**Project Name - Iris Flower Classification**

**Project Type - Classification**

**Contribution - Individual**

**Member Name – RAVI PRAKASH**

**Task - 1**

**Project Summary -**

**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.

**GitHub Link -**

**GitHub Link -https://github.com/raviprakashyadav244530t/Machine--learning--projects/upload/main**

**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.

***Let's Begin !***

***1. Know The Data***

**Import Libraries**

In [ ]:

*# 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**

In [ ]:

*# Load Dataset*

df = pd.read\_csv("https://raw.githubusercontent.com/Apaulgithub/oibsip\_task1/main/Iris.csv")

**Dataset First View**

In [ ]:

*# Dataset First Look*

*# View top 5 rows of the dataset*

df.head()

Out[ ]:

|  | **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**

In [ ]:

*# 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**

In [ ]:

*# 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**

In [ ]:

*# 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**

In [ ]:

*# Missing Values/Null Values Count*

df.isnull().sum()

Out[ ]:

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***

In [ ]:

*# Dataset Columns*

df.columns

Out[ ]:

Index(['Id', 'SepalLengthCm', 'SepalWidthCm', 'PetalLengthCm', 'PetalWidthCm',

'Species'],

dtype='object')

In [ ]:

*# Dataset Describe (all columns included)*

df.describe(include= 'all').round(2)

Out[ ]:

|  | **Id** | **SepalLengthCm** | **SepalWidthCm** | **PetalLengthCm** | **PetalWidthCm** | **Species** |
| --- | --- | --- | --- | --- | --- | --- |
| **count** | 150.00 | 150.00 | 150.00 | 150.00 | 150.00 | 150 |
| **unique** | NaN | NaN | NaN | NaN | NaN | 3 |
| **top** | NaN | NaN | NaN | NaN | NaN | Iris-setosa |
| **freq** | NaN | NaN | NaN | NaN | NaN | 50 |
| **mean** | 75.50 | 5.84 | 3.05 | 3.76 | 1.20 | NaN |
| **std** | 43.45 | 0.83 | 0.43 | 1.76 | 0.76 | NaN |
| **min** | 1.00 | 4.30 | 2.00 | 1.00 | 0.10 | NaN |
| **25%** | 38.25 | 5.10 | 2.80 | 1.60 | 0.30 | NaN |
| **50%** | 75.50 | 5.80 | 3.00 | 4.35 | 1.30 | NaN |
| **75%** | 112.75 | 6.40 | 3.30 | 5.10 | 1.80 | NaN |
| **max** | 150.00 | 7.90 | 4.40 | 6.90 | 2.50 | NaN |

**Check Unique Values for each variable.**

In [ ]:

*# 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**

In [ ]:

*# We don't need the 1st column so let's drop that*

data=df.iloc[:,1:]

In [ ]:

*# New updated dataset*

data.head()

Out[ ]:

|  | **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 |

**What all manipulations have i done?**

Only drop the first column of the dataset.

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

**Chart - 1 : Distribution of Numerical Variables**

In [ ]:

*# 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**

In [ ]:

*# Define colors for each species and the corresponding species labels.*

colors = ['red', 'yellow', 'green']

species = ['Iris-setosa', 'Iris-versicolor', 'Iris-virginica']

In [ ]:

*# 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**

In [ ]:

*# 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**

In [ ]:

*# 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**

In [ ]:

*# 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**

In [ ]:

*# 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**

In [ ]:

*# 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**

In [ ]:

*# Defining the X and y*

x=data.drop(columns=['Species'], axis=1)

y=data['Species']

**3. Data Splitting**

In [ ]:

*# Splitting the data to train and test*

x\_train,x\_test,y\_train,y\_test=train\_test\_split(x,y, test\_size=0.3)

In [ ]:

*# Checking the train distribution of dependent variable*

y\_train.value\_counts()

Out[ ]:

1 37

2 35

0 33

Name: Species, dtype: int64

***6. ML Model Implementation***

In [ ]:

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

In [ ]:

*# 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**

In [ ]:

*# 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.**

In [ ]:

*# 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 |

In [ ]:

*# Updated Evaluation metric Score Chart*

score['Logistic regression'] = lr\_score

score

Out[ ]:

|  | **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**

In [ ]:

*# 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'}

In [ ]:

*# 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)

In [ ]:

*# 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 |

In [ ]:

score['Logistic regression tuned'] = lr\_score2

**Which hyperparameter optimization technique have i used and why?**

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.

