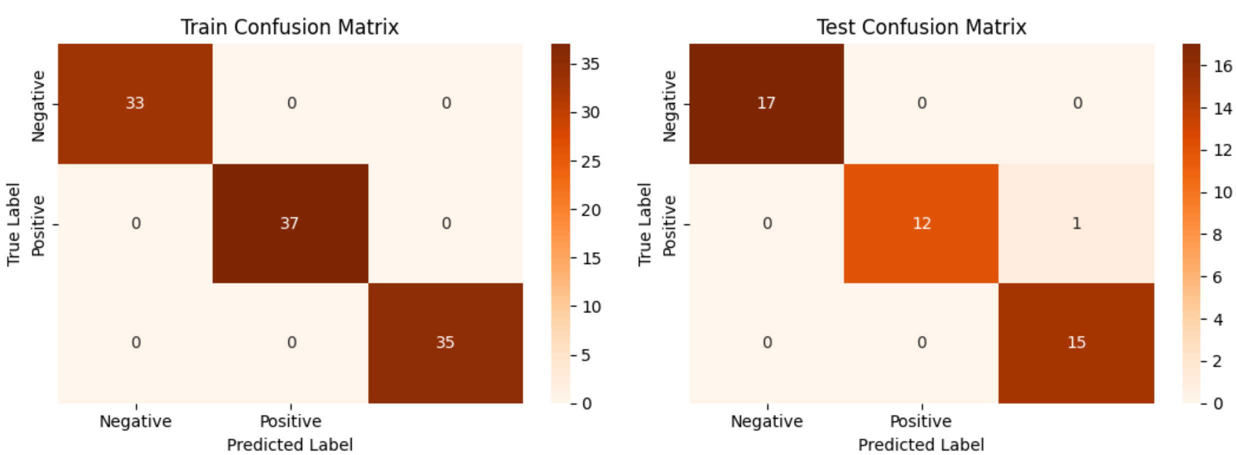
```
# ML Model - 5 Implementation with hyperparameter optimization
techniques (i.e., GridSearch CV, RandomSearch CV, Bayesian
Optimization etc.)
# Define the hyperparameter grid
param_grid = {'learning_rate': np.arange(0.01, 0.3, 0.01),
              'max_depth': np.arange(3, 15, 1),
              'n_estimators': np.arange(100, 200, 10)}
```

```
# Initialize the model
xgb2 = xgb.XGBClassifier(random_state=0)

# Repeated stratified kfold
rskf = RepeatedStratifiedKFold(n_splits=3, n_repeats=3,
random_state=0)

# Initialize RandomizedSearchCV
random_search = RandomizedSearchCV(xgb2, param_grid, n_iter=10,
cv=rskf)

# Fit the RandomizedSearchCV to the training data
random_search.fit(x_train, y_train)

# Select the best hyperparameters
best_params = random_search.best_params_
print("Best hyperparameters: ", best_params)
```

```
Best hyperparameters:  {'n_estimators': 170, 'max_depth': 12,
'learning_rate': 0.25}
```

```
# Initialize model with best parameters
xgb_model2 = xgb.XGBClassifier(learning_rate =
best_params['learning_rate'],
                                 max_depth = best_params['max_depth'],
                               n_estimators =
best_params['n_estimators'],
                                 random_state=0)

# Visualizing evaluation Metric Score chart
xgb2_score = evaluate_model(xgb_model2, x_train, x_test, y_train,
y_test)
```

```
Confusion Matrix:
```



```
Train Classification Report:
|              |   precision |   recall |   f1-score |   support |
|:-------------|------------:|---------:|-----------:|----------:|
| 0            |           1 |        1 |          1 |        33 |
| 1            |           1 |        1 |          1 |        37 |
| 2            |           1 |        1 |          1 |        35 |
| accuracy     |           1 |        1 |          1 |         1 |
| macro avg    |           1 |        1 |          1 |       105 |
| weighted avg |           1 |        1 |          1 |       105 |

Test Classification Report:
|              |   precision |   recall |   f1-score |   support |
|:-------------|------------:|---------:|-----------:|----------:|
| 0            |    1        | 1        |   1        | 17        |
| 1            |    1        | 0.923077 |   0.96     | 13        |
| 2            |    0.9375   | 1        |   0.967742 | 15        |
| accuracy     |    0.977778 | 0.977778 |   0.977778 |  0.977778 |
| macro avg    |    0.979167 | 0.974359 |   0.975914 | 45        |
| weighted avg |    0.979167 | 0.977778 |   0.977692 | 45        |
```

```
score['XGB tuned'] = xgb2_score
```

Here we have used Randomized search to tune the XGB model.

