**PRACTICAL JOURNAL**

in

**Machine Learning**

Submitted to

**Laxman Devram Sonawane College, Kalyan (W) 421301**

in partial fulfilment for the award of the degree of

** Master of Science in Information Technology**

(Affiliated to Mumbai University)

*Submitted by*

**Vrushabh Ravindra Pawar**

Under the guidance of

**Dr. Priyanka Pawar**

Department of Information Technology

Kalyan, Maharashtra

Academic Year 2024-25



The Kalyan Wholesale Merchants Education Society’s

**Laxman Devram Sonawane College,**

**Kalyan (W) 421301**

**Department of Information Technology**

**Masters of Science – Part II**

**Certificate**

This is to certify that **Mr.Vrushabh Ravindra pawar**, Seat number **1313274** , studying in Masters of Science in Information Technology Part II , Semester II has satisfactorily completed the practical of “**Machine Learning** ” as prescribed by University of Mumbai, during the academic year 2024-25.

Subject In-charge Coordinator In-charge ExternalExaminer

College Seal

**MACHINE LEARNING**

**INDEX**

|  |  |  |  |
| --- | --- | --- | --- |
| **Sr.no**. | **Practical** | **Date** | **Sign** |
| **1**. | **Data Pre-processing and Exploration**   1. Load a CSV dataset. Handle missing values, inconsistent formatting, and outliers. 2. Load a dataset, calculate descriptive summary statistics, create visualizations using different graphs, and identify potential features and target variables Note: Explore Univariate and Bivariate graphs (Matplotlib) and Seaborn for visualization 3. Create or Explore datasets to use all pre-processing routines like label encoding, scaling, and binarization. |  |  |
| **2**. | **Testing Hypothesis**   1. Implement and demonstrate the FIND-S algorithm for finding the most specific hypothesis based on a given set of training data samples. Read the training data from CSV file and generate the final specific hypothesis. (Create your dataset) |  |  |
| **3**. | **Linear Models**   1. Simple Linear Regression Fit a linear regression model on a dataset. Interpret coefficients, make predictions, and evaluate performance using metrics like R-squared and MSE 2. Multiple Linear Regression Extend linear regression to multiple feature. Handle feature selection and potential multicollinearity 3. Regualarized Linear Models Implement Regression variants like LASSO and Ridge on any generated dataset |  |  |
| **4**. | **Discriminative Models**   1. Logistic Regression : Perform binary classification using logistic regression. Calculate accuracy, precision, recall, and understand the ROC curve." 2. Implement and demonstrate k-nearest Neighbor algorithm. Read the training data from a .CSV file and build the model to classify a test sample. Print both correct and wrong predictions. 3. Build a decision tree classifier or regressor. Control hyperparameters like tree depth to avoid overfitting. Visualize the tree. 4. Implement a Support Vector Machine for any relevant dataset. 5. Train a random forest ensemble. Experiment with the number of trees and feature sampling. Compare performance to a single decision tree. 6. Implement a gradient boosting machine (e.g., XGBoost). Tune hyperparameters and explore feature importance. |  |  |
| **5**. | **Generative Models**   1. Implement and demonstrate the working of a Naive Bayesian classifier using a sample data set. Build the model to classify a test sample. 2. Implement Hidden Markov Models using hmmlearn |  |  |
| **6**. | **Probabilistic Models**   1. Implement Bayesian Linear Regression to explore prior and posterior distribution. 2. Implement Gaussian Mixture Models for density estimation and unsupervised clustering. |  |  |
| **7**. | **Model Evaluation and Hyperparameter Tuning**   1. Implement cross-validation techniques (k-fold, stratified, etc.) for robust model evaluation 2. Systematically explore combinations of hyperparameters to optimize model performance.(use grid and randomized search) |  |  |
| **8**. | **Bayesian Learning**   1. Implement Bayesian Learning using inferences |  |  |

**Practical 1: Data Pre-processing and Exploration**

**1a. Load a CSV dataset. Handle missing values, inconsistent formatting, and outliers.**

**Code :**

1. **Import Libraries**

**# Import necessary libraries**

import pandas as pd import numpy as np

import seaborn as sns

import matplotlib.pyplot as plt

1. **Load the Dataset**

**# Load the Titanic dataset from a URL**

url="https://raw.githubusercontent.com/datasciencedojo/datasets/master/titanic.csv" data = pd.read\_csv(url)

**# Display the first few rows**

print(data.head())

1. **Handle Missing Values**

**# Check for missing values**

print("Missing values in each column:")

print(data.isnull().sum())

**# Fill missing values in 'Age' with the mean**

data['Age'].fillna(data['Age'].mean(), inplace=True)

**# Fill missing values in 'Embarked' with the most common value** data['Embarked'].fillna(data['Embarked'].mode()[0], inplace=True)

**# Drop rows where 'Cabin' is missing (too many NaNs)**

data.drop(columns=['Cabin'], inplace=True)

**# Verify missing values are handled**

print("\nAfter handling missing values:")

print(data.isnull().sum())

1. **Fix Inconsistent Formatting**

**# Fix inconsistent formatting in the 'Sex' column**

data['Sex'] = data['Sex'].str.lower().str.strip()

**# Verify unique values**

print("\nUnique values in 'Sex' column after formatting:")

print(data['Sex'].unique())

1. **Detect and Handle Outliers**  **# Boxplot for the 'Fare' column** sns.boxplot(data['Fare'], color='skyblue') plt.title('Boxplot of Fare') plt.show()

**# Detect outliers using the IQR method**

Q1 = data['Fare'].quantile(0.25) Q3 = data['Fare'].quantile(0.75) IQR = Q3 - Q1 lower\_bound = Q1 - 1.5 \* IQR upper\_bound = Q3 + 1.5 \* IQR