**Have i seen any improvement? Note down the improvement with updates Evaluation metric Score Chart.**

In [ ]:

*# Updated Evaluation metric Score Chart*

score

Out[ ]:

|  | **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**

In [ ]:

*# 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.**

In [ ]:

*# 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 |

In [ ]:

*# Updated Evaluation metric Score Chart*

score['Decision Tree'] = dt\_score

score

Out[ ]:

|  | **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**

In [ ]:

*# 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}

In [ ]:

*# 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)

In [ ]:

*# 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 |

In [ ]:

score['Decision Tree tuned'] = dt2\_score

##### Which hyperparameter optimization technique have i used and why?

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.

##### Have i seen any improvement? Note down the improvement with updates Evaluation metric Score Chart.

In [ ]:

*# Updated Evaluation metric Score Chart*

score

Out[ ]:

|  | **Logistic regression** | **Logistic regression tuned** | **Decision Tree** | **Decision Tree tuned** |
| --- | --- | --- | --- | --- |
| **Precision Train** | 0.980952 | 0.990741 | 1.000000 | 0.954548 |
| **Precision Test** | 0.979167 | 0.979167 | 0.979167 | 0.960784 |
| **Recall Train** | 0.980952 | 0.990476 | 1.000000 | 0.952381 |
| **Recall Test** | 0.977778 | 0.977778 | 0.977778 | 0.955556 |
| **Accuracy Train** | 0.980952 | 0.990476 | 1.000000 | 0.952381 |
| **Accuracy Test** | 0.977778 | 0.977778 | 0.977778 | 0.955556 |
| **F1 macro Train** | 0.980952 | 0.990478 | 1.000000 | 0.952353 |
| **F1 macro Test** | 0.977692 | 0.977692 | 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

In [ ]:

*# 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.

In [ ]:

*# 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:

| | prec

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 |

In [ ]:

*# Updated Evaluation metric Score Chart*

score['Random Forest'] = rf\_score

score

Out[ ]:

|  | **Logistic regression** | **Logistic regression tuned** | **Decision Tree** | **Decision Tree tuned** | **Random Forest** |
| --- | --- | --- | --- | --- | --- |
| **Precision Train** | 0.980952 | 0.990741 | 1.000000 | 0.954548 | 1.000000 |
| **Precision Test** | 0.979167 | 0.979167 | 0.979167 | 0.960784 | 0.979167 |
| **Recall Train** | 0.980952 | 0.990476 | 1.000000 | 0.952381 | 1.000000 |
| **Recall Test** | 0.977778 | 0.977778 | 0.977778 | 0.955556 | 0.977778 |
| **Accuracy Train** | 0.980952 | 0.990476 | 1.000000 | 0.952381 | 1.000000 |
| **Accuracy Test** | 0.977778 | 0.977778 | 0.977778 | 0.955556 | 0.977778 |
| **F1 macro Train** | 0.980952 | 0.990478 | 1.000000 | 0.952353 | 1.000000 |
| **F1 macro Test** | 0.977692 | 0.977692 | 0.977692 | 0.955093 | 0.977692 |

#### 2. Cross- Validation & Hyperparameter Tuning

In [ ]:

*# 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}

In [ ]:

*# 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)

In [ ]:

*# 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 |

In [ ]:

score['Random Forest tuned'] = rf2\_score

##### Which hyperparameter optimization technique have i used and why?

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.

##### Have i seen any improvement? Note down the improvement with updates Evaluation metric Score Chart.

In [ ]:

*# Updated Evaluation metric Score Chart*

score

Out[ ]:

|  | **Logistic regression** | **Logistic regression tuned** | **Decision Tree** | **Decision Tree tuned** | **Random Forest** | **Random Forest tuned** |
| --- | --- | --- | --- | --- | --- | --- |
| **Precision Train** | 0.980952 | 0.990741 | 1.000000 | 0.954548 | 1.000000 | 0.971693 |
| **Precision Test** | 0.979167 | 0.979167 | 0.979167 | 0.960784 | 0.979167 | 0.979167 |
| **Recall Train** | 0.980952 | 0.990476 | 1.000000 | 0.952381 | 1.000000 | 0.971429 |
| **Recall Test** | 0.977778 | 0.977778 | 0.977778 | 0.955556 | 0.977778 | 0.977778 |
| **Accuracy Train** | 0.980952 | 0.990476 | 1.000000 | 0.952381 | 1.000000 | 0.971429 |
| **Accuracy Test** | 0.977778 | 0.977778 | 0.977778 | 0.955556 | 0.977778 | 0.977778 |
| **F1 macro Train** | 0.980952 | 0.990478 | 1.000000 | 0.952353 | 1.000000 | 0.971434 |
| **F1 macro Test** | 0.977692 | 0.977692 | 0.977692 | 0.955093 | 0.977692 | 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)

In [ ]:

*# 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.