Randomized search is a popular technique because it can be more efficient than exhaustive search methods like grid search. Instead of trying all possible combinations of hyperparameters, randomized search samples a random subset of the hyperparameter space. This can save time and computational resources while still finding good hyperparameters for the model.

```
# Updated Evaluation metric Score Chart
score
```

```
                 Logistic regression  Logistic regression tuned  \
Precision Train             0.980952                   0.990741
Precision Test              0.979167                   0.979167
Recall Train                0.980952                   0.990476
Recall Test                 0.977778                   0.977778
Accuracy Train              0.980952                   0.990476
Accuracy Test               0.977778                   0.977778
F1 macro Train              0.980952                   0.990478
F1 macro Test               0.977692                   0.977692

                 Decision Tree  Decision Tree tuned  Random Forest  \
Precision Train       1.000000             0.954548       1.000000
Precision Test        0.979167             0.960784       0.979167
Recall Train          1.000000             0.952381       1.000000
Recall Test           0.977778             0.955556       0.977778
Accuracy Train        1.000000             0.952381       1.000000
Accuracy Test         0.977778             0.955556       0.977778
```

```
F1 macro Train        1.000000             0.952353      1.000000
F1 macro Test         0.977692             0.955093      0.977692

                 Random Forest tuned       SVM  SVM tuned       XGB
XGB tuned
Precision Train             0.971693  0.980952   0.980952  1.000000
1.000000
Precision Test              0.979167  0.979167   0.979167  0.979167
0.979167
Recall Train                0.971429  0.980952   0.980952  1.000000
1.000000
Recall Test                 0.977778  0.977778   0.977778  0.977778
0.977778
Accuracy Train              0.971429  0.980952   0.980952  1.000000
1.000000
Accuracy Test               0.977778  0.977778   0.977778  0.977778
0.977778
F1 macro Train              0.971434  0.980952   0.980952  1.000000
1.000000
F1 macro Test               0.977692  0.977692   0.977692  0.977692
0.977692
```

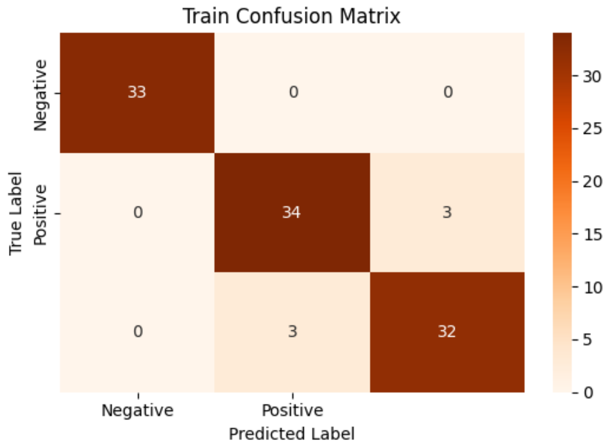
It appears that hyperparameter tuning did not improve the performance of the XGBoost model on the test set. The precision, recall, accuracy and F1 scores on the test set are same for both the untuned and tuned XGBoost models.

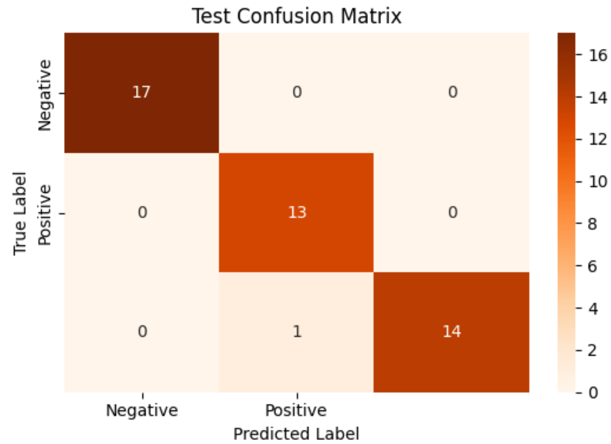
ML Model - 6 : Naive Bayes

```
# ML Model - 6 Implementation
nb_model = GaussianNB()

# Model is trained (fit) and predicted in the evaluate model
```