**# Capping outliers**

data['Fare'] = np.where(data['Fare'] > upper\_bound, upper\_bound, np.where(data['Fare'] < lower\_bound, lower\_bound, data['Fare']))

**# Verify with an updated** **boxplot**

sns.boxplot(data['Fare'], color='lightgreen')

plt.title('Boxplot of Fare (After Handling Outliers)')

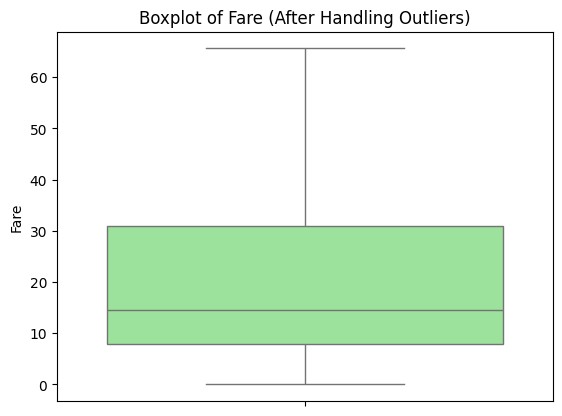
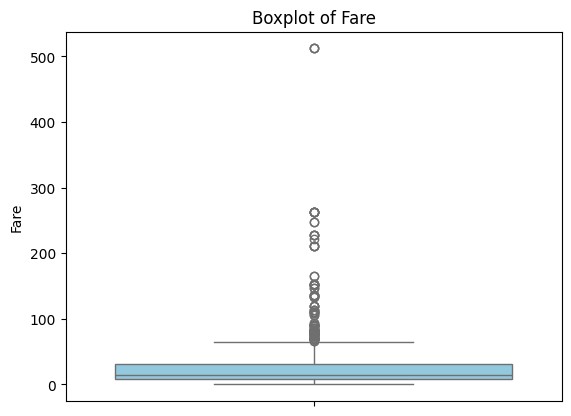
plt.show()

1. **Save the Cleaned Dataset**  **# Save the cleaned dataset**

data.to\_csv('cleaned\_titanic.csv', index=False)

print("\nCleaned dataset saved as 'cleaned\_titanic.csv'") .

**Output :**



**1b. Load a dataset, calculate descriptive summary statistics, create visualizations using different graphs, and identify potential features and target variables Note:**

**Explore Univariate and Bivariate graphs (Matplotlib) and Seaborn for visualization**

**Code :**

1. **Import Necessary Libraries # Import required libraries**

import pandas as pd

import numpy as np

import matplotlib.pyplot as plt

import seaborn as sns

1. **Load the Dataset**

**# Load the dataset from the URL**

url = "https://raw.githubusercontent.com/mwaskom/seaborn-data/master/iris.csv"

data = pd.read\_csv(url)

**# Display the first few rows**

print("First 5 rows of the dataset:") print(data.head())

1. **Calculate Descriptive Summary Statistics # Dataset information**

print("\nDataset Info:")

print(data.info())

**# Summary statistics for numerical columns**

print("\nDescriptive Statistics for Numerical Columns:") print(data.describe())

**# Check unique values for categorical columns**

print("\nUnique values in 'species' column:") print(data['species'].value\_counts())

1. **Univariate Analysis**

**# Histograms for numerical columns**

data.hist(figsize=(10,8), color='skyblue', edgecolor='black') plt.suptitle("Histograms of Numerical Features")

plt.show()

**# Bar plot for 'species' column** sns.countplot(x='species', data=data, palette='pastel') plt.title("Count of Each Species") plt.show()

1. **Bivariate Analysis**

**# Scatter plot for two features**

plt.figure(figsize=(8, 6))

plt.scatter(data['sepal\_length'], data['sepal\_width'], alpha=0.7, c='blue')

plt.title("Sepal Length vs Sepal Width")

plt.xlabel("Sepal Length")

plt.ylabel("Sepal Width")

plt.show()

**# Pairplot to visualize relationships between features** sns.pairplot(data, hue='species', palette='husl', diag\_kind='kde') plt.suptitle("Pairplot of Features by Species", y=1.02)

plt.show()

**# Boxplot for petal\_length across species** sns.boxplot(x='species', y='petal\_length', data=data, palette='Set3')

plt.title("Boxplot of Petal Length by Species")

plt.show()

1. **Identify Potential Features and Target Variables**

**# Separate features and target**

features = data.drop(columns=['species'])

**# Drop the target**

column target = data['species']

**# Target variable**

print("\nFeatures:")

print(features.head())

print("\nTarget:")

print(target.head())

**# Visualize target distribution** sns.countplot(x=target, palette='viridis') plt.title("Target Variable Distribution")

plt.show()

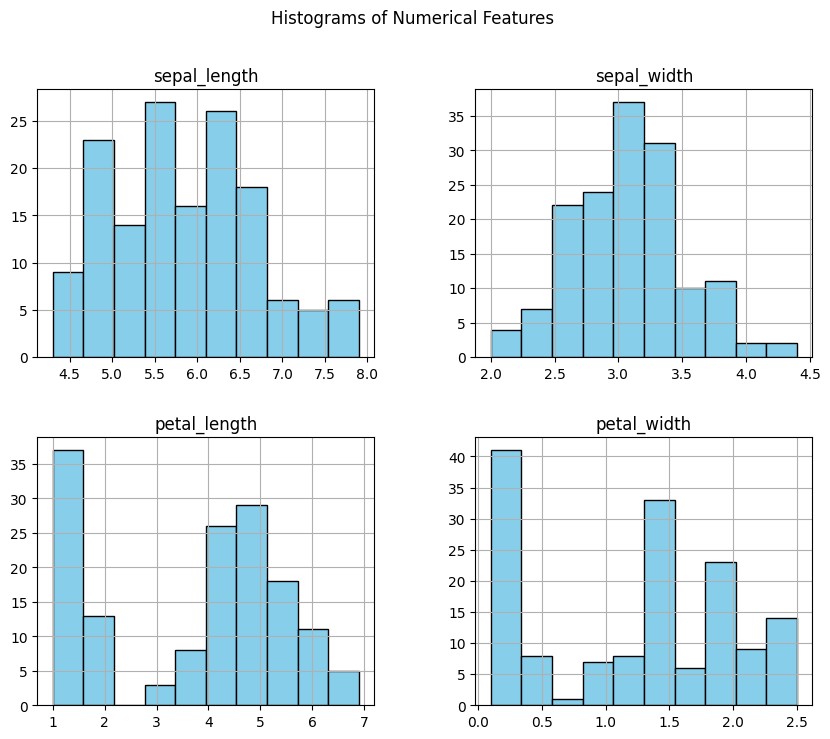
1. **Save the Cleaned and Processed Dataset**