In [ ]:

*# 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 |

In [ ]:

*# Updated Evaluation metric Score Chart*

score['SVM'] = svm\_score

score

Out[ ]:

|  | **Logistic regression** | **Logistic regression tuned** | **Decision Tree** | **Decision Tree tuned** | **Random Forest** | **Random Forest tuned** | **SVM** |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Precision Train** | 0.980952 | 0.990741 | 1.000000 | 0.954548 | 1.000000 | 0.971693 | 0.980952 |
| **Precision Test** | 0.979167 | 0.979167 | 0.979167 | 0.960784 | 0.979167 | 0.979167 | 0.979167 |
| **Recall Train** | 0.980952 | 0.990476 | 1.000000 | 0.952381 | 1.000000 | 0.971429 | 0.980952 |
| **Recall Test** | 0.977778 | 0.977778 | 0.977778 | 0.955556 | 0.977778 | 0.977778 | 0.977778 |
| **Accuracy Train** | 0.980952 | 0.990476 | 1.000000 | 0.952381 | 1.000000 | 0.971429 | 0.980952 |
| **Accuracy Test** | 0.977778 | 0.977778 | 0.977778 | 0.955556 | 0.977778 | 0.977778 | 0.977778 |
| **F1 macro Train** | 0.980952 | 0.990478 | 1.000000 | 0.952353 | 1.000000 | 0.971434 | 0.980952 |
| **F1 macro Test** | 0.977692 | 0.977692 | 0.977692 | 0.955093 | 0.977692 | 0.977692 | 0.977692 |

#### 2. Cross- Validation & Hyperparameter Tuning

In [ ]:

*# 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)

*# 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}

In [ ]:

*# 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)

In [ ]:

*# 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 |

In [ ]:

score['SVM tuned'] = svm2\_score

##### Which hyperparameter optimization technique have i used and why?

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.

##### Have i seen any improvement? Note down the improvement with updates Evaluation metric Score Chart.

In [ ]:

*# Updated Evaluation metric Score Chart*

score

Out[ ]:

|  | **Logistic regression** | **Logistic regression tuned** | **Decision Tree** | **Decision Tree tuned** | **Random Forest** | **Random Forest tuned** | **SVM** | **SVM tuned** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Precision Train** | 0.980952 | 0.990741 | 1.000000 | 0.954548 | 1.000000 | 0.971693 | 0.980952 | 0.980952 |
| **Precision Test** | 0.979167 | 0.979167 | 0.979167 | 0.960784 | 0.979167 | 0.979167 | 0.979167 | 0.979167 |
| **Recall Train** | 0.980952 | 0.990476 | 1.000000 | 0.952381 | 1.000000 | 0.971429 | 0.980952 | 0.980952 |
| **Recall Test** | 0.977778 | 0.977778 | 0.977778 | 0.955556 | 0.977778 | 0.977778 | 0.977778 | 0.977778 |
| **Accuracy Train** | 0.980952 | 0.990476 | 1.000000 | 0.952381 | 1.000000 | 0.971429 | 0.980952 | 0.980952 |
| **Accuracy Test** | 0.977778 | 0.977778 | 0.977778 | 0.955556 | 0.977778 | 0.977778 | 0.977778 | 0.977778 |
| **F1 macro Train** | 0.980952 | 0.990478 | 1.000000 | 0.952353 | 1.000000 | 0.971434 | 0.980952 | 0.980952 |
| **F1 macro Test** | 0.977692 | 0.977692 | 0.977692 | 0.955093 | 0.977692 | 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

In [ ]:

*# 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.