1. Explain the ML Model used and it's performance using Evaluation metric Score Chart.

```
# Visualizing evaluation Metric Score chart
nb_score = evaluate_model(nb_model, x_train, x_test, y_train, y_test)


Confusion Matrix:
```






```
Train Classification Report:
|              |   precision |   recall |   f1-score |     support |
|:-------------|------------:|---------:|-----------:|------------:|
| 0            |    1        | 1        |   1        |  33         |
| 1            |    0.918919 | 0.918919 |   0.918919 |  37         |
| 2            |    0.914286 | 0.914286 |   0.914286 |  35         |
| accuracy     |    0.942857 | 0.942857 |   0.942857 |   0.942857 |
| macro avg    |    0.944402 | 0.944402 |   0.944402 | 105         |
| weighted avg |    0.942857 | 0.942857 |   0.942857 | 105         |

Test Classification Report:
|              |   precision |   recall |   f1-score |   support |
|:-------------|------------:|---------:|-----------:|----------:|
| 0            |    1        | 1        |   1        | 17        |
| 1            |    0.928571 | 1        |   0.962963 | 13        |
| 2            |    1        | 0.933333 |   0.965517 | 15        |
| accuracy     |    0.977778 | 0.977778 |   0.977778 |  0.977778 |
| macro avg    |    0.97619  | 0.977778 |   0.97616  | 45        |
| weighted avg |    0.979365 | 0.977778 |   0.977806 | 45        |
```

```python
# Updated Evaluation metric Score Chart
score['Naive Bayes'] = nb_score
score
```

```
                 Logistic regression  Logistic regression tuned  \
Precision Train             0.980952                   0.990741
Precision Test              0.979167                   0.979167
Recall Train                0.980952                   0.990476
Recall Test                 0.977778                   0.977778
Accuracy Train              0.980952                   0.990476
Accuracy Test               0.977778                   0.977778
F1 macro Train              0.980952                   0.990478
F1 macro Test               0.977692                   0.977692

                 Decision Tree  Decision Tree tuned  Random Forest  \
```

```
Precision Train         1.000000                 0.954548       1.000000
Precision Test          0.979167                 0.960784       0.979167
Recall Train            1.000000                 0.952381       1.000000
Recall Test             0.977778                 0.955556       0.977778
Accuracy Train          1.000000                 0.952381       1.000000
Accuracy Test           0.977778                 0.955556       0.977778
F1 macro Train          1.000000                 0.952353       1.000000
F1 macro Test           0.977692                 0.955093       0.977692

                 Random Forest tuned       SVM  SVM tuned       XGB  \
Precision Train             0.971693  0.980952   0.980952  1.000000
Precision Test              0.979167  0.979167   0.979167  0.979167
Recall Train                0.971429  0.980952   0.980952  1.000000
Recall Test                 0.977778  0.977778   0.977778  0.977778
Accuracy Train              0.971429  0.980952   0.980952  1.000000
Accuracy Test               0.977778  0.977778   0.977778  0.977778
F1 macro Train              0.971434  0.980952   0.980952  1.000000
F1 macro Test               0.977692  0.977692   0.977692  0.977692

                 XGB tuned  Naive Bayes
Precision Train   1.000000     0.942857
Precision Test    0.979167     0.979365
Recall Train      1.000000     0.942857
Recall Test       0.977778     0.977778
Accuracy Train    1.000000     0.942857
Accuracy Test     0.977778     0.977778
F1 macro Train    1.000000     0.942857
F1 macro Test     0.977692     0.977806
```

2. Cross- Validation & Hyperparameter Tuning

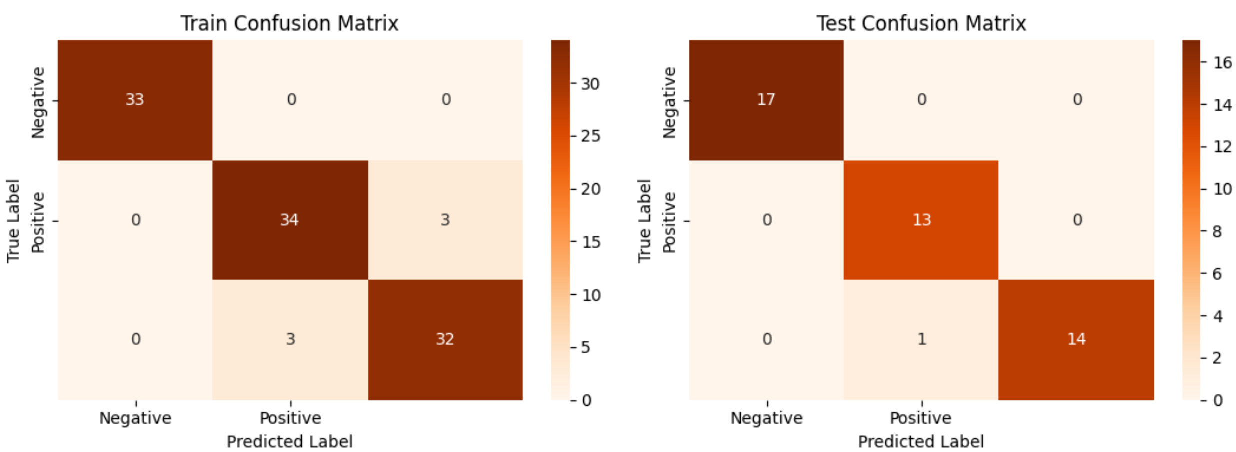
```
# ML Model - 6 Implementation with hyperparameter optimization
techniques (i.e., GridSearch CV, RandomSearch CV, Bayesian
Optimization etc.)
# Define the hyperparameter grid
param_grid = {'var_smoothing': np.logspace(0,-9, num=100)}

# Initialize the model
naive = GaussianNB()

# repeated stratified kfold
rskf = RepeatedStratifiedKFold(n_splits=4, n_repeats=4,
random_state=0)

# Initialize GridSearchCV
GridSearch = GridSearchCV(naive, param_grid, cv=rskf, n_jobs=-1)

# Fit the GridSearchCV to the training data
GridSearch.fit(x_train, y_train)
```

```python
# Select the best hyperparameters
best_params = GridSearch.best_params_
print("Best hyperparameters: ", best_params)
```

```
Best hyperparameters:  {'var_smoothing': 0.0001519911082952933}
```

```python
# Initiate model with best parameters
nb_model2 = GaussianNB(var_smoothing = best_params['var_smoothing'])

# Visualizing evaluation Metric Score chart
nb2_score = evaluate_model(nb_model2, x_train, x_test, y_train,
y_test)
```

```
Confusion Matrix:
```




```
Train Classification Report:
|              |   precision |   recall |   f1-score |    support |
|:-------------|------------:|---------:|-----------:|-----------:|
| 0            |    1        | 1        |   1        |  33        |
| 1            |    0.918919 | 0.918919 |   0.918919 |  37        |
| 2            |    0.914286 | 0.914286 |   0.914286 |  35        |
| accuracy     |    0.942857 | 0.942857 |   0.942857 |   0.942857 |
| macro avg    |    0.944402 | 0.944402 |   0.944402 | 105        |
| weighted avg |    0.942857 | 0.942857 |   0.942857 | 105        |

Test Classification Report:
|              |   precision |   recall |   f1-score |   support |
|:-------------|------------:|---------:|-----------:|----------:|
| 0            |    1        | 1        |   1        | 17        |
| 1            |    0.928571 | 1        |   0.962963 | 13        |
| 2            |    1        | 0.933333 |   0.965517 | 15        |
| accuracy     |    0.977778 | 0.977778 |   0.977778 |  0.977778 |
| macro avg    |    0.97619  | 0.977778 |   0.97616  | 45        |
| weighted avg |    0.979365 | 0.977778 |   0.977806 | 45        |
```

```python
score['Naive Bayes tuned']= nb2_score
```