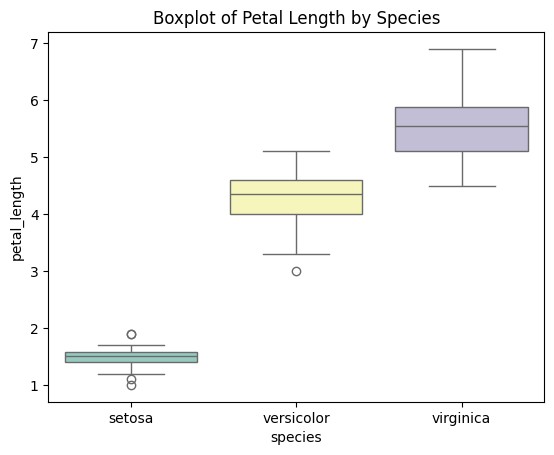
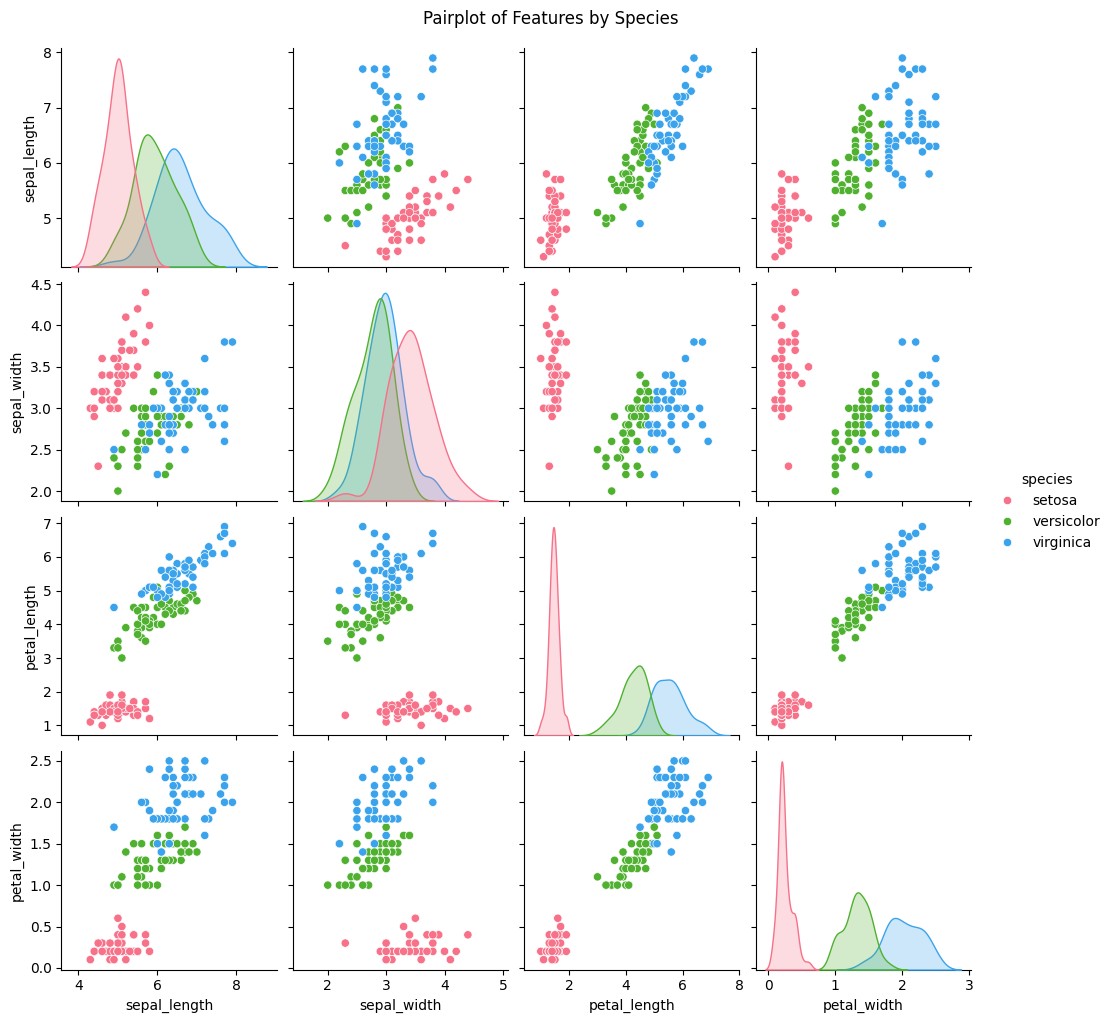
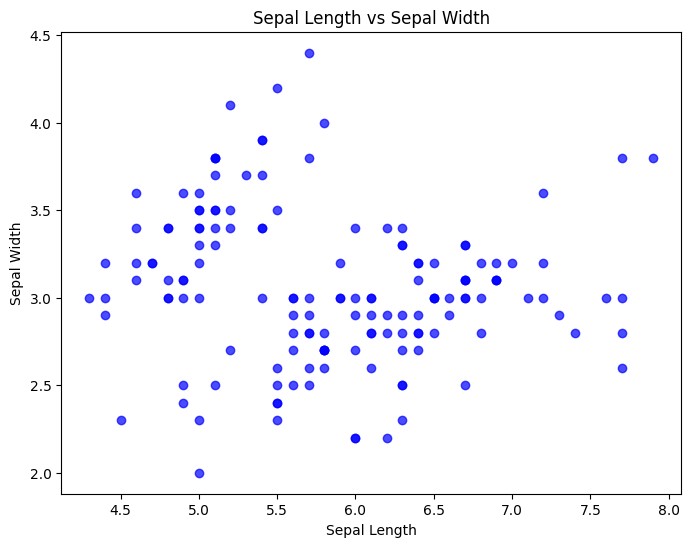
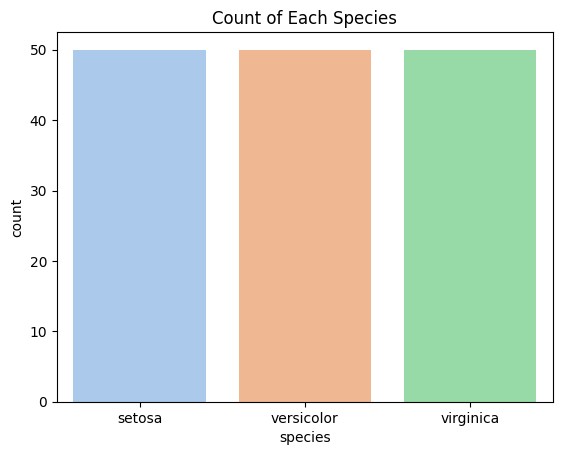
**# Save the dataset**