In [ ]:

*# 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 |

In [ ]:

*# Updated Evaluation metric Score Chart*

score['XGB'] = xgb\_score

score

Out[ ]:

|  | **Logistic regression** | **Logistic regression tuned** | **Decision Tree** | **Decision Tree tuned** | **Random Forest** | **Random Forest tuned** | **SVM** | **SVM tuned** | **XGB** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Precision Train** | 0.980952 | 0.990741 | 1.000000 | 0.954548 | 1.000000 | 0.971693 | 0.980952 | 0.980952 | 1.000000 |
| **Precision Test** | 0.979167 | 0.979167 | 0.979167 | 0.960784 | 0.979167 | 0.979167 | 0.979167 | 0.979167 | 0.979167 |
| **Recall Train** | 0.980952 | 0.990476 | 1.000000 | 0.952381 | 1.000000 | 0.971429 | 0.980952 | 0.980952 | 1.000000 |
| **Recall Test** | 0.977778 | 0.977778 | 0.977778 | 0.955556 | 0.977778 | 0.977778 | 0.977778 | 0.977778 | 0.977778 |
| **Accuracy Train** | 0.980952 | 0.990476 | 1.000000 | 0.952381 | 1.000000 | 0.971429 | 0.980952 | 0.980952 | 1.000000 |
| **Accuracy Test** | 0.977778 | 0.977778 | 0.977778 | 0.955556 | 0.977778 | 0.977778 | 0.977778 | 0.977778 | 0.977778 |
| **F1 macro Train** | 0.980952 | 0.990478 | 1.000000 | 0.952353 | 1.000000 | 0.971434 | 0.980952 | 0.980952 | 1.000000 |
| **F1 macro Test** | 0.977692 | 0.977692 | 0.977692 | 0.955093 | 0.977692 | 0.977692 | 0.977692 | 0.977692 | 0.977692 |

#### 2. Cross- Validation & Hyperparameter Tuning

In [ ]:

*# 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}

In [ ]:

*# 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)

In [ ]:

*# 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 |

In [ ]:

score['XGB tuned'] = xgb2\_score

##### Which hyperparameter optimization technique have i used and why?

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.

##### Have i seen any improvement? Note down the improvement with updates Evaluation metric Score Chart.

In [ ]:

*# Updated Evaluation metric Score Chart*

score

Out[ ]:

|  | **Logistic regression** | **Logistic regression tuned** | **Decision Tree** | **Decision Tree tuned** | **Random Forest** | **Random Forest tuned** | **SVM** | **SVM tuned** | **XGB** | **XGB tuned** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Precision Train** | 0.980952 | 0.990741 | 1.000000 | 0.954548 | 1.000000 | 0.971693 | 0.980952 | 0.980952 | 1.000000 | 1.000000 |
| **Precision Test** | 0.979167 | 0.979167 | 0.979167 | 0.960784 | 0.979167 | 0.979167 | 0.979167 | 0.979167 | 0.979167 | 0.979167 |
| **Recall Train** | 0.980952 | 0.990476 | 1.000000 | 0.952381 | 1.000000 | 0.971429 | 0.980952 | 0.980952 | 1.000000 | 1.000000 |
| **Recall Test** | 0.977778 | 0.977778 | 0.977778 | 0.955556 | 0.977778 | 0.977778 | 0.977778 | 0.977778 | 0.977778 | 0.977778 |
| **Accuracy Train** | 0.980952 | 0.990476 | 1.000000 | 0.952381 | 1.000000 | 0.971429 | 0.980952 | 0.980952 | 1.000000 | 1.000000 |
| **Accuracy Test** | 0.977778 | 0.977778 | 0.977778 | 0.955556 | 0.977778 | 0.977778 | 0.977778 | 0.977778 | 0.977778 | 0.977778 |
| **F1 macro Train** | 0.980952 | 0.990478 | 1.000000 | 0.952353 | 1.000000 | 0.971434 | 0.980952 | 0.980952 | 1.000000 | 1.000000 |
| **F1 macro Test** | 0.977692 | 0.977692 | 0.977692 | 0.955093 | 0.977692 | 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**

In [ ]:

*# 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.**

In [ ]:

*# 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 |

In [ ]:

*# Updated Evaluation metric Score Chart*

score['Naive Bayes'] = nb\_score

score

Out[ ]:

|  | **Logistic regression** | **Logistic regression tuned** | **Decision Tree** | **Decision Tree tuned** | **Random Forest** | **Random Forest tuned** | **SVM** | **SVM tuned** | **XGB** | **XGB tuned** | **Naive Bayes** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Precision Train** | 0.980952 | 0.990741 | 1.000000 | 0.954548 | 1.000000 | 0.971693 | 0.980952 | 0.980952 | 1.000000 | 1.000000 | 0.942857 |
| **Precision Test** | 0.979167 | 0.979167 | 0.979167 | 0.960784 | 0.979167 | 0.979167 | 0.979167 | 0.979167 | 0.979167 | 0.979167 | 0.979365 |
| **Recall Train** | 0.980952 | 0.990476 | 1.000000 | 0.952381 | 1.000000 | 0.971429 | 0.980952 | 0.980952 | 1.000000 | 1.000000 | 0.942857 |
| **Recall Test** | 0.977778 | 0.977778 | 0.977778 | 0.955556 | 0.977778 | 0.977778 | 0.977778 | 0.977778 | 0.977778 | 0.977778 | 0.977778 |
| **Accuracy Train** | 0.980952 | 0.990476 | 1.000000 | 0.952381 | 1.000000 | 0.971429 | 0.980952 | 0.980952 | 1.000000 | 1.000000 | 0.942857 |
| **Accuracy Test** | 0.977778 | 0.977778 | 0.977778 | 0.955556 | 0.977778 | 0.977778 | 0.977778 | 0.977778 | 0.977778 | 0.977778 | 0.977778 |
| **F1 macro Train** | 0.980952 | 0.990478 | 1.000000 | 0.952353 | 1.000000 | 0.971434 | 0.980952 | 0.980952 | 1.000000 | 1.000000 | 0.942857 |
| **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 |

**2. Cross- Validation & Hyperparameter Tuning**

In [ ]:

*# 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)

*# Select the best hyperparameters*

best\_params = GridSearch.best\_params\_

print("Best hyperparameters: ", best\_params)

Best hyperparameters: {'var\_smoothing': 0.0001519911082952933}

In [ ]:

*# Initiate model with best parameters*

nb\_model2 = GaussianNB(var\_smoothing = best\_params['var\_smoothing'])

In [ ]:

*# 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 |

In [ ]:

score['Naive Bayes tuned']= nb2\_score

**Which hyperparameter optimization technique have i used and why?**

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.

**Have i seen any improvement? Note down the improvement with updates Evaluation metric Score Chart.**

In [ ]:

*# Updated Evaluation metric Score Chart*

score

Out[ ]:

|  | **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** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Precision Train** | 0.980952 | 0.990741 | 1.000000 | 0.954548 | 1.000000 | 0.971693 | 0.980952 | 0.980952 | 1.000000 | 1.000000 | 0.942857 | 0.942857 |
| **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 |
| **Recall Train** | 0.980952 | 0.990476 | 1.000000 | 0.952381 | 1.000000 | 0.971429 | 0.980952 | 0.980952 | 1.000000 | 1.000000 | 0.942857 | 0.942857 |
| **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 |
| **Accuracy Train** | 0.980952 | 0.990476 | 1.000000 | 0.952381 | 1.000000 | 0.971429 | 0.980952 | 0.980952 | 1.000000 | 1.000000 | 0.942857 | 0.942857 |
| **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 |
| **F1 macro Train** | 0.980952 | 0.990478 | 1.000000 | 0.952353 | 1.000000 | 0.971434 | 0.980952 | 0.980952 | 1.000000 | 1.000000 | 0.942857 | 0.942857 |
| **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 |

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**

In [ ]:

*# 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.**

In [ ]:

*# 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 |

In [ ]:

*# Updated Evaluation metric Score Chart*

score['Neural Network'] = neural\_score

score

Out[ ]:

|  | **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** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Precision Train** | 0.980952 | 0.990741 | 1.000000 | 0.954548 | 1.000000 | 0.971693 | 0.980952 | 0.980952 | 1.000000 | 1.000000 | 0.942857 | 0.942857 | 0.980952 |
| **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 |
| **Recall Train** | 0.980952 | 0.990476 | 1.000000 | 0.952381 | 1.000000 | 0.971429 | 0.980952 | 0.980952 | 1.000000 | 1.000000 | 0.942857 | 0.942857 | 0.980952 |
| **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 |
| **Accuracy Train** | 0.980952 | 0.990476 | 1.000000 | 0.952381 | 1.000000 | 0.971429 | 0.980952 | 0.980952 | 1.000000 | 1.000000 | 0.942857 | 0.942857 | 0.980952 |
| **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 |
| **F1 macro Train** | 0.980952 | 0.990478 | 1.000000 | 0.952353 | 1.000000 | 0.971434 | 0.980952 | 0.980952 | 1.000000 | 1.000000 | 0.942857 | 0.942857 | 0.980952 |
| **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 |

**2. Cross- Validation & Hyperparameter Tuning**

In [ ]:

*# 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': 0.0068000000000000005}

In [ ]:

*# Initiate model with best parameters*

nn\_model2 = MLPClassifier(hidden\_layer\_sizes = best\_params['hidden\_layer\_sizes'],

alpha = best\_params['alpha'],

random\_state = 0)

In [ ]:

*# 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 |

In [ ]:

score['Neural Network tuned']= neural2\_score

**Which hyperparameter optimization technique have i used and why?**

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.