Here we have used the GridSearchCV for optimization of the Naive Bayes model.

GridSearchCV is an exhaustive search method that tries all possible combinations of hyperparameters specified in the hyperparameter grid. This technique can be useful when the number of hyperparameters to tune is small and the range of possible values for each hyperparameter is limited. GridSearchCV can find the best combination of hyperparameters, but it can be computationally expensive for large hyperparameter grids.

```python
# Updated Evaluation metric Score Chart
score
```

	Logistic regression	Logistic regression tuned
Precision Train	0.980952	0.990741
Precision Test	0.979167	0.979167
Recall Train	0.980952	0.990476
Recall Test	0.977778	0.977778
Accuracy Train	0.980952	0.990476
Accuracy Test	0.977778	0.977778
F1 macro Train	0.980952	0.990478
F1 macro Test	0.977692	0.977692

	Decision Tree	Decision Tree tuned	Random Forest
Precision Train	1.000000	0.954548	1.000000
Precision Test	0.979167	0.960784	0.979167
Recall Train	1.000000	0.952381	1.000000
Recall Test	0.977778	0.955556	0.977778
Accuracy Train	1.000000	0.952381	1.000000
Accuracy Test	0.977778	0.955556	0.977778
F1 macro Train	1.000000	0.952353	1.000000
F1 macro Test	0.977692	0.955093	0.977692

	Random Forest tuned	SVM	SVM tuned	XGB
Precision Train	0.971693	0.980952	0.980952	1.000000
Precision Test	0.979167	0.979167	0.979167	0.979167
Recall Train	0.971429	0.980952	0.980952	1.000000
Recall Test	0.977778	0.977778	0.977778	0.977778
Accuracy Train	0.971429	0.980952	0.980952	1.000000
Accuracy Test	0.977778	0.977778	0.977778	0.977778
F1 macro Train	0.971434	0.980952	0.980952	1.000000
F1 macro Test	0.977692	0.977692	0.977692	0.977692

	XGB tuned	Naive Bayes	Naive Bayes tuned
Precision Train	1.000000	0.942857	0.942857
Precision Test	0.979167	0.979365	0.979365
Recall Train	1.000000	0.942857	0.942857
Recall Test	0.977778	0.977778	0.977778
Accuracy Train	1.000000	0.942857	0.942857
Accuracy Test	0.977778	0.977778	0.977778

```
F1 macro Train     1.000000     0.942857              0.942857
F1 macro Test      0.977692     0.977806              0.977806
```