data.to\_csv('processed\_iris.csv', index=False)

print("\nProcessed dataset saved as 'processed\_iris.csv'")

**Output :**





**1c. Create or Explore datasets to use all pre-processing routines like label encoding, scaling, and binarization.**

**Code :**

1. **Import Necessary Libraries # Import required libraries**

import pandas as pd

import numpy as np from sklearn.preprocessing

import LabelEncoder, MinMaxScaler, StandardScaler, Binarizer

1. **Create or Load a Dataset # Create a sample dataset**

data = pd.DataFrame({

'Category': ['A', 'B', 'C', 'A', 'B', 'C'],

**# Categorical variable**

'Age': [23, 45, 31, 22, 35, 30],

**# Numerical variable**

'Income': [50000, 60000, 70000, 80000, 90000, 100000],

**# Numerical variable 'Has\_Car':**

['Yes', 'No', 'Yes', 'No', 'Yes', 'No']

**# Binary categorical variable** })

**# Display the dataset**

print("Sample Dataset:")

print(data)

1. **Apply Pre-Processing Routines**

**# Label Encoding for 'Category' column**

label\_encoder = LabelEncoder()

data['Category\_Encoded'] = label\_encoder.fit\_transform(data['Category'])

**# Label Encoding for binary column** **'Has\_Car'**

data['Has\_Car\_Encoded'] = label\_encoder.fit\_transform(data['Has\_Car']) print("\nAfter Label Encoding:")

print(data)

**# Min-Max Scaling for 'Income'**

min\_max\_scaler = MinMaxScaler()

data['Income\_MinMax'] = min\_max\_scaler.fit\_transform(data[['Income']])

**# Standard Scaling for 'Age'**

standard\_scaler = StandardScaler()

data['Age\_Standardized'] = standard\_scaler.fit\_transform(data[['Age']]) print("\nAfter Scaling:")

print(data)

**# Binarization for 'Income' with a threshold of 75,000**

binarizer = Binarizer(threshold=75000)

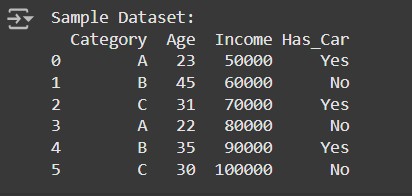
data['Income\_Binary'] = binarizer.fit\_transform(data[['Income']]) print("\nAfter Binarization:")

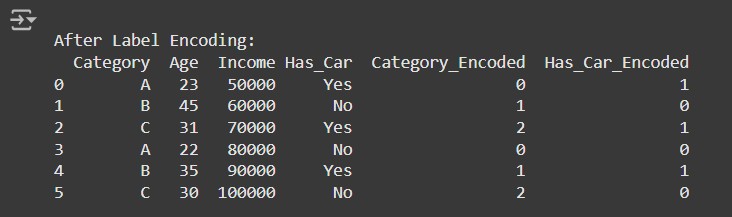
print(data)