**Have i seen any improvement? Note down the improvement with updates Evaluation metric Score Chart.**

In [ ]:

*# Updated Evaluation metric Score Chart*

score

Out[ ]:

|  | **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 | 1.000000 | 0.954548 | 1.000000 | 0.971693 | 0.980952 | 0.980952 | 1.000000 | 1.000000 | 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 | 1.000000 | 0.952381 | 1.000000 | 0.971429 | 0.980952 | 0.980952 | 1.000000 | 1.000000 | 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 | 1.000000 | 0.952381 | 1.000000 | 0.971429 | 0.980952 | 0.980952 | 1.000000 | 1.000000 | 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 | 1.000000 | 0.952353 | 1.000000 | 0.971434 | 0.980952 | 0.980952 | 1.000000 | 1.000000 | 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 |

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.

In [ ]:

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***

In [ ]:

*# 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

Out[ ]:

|  | **Precision Train** | **Precision Test** | **Recall Train** | **Recall Test** | **Accuracy Train** | **Accuracy Test** | **F1 macro Train** | **F1 macro Test** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Decision Tree tuned** | 0.954548 | 0.960784 | 0.952381 | 0.955556 | 0.952381 | 0.955556 | 0.952353 | 0.955093 |
| **Random Forest tuned** | 0.971693 | 0.979167 | 0.971429 | 0.977778 | 0.971429 | 0.977778 | 0.971434 | 0.977692 |
| **Naive Bayes** | 0.942857 | 0.979365 | 0.942857 | 0.977778 | 0.942857 | 0.977778 | 0.942857 | 0.977806 |
| **Naive Bayes tuned** | 0.942857 | 0.979365 | 0.942857 | 0.977778 | 0.942857 | 0.977778 | 0.942857 | 0.977806 |

In [ ]:

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

In [ ]:

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

In [ ]:

*# 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 |

**1. Which Evaluation metrics did i consider for a positive business impact and why?**

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.

**2. Which ML model did i choose from the above created models as our final prediction model and why?**

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.

**3. Explain the model which i have used for the prediction**

In [ ]:

*# Define a list of category labels for reference.*

Category\_RF = ['Iris-Setosa', 'Iris-Versicolor', 'Iris-Virginica']

In [ ]:

*# 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**

In the Iris flower classification project, the tuned Random Forest model has been selected as the final prediction model. The project aimed to classify Iris flowers into three distinct species: Iris-Setosa, Iris-Versicolor, and Iris-Virginica. After extensive data exploration, preprocessing, and model evaluation, the following conclusions can be drawn:

1. **Data Exploration:** Through a thorough examination of the dataset, we gained insights into the characteristics and distributions of features. We found that Iris-Setosa exhibited distinct features compared to the other two species.
2. **Data Preprocessing:** Data preprocessing steps, including handling missing values and encoding categorical variables, were performed to prepare the dataset for modeling.
3. **Model Selection:** After experimenting with various machine learning models, tuned Random Forest was chosen as the final model due to its simplicity, interpretability, and good performance in classifying Iris species.
4. **Model Training and Evaluation:** The Random Forest (tuned) model was trained on the training dataset and evaluated using appropriate metrics. The model demonstrated satisfactory accuracy and precision in classifying Iris species.
5. **Challenges and Future Work:** The project encountered challenges related to feature engineering and model fine-tuning. Future work may involve exploring more advanced modeling techniques to improve classification accuracy further.
6. **Practical Application:** The Iris flower classification model can be applied in real-world scenarios, such as botany and horticulture, to automate the identification of Iris species based on physical characteristics.

In conclusion, the Iris flower classification project successfully employed Random Forest (tuned) as the final prediction model to classify Iris species. The project's outcomes have practical implications in the field of botany and offer valuable insights into feature importance for species differentiation. Further refinements and enhancements may lead to even more accurate and reliable classification models in the future.