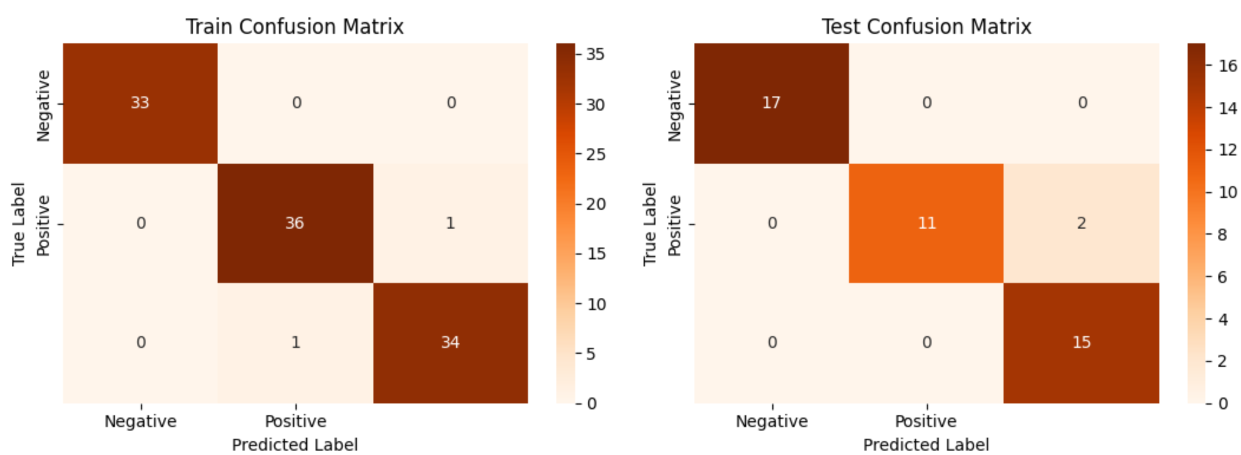
It appears that hyperparameter tuning did not improved the performance of the Naive Bayes model on the test set. The tuned Naive Bayes model has precision, recall, accuracy and F1 score on the test set as same as in the untuned Naive Bayes model.

ML Model - 7 : Neural Network

```
# ML Model - 7 Implementation
nn_model = MLPClassifier(random_state=0)

# Model is trained (fit) and predicted in the evaluate model
```

1. Explain the ML Model used and it's performance using Evaluation metric Score Chart.

```
# Visualizing evaluation Metric Score chart
neural_score = evaluate_model(nn_model, x_train, x_test, y_train,
y_test)


Confusion Matrix:
```




```
Train Classification Report:
|              |   precision |    recall |   f1-score |     support |
|:-------------|------------:|----------:|-----------:|------------:|
| 0            |    1        | 1         |   1        |  33         |
| 1            |    0.972973 | 0.972973  |   0.972973 |  37         |
| 2            |    0.971429 | 0.971429  |   0.971429 |  35         |
| accuracy     |    0.980952 | 0.980952  |   0.980952 |   0.980952  |
| macro avg    |    0.981467 | 0.981467  |   0.981467 | 105         |
| weighted avg |    0.980952 | 0.980952  |   0.980952 | 105         |
```

```
Test Classification Report:
|              |   precision |   recall |   f1-score |   support |
|:-------------|------------:|---------:|-----------:|----------:|
| 0            |    1        | 1        |   1        | 17        |
| 1            |    1        | 0.846154 |   0.916667 | 13        |
| 2            |    0.882353 | 1        |   0.9375   | 15        |
| accuracy     |    0.955556 | 0.955556 |   0.955556 |  0.955556 |
| macro avg    |    0.960784 | 0.948718 |   0.951389 | 45        |
| weighted avg |    0.960784 | 0.955556 |   0.955093 | 45        |
```

```python
# Updated Evaluation metric Score Chart
score['Neural Network'] = neural_score
score
```

```
                 Logistic regression  Logistic regression tuned  \
Precision Train             0.980952                   0.990741
Precision Test              0.979167                   0.979167
Recall Train                0.980952                   0.990476
Recall Test                 0.977778                   0.977778
Accuracy Train              0.980952                   0.990476
Accuracy Test               0.977778                   0.977778
F1 macro Train              0.980952                   0.990478
F1 macro Test               0.977692                   0.977692

                 Decision Tree  Decision Tree tuned  Random Forest  \
Precision Train       1.000000             0.954548       1.000000
Precision Test        0.979167             0.960784       0.979167
Recall Train          1.000000             0.952381       1.000000
Recall Test           0.977778             0.955556       0.977778
Accuracy Train        1.000000             0.952381       1.000000
Accuracy Test         0.977778             0.955556       0.977778
F1 macro Train        1.000000             0.952353       1.000000
F1 macro Test         0.977692             0.955093       0.977692

                 Random Forest tuned       SVM  SVM tuned       XGB  \
Precision Train             0.971693  0.980952   0.980952  1.000000
Precision Test              0.979167  0.979167   0.979167  0.979167
Recall Train                0.971429  0.980952   0.980952  1.000000
Recall Test                 0.977778  0.977778   0.977778  0.977778
Accuracy Train              0.971429  0.980952   0.980952  1.000000
Accuracy Test               0.977778  0.977778   0.977778  0.977778
F1 macro Train              0.971434  0.980952   0.980952  1.000000
F1 macro Test               0.977692  0.977692   0.977692  0.977692

                 XGB tuned  Naive Bayes  Naive Bayes tuned  Neural
Network
Precision Train   1.000000     0.942857           0.942857
0.980952
Precision Test    0.979167     0.979365           0.979365
0.960784
```