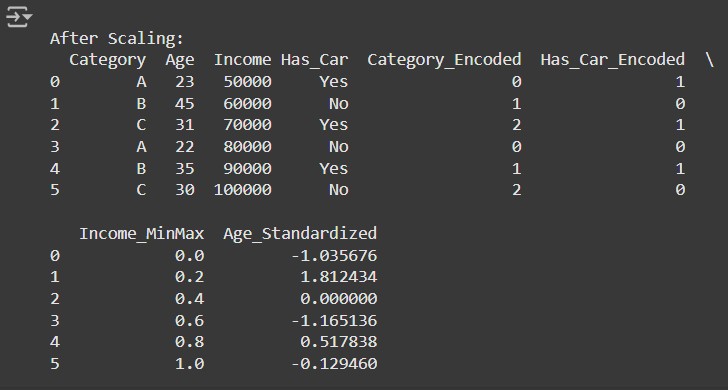
1. **Save the Processed Dataset**

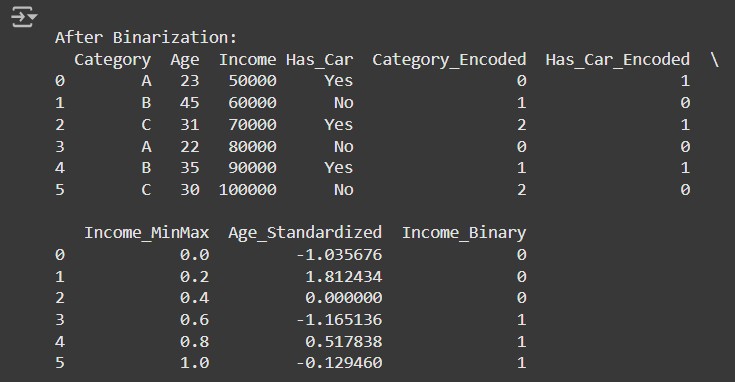
**# Save the processed dataset** data.to\_csv('processed\_data.csv', index=False) print("\nProcessed dataset saved as 'processed\_data.csv'")

**Output :**









**Practical 2 : Testing Hypothesis**

**Implement and demonstrate the FIND-S algorithm for finding the most specific hypothesis based on a given set of training data samples. Read the training data from a. CSV file and generate the final specific hypothesis. (Create your dataset)**

**CODE :**

import pandas as pd

**# Step 1: Create the Dataset and Load It**

data = {'Outlook': ['Sunny', 'Sunny', 'Overcast', 'Rainy', 'Rainy', 'Rainy', 'Overcast', 'Sunny', 'Sunny', 'Rainy'],

'Temperature': ['Hot', 'Hot', 'Hot', 'Mild', 'Cool', 'Cool', 'Cool', 'Mild', 'Cool', 'Mild'],

'Humidity': ['High', 'High', 'High', 'High', 'Normal', 'Normal', 'Normal', 'High', 'Normal', 'Normal'],

'Wind': ['Weak', 'Strong', 'Weak', 'Weak', 'Weak', 'Strong', 'Strong', 'Weak', 'Weak', 'Weak'],

'PlayTennis': ['No', 'No', 'Yes', 'Yes', 'Yes', 'No', 'Yes', 'No', 'Yes', 'Yes']

}

**# Load dataset into a pandas DataFrame**

df = pd.DataFrame(data)

**# Step 2: Implementing the FIND-S Algorithm**

def find\_s\_algorithm(data):

**# Get the positive examples (PlayTennis = 'Yes')**

    positive\_examples = data[data['PlayTennis'] == 'Yes']

**# Initialize hypothesis with the first positive example (most specific)**

    hypothesis = positive\_examples.iloc[0].drop('PlayTennis')

**# Loop through the rest of the positive examples and generalize the hypothesis**

    for index, row in positive\_examples.iterrows():

        for feature in hypothesis.index:

            if hypothesis[feature] != row[feature]:

                hypothesis[feature] = '?'

    return hypothesis

**Step 3: Apply FIND-S to the dataset**

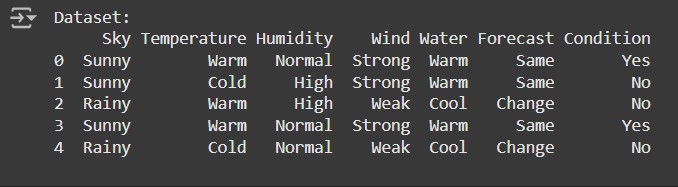
hypothesis = find\_s\_algorithm(df)

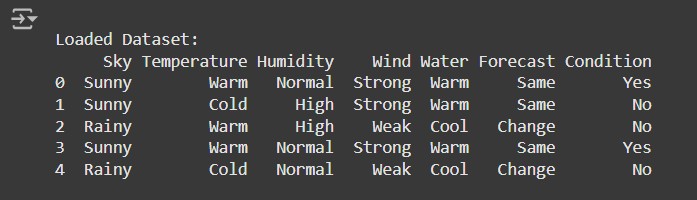
**# Display the final specific hypothesis**

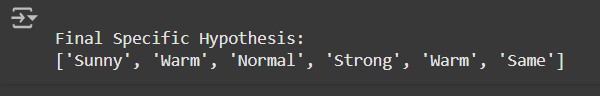
print("The most specific hypothesis is:")

print(hypothesis)

**Output :**







**Practical 3 : Linear Models**

**3a. Simple Linear Regression**

Fit a linear regression model on a dataset. Interpret coefficients, make predictions, and evaluate performance using metrics like R-squared and MSE

**Code :**

**Step 1: Import Libraries # Import required libraries** import pandas as pd

import numpy as np

import matplotlib.pyplot as plt

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

from sklearn.linear\_model import LinearRegression

from sklearn.metrics import mean\_squared\_error, r2\_score

**Step 2: Create a Dataset and Save as CSV**

**# Create a sample dataset**

data = {

'House\_Size': [750, 800, 850, 900, 1000, 1100, 1200, 1300, 1400, 1500],

'Price': [150000, 160000, 165000, 170000, 180000, 190000, 200000, 210000, 220000, 230000]

}

**# Convert the dataset into a DataFrame**

df = pd.DataFrame(data)

**# Save to CSV file** df.to\_csv('house\_prices.csv', index=False)

**# Display the dataset** print("Dataset:") print(df)

**Step 3: Load the Dataset**

**# Load the dataset**

dataset = pd.read\_csv('house\_prices.csv')

**# Display the first few rows** print("\nLoaded Dataset:") print(dataset.head())

**Step 4: Split the Dataset into Training and Test Sets**

**# Features and target variable**

X = dataset[['House\_Size']]  **# Feature: House size**

y = dataset['Price'] **# Target: Price**

**# Split data into training and testing sets (80% train, 20% test)**

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42) print("\nTraining and Testing Data Sizes:")

print("Training Data Size:", X\_train.shape[0])

print("Testing Data Size:", X\_test.shape[0])