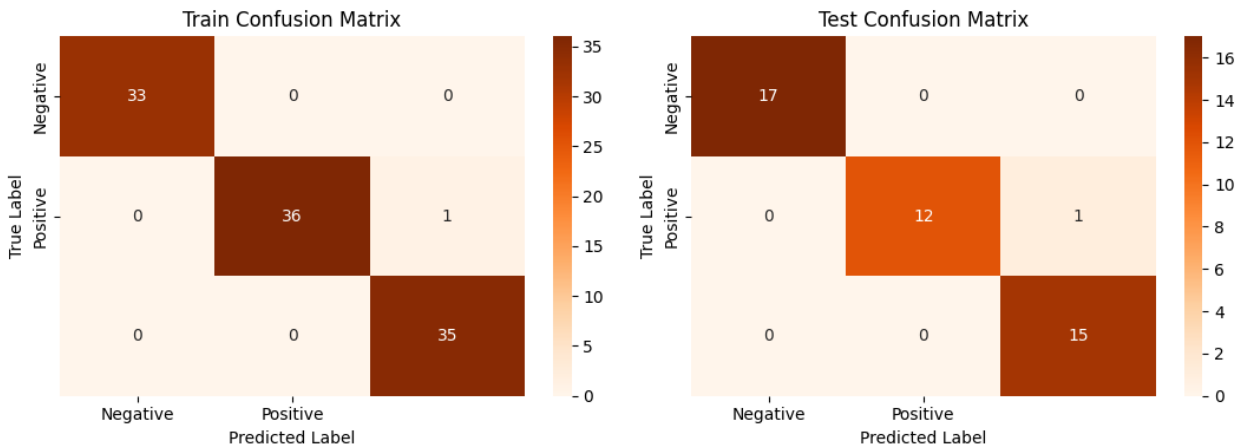
```
Recall Train       1.000000     0.942857          0.942857
0.980952
Recall Test        0.977778     0.977778          0.977778
0.955556
Accuracy Train     1.000000     0.942857          0.942857
0.980952
Accuracy Test      0.977778     0.977778          0.977778
0.955556
F1 macro Train     1.000000     0.942857          0.942857
0.980952
F1 macro Test      0.977692     0.977806          0.977806
0.955093
```

2. Cross- Validation & Hyperparameter Tuning

```
# ML Model - 7 Implementation with hyperparameter optimization
techniques (i.e., GridSearch CV, RandomSearch CV, Bayesian
Optimization etc.)
# Define the hyperparameter grid
param_grid = {'hidden_layer_sizes': np.arange(10, 100, 10),
              'alpha': np.arange(0.0001, 0.01, 0.0001)}

# Initialize the model
neural = MLPClassifier(random_state=0)

# Repeated stratified kfold
rskf = RepeatedStratifiedKFold(n_splits=3, n_repeats=3,
random_state=0)

# Initialize RandomizedSearchCV
random_search = RandomizedSearchCV(neural, param_grid, n_iter=10,
cv=rskf, n_jobs=-1)

# Fit the RandomizedSearchCV to the training data
random_search.fit(x_train, y_train)

# Select the best hyperparameters
best_params = random_search.best_params_
print("Best hyperparameters: ", best_params)

Best hyperparameters:  {'hidden_layer_sizes': 40, 'alpha':
0.006800000000000005}

# Initiate model with best parameters
nn_model2 = MLPClassifier(hidden_layer_sizes =
best_params['hidden_layer_sizes'],
                        alpha = best_params['alpha'],
                        random_state = 0)
```

```python
# Visualizing evaluation Metric Score chart
neural2_score = evaluate_model(nn_model2, x_train, x_test, y_train,
y_test)
```

```
Confusion Matrix:
```




```
Train Classification Report:
```

	precision	recall	f1-score	support
0	1	1	1	33
1	1	0.972973	0.986301	37
2	0.972222	1	0.985915	35
accuracy	0.990476	0.990476	0.990476	0.990476
macro avg	0.990741	0.990991	0.990739	105
weighted avg	0.990741	0.990476	0.990478	105

```
Test Classification Report:
```

	precision	recall	f1-score	support
0	1	1	1	17
1	1	0.923077	0.96	13
2	0.9375	1	0.967742	15
accuracy	0.977778	0.977778	0.977778	0.977778
macro avg	0.979167	0.974359	0.975914	45
weighted avg	0.979167	0.977778	0.977692	45

```python
score['Neural Network tuned']= neural2_score
```

Here we have used Randomized search to tune the Neural Network model.