**Step 5: Fit a Linear Regression Model**

**# Initialize and fit the linear regression model**

model = LinearRegression()

model.fit(X\_train, y\_train)

**# Display the coefficients**

print("\nModel Coefficients:")

print("Slope (m):", model.coef\_[0]) print("Intercept (b):", model.intercept\_)

**Step 6: Make Predictions**

**# Predict on the test set**

y\_pred = model.predict(X\_test)

**# Display predictions**

print("\nPredictions on Test Data:")

print("Actual Prices:", y\_test.values)

print("Predicted Prices:", y\_pred)

**Step 7: Evaluate the Model**

**# Calculate evaluation metrics**

mse = mean\_squared\_error(y\_test, y\_pred) r2 = r2\_score(y\_test, y\_pred)

**# Display metrics**

print("\nModel Performance Metrics:") print("Mean Squared Error (MSE):", mse) print("R-squared (R²):", r2)

**Step 8: Visualize the Results # Scatter plot of the training data** plt.scatter(X\_train, y\_train, color='blue', label='Training Data')

**# Plot the regression line**

plt.plot(X\_train, model.predict(X\_train), color='red', label='Regression Line')

**# Scatter plot of the test data**

plt.scatter(X\_test, y\_test, color='green', label='Test Data')

plt.title("Simple Linear Regression")

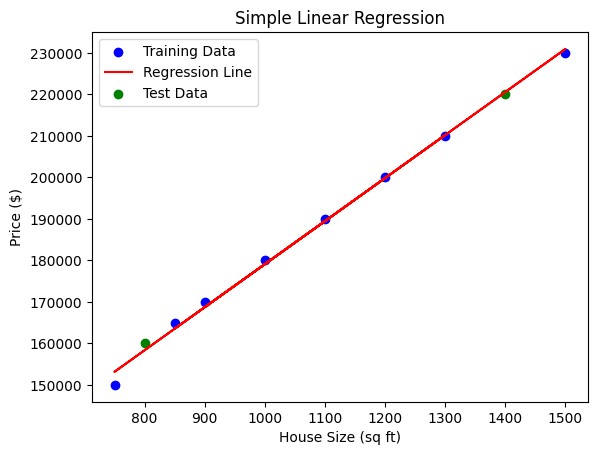
plt.xlabel("House Size (sq ft)")

plt.ylabel("Price ($)")

plt.legend()

plt.show()

**Output :**



**3b. Multiple Linear Regression :**

Extend linear regression to multiple feature. Handle feature selection and potential multicollinearity

**Code :**

import pandas as pd

import numpy as np

import matplotlib.pyplot as plt

import seaborn as sns

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

from sklearn.linear\_model import LinearRegression

from sklearn.metrics import mean\_squared\_error, r2\_score

from statsmodels.stats.outliers\_influence import variance\_inflation\_factor from sklearn.preprocessing import LabelEncoder

**# Import LabelEncoder**

from sklearn.impute import SimpleImputer

**# Load dataset**

from google.colab import files uploaded = files.upload() **# Upload your CSV file**

**# Read the CSV file**

data =pd.read\_csv(list(uploaded.keys())[0])

**# Display the first few rows**

print(data.head())

**# Check for null values and basic statistics**

print(data.info())

print(data.describe())

**# Define a function to calculate VIF**

def calculate\_vif(df):

**# Select only numeric features for VIF calculation**

numeric\_df = df.select\_dtypes(include=np.number)

**# Drop rows with infinite or missing values**

numeric\_df = numeric\_df.replace([np.inf, -np.inf], np.nan).dropna()

vif\_data = pd.DataFrame()

vif\_data["feature"] = numeric\_df.columns

vif\_data["VIF"] = [variance\_inflation\_factor(numeric\_df.values, i)

for i in range(numeric\_df.shape[1])]

return vif\_data

**# Selecting features and target variable**

X = data.drop("Survived", axis=1)

# Changed 'y' to 'Survived' y = data["Survived"]

**# Handle categorical features (e.g., using Label Encoding)**

for col in X.select\_dtypes(include=['object']).columns:

le = LabelEncoder()

X[col] = le.fit\_transform(X[col])

**# Impute missing values using the mean (you can choose other strategies)**

imputer = SimpleImputer(strategy='mean')

**# Create an imputer instance**

X = pd.DataFrame(imputer.fit\_transform(X), columns=X.columns)

**# Impute and update X**

**# Calculate VIF for initial features**

print("VIF before handling multicollinearity:")

print(calculate\_vif(X)) **# Call the modified function**

**# Drop features based on VIF analysis (example: drop 'X1' if VIF is high)**

**# Check if the column exists before dropping**

if 'X1' in X.columns:

X = X.drop("X1", axis=1) # Replace 'X1' with the actual high VIF feature name

else:

print("Column 'X1' not found in the DataFrame.")

**# Recalculate VIF**

print("VIF after handling multicollinearity:")

print(calculate\_vif(X))

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

**# Initialize and fit the model**

model = LinearRegression()

model.fit(X\_train, y\_train)

**# Get coefficients and intercept**

print("Coefficients:", model.coef\_)

print("Intercept:", model.intercept\_)

**# Predictions**

y\_pred = model.predict(X\_test)

**# Evaluation metrics**

rmse = np.sqrt(mean\_squared\_error(y\_test, y\_pred)) r2 = r2\_score(y\_test, y\_pred) print(f"RMSE: {rmse}")

print(f"R^2: {r2}")

from sklearn.feature\_selection import RFE

**# Recursive Feature Elimination**

rfe = RFE(estimator=LinearRegression(), n\_features\_to\_select=5)

# Adjust features

rfe.fit(X\_train, y\_train)

**# Selected features**

print("Selected Features:", X.columns[rfe.support\_])

**# Scatter plot of actual vs predicted values**

plt.scatter(y\_test, y\_pred) plt.xlabel("Actual")

plt.ylabel("Predicted")

plt.title("Actual vs Predicted")

plt.show()

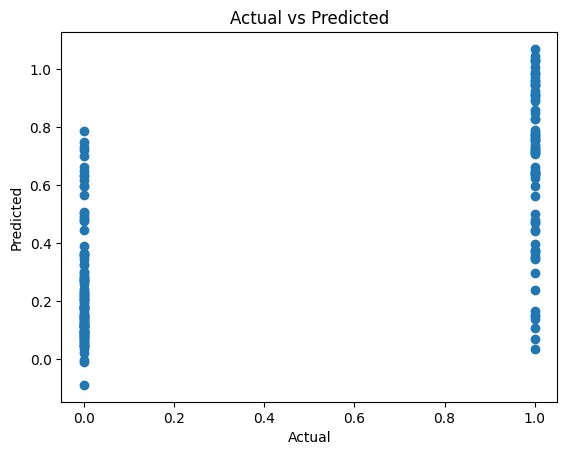
**# Residuals**

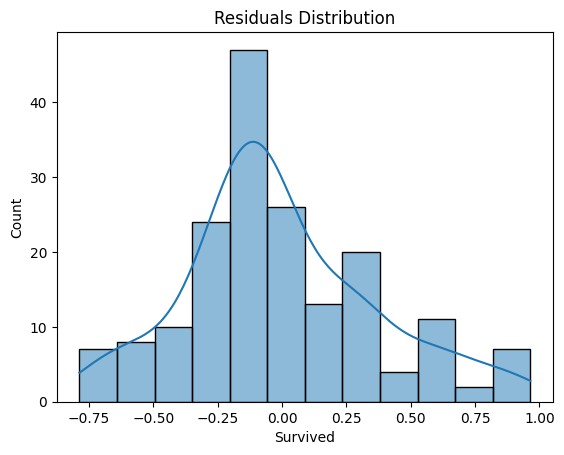
residuals = y\_test - y\_pred sns.histplot(residuals, kde=True)

plt.title("Residuals Distribution")

plt.show()

**Output :**





**3c. Regualarized Linear Models :**

Implement Regression variants like LASSO and Ridge on any generated dataset

**Code :**

1. **Set Up the Environment**

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

import seaborn as sns

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

from sklearn.linear\_model import Ridge, Lasso, ElasticNet

from sklearn.metrics import mean\_squared\_error, r2\_score

from sklearn.datasets import make\_regression

**# Set random seed for reproducibility**

np.random.seed(42)

2. **Generate a Synthetic Dataset**

**# Generate synthetic data**

X, y = make\_regression(n\_samples=1000,

**# Number of samples**

n\_features=10,

**# Number of features**

noise=15,

**# Add some noise**

random\_state=42

)

**# Convert to DataFrame for exploration**

data = pd.DataFrame(X, columns=[f"X{i}"

for i in range(1, 11)]) data["y"] = y

**# Display the first few rows**

print(data.head())

3. **Split the Dataset**

**# Split data into training and testing sets**

X\_train, X\_test, y\_train, y\_test = train\_test\_split(data.drop("y", axis=1),

**# Features**

data["y"],

**# Target variable**

test\_size=0.2,

**# 20% for testing**

random\_state=42

)

4. **Train and Evaluate Ridge Regression**

**# Initialize Ridge Regression with a regularization parameter (alpha)**

ridge = Ridge(alpha=1.0)

**# Train the model**

ridge.fit(X\_train, y\_train)

**# Predictions**

ridge\_pred = ridge.predict(X\_test)

**# Evaluate Ridge Regression**

ridge\_rmse = np.sqrt(mean\_squared\_error(y\_test, ridge\_pred))

ridge\_r2 = r2\_score(y\_test, ridge\_pred)

print(f"Ridge RMSE: {ridge\_rmse}")

print(f"Ridge R^2: {ridge\_r2}")

1. **Train and Evaluate Lasso Regression**

**# Initialize Lasso Regression**

lasso = Lasso(alpha=0.1)

**# Train the model**

lasso.fit(X\_train, y\_train)

**# Predictions**

lasso\_pred = lasso.predict(X\_test)

**# Evaluate Lasso Regression**

lasso\_rmse = np.sqrt(mean\_squared\_error(y\_test, lasso\_pred))

lasso\_r2 = r2\_score(y\_test, lasso\_pred)

print(f"Lasso RMSE: {lasso\_rmse}")

print(f"Lasso R^2: {lasso\_r2}")

**# Features shrunk to**

zero print("Lasso Coefficients:", lasso.coef\_)

1. **Train and Evaluate ElasticNet Regression**

**# Initialize ElasticNet**

elastic\_net = ElasticNet(alpha=0.1, l1\_ratio=0.5) # l1\_ratio balances L1 and L2 penalties

**# Train the model**

elastic\_net.fit(X\_train, y\_train)

**# Predictions**

elastic\_net\_pred = elastic\_net.predict(X\_test)

**# Evaluate**

ElasticNet Regression elastic\_net\_rmse = np.sqrt(mean\_squared\_error(y\_test, elastic\_net\_pred)) elastic\_net\_r2 = r2\_score(y\_test, elastic\_net\_pred)

print(f"ElasticNet RMSE: {elastic\_net\_rmse}")

print(f"ElasticNet R^2: {elastic\_net\_r2}")

1. **Compare Results**

**# Collect metrics**

metrics = pd.DataFrame({

"Model": ["Ridge", "Lasso", "ElasticNet"],

"RMSE": [ridge\_rmse, lasso\_rmse, elastic\_net\_rmse],

"R^2": [ridge\_r2, lasso\_r2, elastic\_net\_r2]