Randomized search is a popular technique because it can be more efficient than exhaustive search methods like grid search. Instead of trying all possible combinations of hyperparameters, randomized search samples a random subset of the hyperparameter space. This can save time and computational resources while still finding good hyperparameters for the model.

```python
# Updated Evaluation metric Score Chart
score
```

```
                 Logistic regression  Logistic regression tuned  \
Precision Train             0.980952                   0.990741
Precision Test              0.979167                   0.979167
Recall Train                0.980952                   0.990476
Recall Test                 0.977778                   0.977778
Accuracy Train              0.980952                   0.990476
Accuracy Test               0.977778                   0.977778
F1 macro Train              0.980952                   0.990478
F1 macro Test               0.977692                   0.977692

                 Decision Tree  Decision Tree tuned  Random Forest  \
Precision Train       1.000000             0.954548       1.000000
Precision Test        0.979167             0.960784       0.979167
Recall Train          1.000000             0.952381       1.000000
Recall Test           0.977778             0.955556       0.977778
Accuracy Train        1.000000             0.952381       1.000000
Accuracy Test         0.977778             0.955556       0.977778
F1 macro Train        1.000000             0.952353       1.000000
F1 macro Test         0.977692             0.955093       0.977692

                 Random Forest tuned       SVM  SVM tuned       XGB  \
Precision Train             0.971693  0.980952   0.980952  1.000000
Precision Test              0.979167  0.979167   0.979167  0.979167
Recall Train                0.971429  0.980952   0.980952  1.000000
Recall Test                 0.977778  0.977778   0.977778  0.977778
Accuracy Train              0.971429  0.980952   0.980952  1.000000
Accuracy Test               0.977778  0.977778   0.977778  0.977778
F1 macro Train              0.971434  0.980952   0.980952  1.000000
F1 macro Test               0.977692  0.977692   0.977692  0.977692

                 XGB tuned  Naive Bayes  Naive Bayes tuned  Neural
Network  \
Precision Train   1.000000     0.942857           0.942857
0.980952
Precision Test    0.979167     0.979365           0.979365
0.960784
Recall Train      1.000000     0.942857           0.942857
0.980952
Recall Test       0.977778     0.977778           0.977778
0.955556
Accuracy Train    1.000000     0.942857           0.942857
0.980952
Accuracy Test     0.977778     0.977778           0.977778
0.955556
F1 macro Train    1.000000     0.942857           0.942857
0.980952
F1 macro Test     0.977692     0.977806           0.977806
```

```
0.955093

                 Neural Network tuned
Precision Train              0.990741
Precision Test               0.979167
Recall Train                 0.990476
Recall Test                  0.977778
Accuracy Train               0.990476
Accuracy Test                0.977778
F1 macro Train               0.990478
F1 macro Test                0.977692
```

It appears that hyperparameter tuning improve the performance of the neural network model on the test set. The precision, recall, accuracy and F1 scores on the test set are increased for the tuned neural network model compare to untuned neural network model.

```
print(score.to_markdown())
```

```
|                 |   Logistic regression |   Logistic regression
tuned |   Decision Tree |   Decision Tree tuned |   Random Forest |
Random Forest tuned |       SVM |   SVM tuned |       XGB |   XGB tuned
|   Naive Bayes |   Naive Bayes tuned |   Neural Network |   Neural
Network tuned |
|:----------------|----------------------:|---------------------------
-:|----------------:|----------------------:|----------------:|-------
---------------:|----------:|------------:|----------:|------------:|---
-----------:|--------------------:|-----------------:|----------------
-------:|
| Precision Train |              0.980952 |
0.990741 |        1        |              0.954548 |        1        |
0.971693 | 0.980952 |    0.980952 | 1        |    1        |
0.942857 |            0.942857 |         0.980952 |
0.990741 |
| Precision Test  |              0.979167 |
0.979167 |        0.979167 |              0.960784 |        0.979167 |
0.979167 | 0.979167 |    0.979167 | 0.979167 |    0.979167 |
0.979365 |            0.979365 |         0.960784 |
0.979167 |
| Recall Train    |              0.980952 |
0.990476 |        1        |              0.952381 |        1        |
0.971429 | 0.980952 |    0.980952 | 1        |    1        |
0.942857 |            0.942857 |         0.980952 |
0.990476 |
| Recall Test     |              0.977778 |
0.977778 |        0.977778 |              0.955556 |        0.977778 |
0.977778 | 0.977778 |    0.977778 | 0.977778 |    0.977778 |
0.977778 |            0.977778 |         0.955556 |
0.977778 |
| Accuracy Train  |              0.980952 |
```

```
0.990476 |        1        |              0.952381 |        1        |
0.971429 | 0.980952 |    0.980952 | 1        |    1        |
0.942857 |            0.942857 |         0.980952 |
0.990476 |
| Accuracy Test   |              0.977778 |
0.977778 |        0.977778 |              0.955556 |        0.977778 |
0.977778 | 0.977778 |    0.977778 | 0.977778 |    0.977778 |
0.977778 |            0.977778 |         0.955556 |
0.977778 |
| F1 macro Train  |              0.980952 |
0.990478 |        1        |              0.952353 |        1        |
0.971434 | 0.980952 |    0.980952 | 1        |    1        |
0.942857 |            0.942857 |         0.980952 |
0.990478 |
| F1 macro Test   |              0.977692 |
0.977692 |        0.977692 |              0.955093 |        0.977692 |
0.977692 | 0.977692 |    0.977692 | 0.977692 |    0.977692 |
0.977806 |            0.977806 |         0.955093 |
0.977692 |
```

Selection of best model

```
# Removing the overfitted models which have precision, recall, f1
scores for train as 1
score_t = score.transpose()            # taking transpose of the score
dataframe to create new difference column
remove_models = score_t[score_t['Recall Train']>=0.98].index  #
creating a list of models which have 1 for train and score_t['Accuracy
Train']==1.0 and score_t['Precision Train']==1.0 and score_t['F1 macro
Train']==1.0
remove_models

adj = score_t.drop(remove_models)                     # creating a new
dataframe with required models
adj

                     Precision Train  Precision Test  Recall Train  \
Decision Tree tuned         0.954548        0.960784      0.952381
Random Forest tuned         0.971693        0.979167      0.971429
Naive Bayes                 0.942857        0.979365      0.942857
Naive Bayes tuned           0.942857        0.979365      0.942857

                     Recall Test  Accuracy Train  Accuracy Test  \
Decision Tree tuned     0.955556        0.952381       0.955556
Random Forest tuned     0.977778        0.971429       0.977778
Naive Bayes             0.977778        0.942857       0.977778
Naive Bayes tuned       0.977778        0.942857       0.977778

                     F1 macro Train  F1 macro Test
Decision Tree tuned        0.952353       0.955093
```

```
Random Forest tuned          0.971434       0.977692
Naive Bayes                  0.942857       0.977806
Naive Bayes tuned            0.942857       0.977806
```

```
def select_best_model(df, metrics):

    best_models = {}
    for metric in metrics:
        max_test = df[metric + ' Test'].max()
        best_model_test = df[df[metric + ' Test'] ==
max_test].index[0]
        best_model = best_model_test
        best_models[metric] = best_model
    return best_models

metrics = ['Precision', 'Recall', 'Accuracy', 'F1 macro']

best_models = select_best_model(adj, metrics)
print("The best models are:")
for metric, best_model in best_models.items():
    print(f"{metric}: {best_model} - {adj[metric+' Test']
[best_model].round(4)}")
```

```
The best models are:
Precision: Naive Bayes - 0.9794
Recall: Random Forest tuned - 0.9778
Accuracy: Random Forest tuned - 0.9778
F1 macro: Naive Bayes - 0.9778
```

```
# Take recall as the primary evaluation metric
score_smpl = score.transpose()
remove_overfitting_models = score_smpl[score_smpl['Recall
Train']>=0.98].index
remove_overfitting_models
new_score = score_smpl.drop(remove_overfitting_models)
new_score = new_score.drop(['Precision Train','Precision
Test','Accuracy Train','Accuracy Test','F1 macro Train','F1 macro
Test'], axis=1)
new_score.index.name = 'Classification Model'
print(new_score.to_markdown())
```

```
| Classification Model   |   Recall Train |   Recall Test |
|:-----------------------|---------------:|--------------:|
| Decision Tree tuned    |       0.952381 |      0.955556 |
| Random Forest tuned    |       0.971429 |      0.977778 |
| Naive Bayes            |       0.942857 |      0.977778 |
| Naive Bayes tuned      |       0.942857 |      0.977778 |
```

After carefully considering the potential consequences of false positives and false negatives in the context of our business objectives, I have selected recall as the primary evaluation metric for our Iris flower classification model. This means that our goal is to maximize the number of true positives (correctly identified the different iris flowers) while minimizing the number of false negatives (incorrectly identified the flowers not a iris flower). By doing so, we aim to ensure that we correctly identify as many different iris flowers, even if it means that we may have some false positives.

The ML model i choose from the above created models as our final prediction model

After evaluating the performance of several machine learning models on the Iris dataset, I have selected the tuned Random Forest as our final prediction model. This decision was based on the model's performance on our primary evaluation metric of recall, which measures the ability of the model to correctly identify different iris flowers. In our analysis, we found that the Random Forest (tuned) had the highest recall score among the models we evaluated.

I choose recall as the primary evaluation metric because correctly identifying different iris flowers are critical to achieving our business objectives. By selecting a model with a high recall score, we aim to ensure that we correctly identify as many different iris flowers as possible, even if it means that we may have some false positives. Overall, we believe that the Random Forest (tuned) is the best choice for our needs and will help us achieve a positive business impact.

The model which i have used for the prediction

```python
# Define a list of category labels for reference.
Category_RF = ['Iris-Setosa', 'Iris-Versicolor', 'Iris-Virginica']

# In this example, it's a data point with Sepal Length, Sepal Width,
Petal Length, and Petal Width.
x_rf = np.array([[5.1, 3.5, 1.4, 0.2]])

# Use the tuned random forest model (rf_model2) to make a prediction.
x_rf_prediction = rf_model2.predict(x_rf)
x_rf_prediction[0]

# Display the predicted category label.
print(Category_RF[int(x_rf_prediction[0])])
```

```
Iris-Setosa
```

Conclusion

The project used a tuned Random Forest model to classify Iris flowers into three species — Setosa, Versicolor, and Virginica.

1. Data exploration helped identify key differences among the species, especially Iris-Setosa.

2. Data preprocessing was done to clean and prepare the dataset for modeling.

3. The Random Forest model performed well, showing good accuracy and reliability.

4. The project highlighted the importance of certain features in classifying the species.

5. Challenges included model fine-tuning and feature selection.

6. In the future, advanced models can be explored to improve accuracy further.

7. The model can be used in botany and horticulture to automatically identify Iris species.