})

print(metrics)

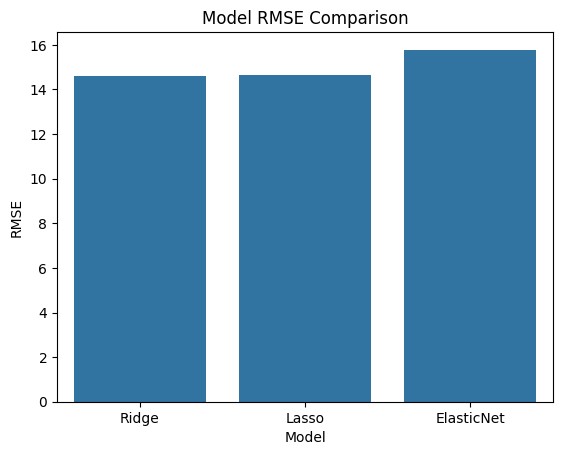
**# Plot RMSE comparison**

sns.barplot(data=metrics, x="Model", y="RMSE")

plt.title("Model RMSE Comparison")

plt.show()

**Output :**



**Practical 4 : Discriminative Models**

**4a. Logistic Regression :**

Perform binary classification using logistic regression. Calculate accuracy, precision, recall, and understand the ROC curve."

**Code :**

**Step 1: Import Required Libraries**

**# Import necessary libraries**

import numpy as np

import pandas as pd

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

from sklearn.linear\_model import LogisticRegression

from sklearn.metrics import accuracy\_score, precision\_score, recall\_score, roc\_curve, auc

import matplotlib.pyplot as plt

**Step 2: Prepare the Dataset**

from sklearn.datasets import make\_classification

**# Create a synthetic dataset**

X, y = make\_classification(n\_samples=1000, n\_features=10, n\_classes=2, random\_state=42)

**# Split data into training and testing sets**

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.3, random\_state=42)

**Step 3: Train the Logistic Regression Model**

**# Initialize the logistic regression model** logreg = LogisticRegression()

**# Train the model on the training data** logreg.fit(X\_train, y\_train)

**Step 4: Make Predictions**

**# Predict labels for the test set** y\_pred = logreg.predict(X\_test)

**# Predict probabilities for the ROC curve** y\_prob = logreg.predict\_proba(X\_test)[:, 1]

**Step 5: Evaluate the Model**

**# Calculate metrics**

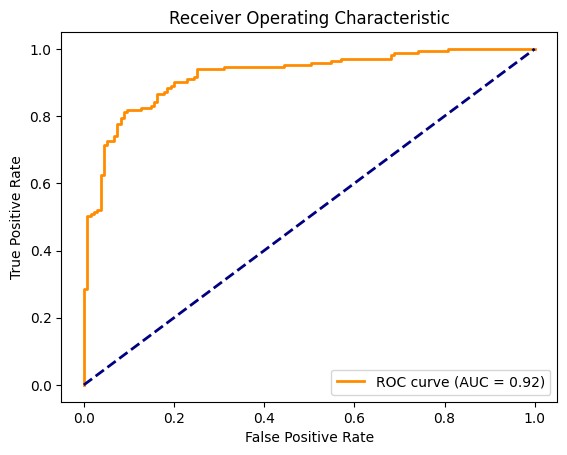
accuracy = accuracy\_score(y\_test, y\_pred)

precision = precision\_score(y\_test, y\_pred)

recall = recall\_score(y\_test, y\_pred)

print(f"Accuracy: {accuracy:.2f}")

**Output :**



**4b .Implement and demonstrate k-nearest Neighbor algorithm. Read the training data from a .CSV file and build the model to classify a test sample. Print both correct and wrong predictions.**

**Code :**

**Step 1:** **Import Required Libraries # Import necessary libraries**

import pandas as pd

import numpy as np

from sklearn.model\_selection import train\_test\_split from sklearn.neighbors import KNeighborsClassifier from sklearn.metrics import accuracy\_score

from google.colab import files

**Step 2: Create or Upload the CSV File**

**# Check if the user wants to create a dataset or upload one**

print("Do you have a CSV file to upload? (yes/no)") response = input().lower()

if response == "yes":

uploaded = files.upload() filename = list(uploaded.keys())[0] else:

**# Create a synthetic dataset**

from sklearn.datasets import make\_classification

**# Generate synthetic data**

X,y=make\_classification(n\_samples=200,n\_features=5, n\_classes=2, random\_state=42**)**

**# Combine features and target into a single DataFrame**

data = pd.DataFrame(X, columns=[f"Feature\_{i}"

for i in range(X.shape[1])])

data['Target'] = y

**# Save the dataset to a CSV file**

filename = "synthetic\_data.csv"

data.to\_csv(filename, index=False)

print(f"Synthetic dataset saved as {filename}.")

**Step 3: Load the CSV File into a DataFrame**

# Load the dataset into a DataFrame

data = pd.read\_csv(filename)

**# Display the first few rows of the dataset**

print("Loaded Dataset:")

print(data.head())

**Step 4: Preprocess the Data**

**# Separate features (X) and labels (y)**

X = data.iloc[:, :-1].values # All columns except the last one y = data.iloc[:, -1].values # Last column as the target

**# Split the dataset into training and testing sets (80% train, 20% test)**

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

**Step 5: Train the k-NN Model # Initialize the k-NN model with k=3** knn = KNeighborsClassifier(n\_neighbors=3) **# Train the model on the training data** knn.fit(X\_train, y\_train)

**Step 6: Predict Test Samples # Predict the labels for the test set** y\_pred = knn.predict(X\_test)

**Step 7: Evaluate and Print Predictions # Calculate and display the accuracy** accuracy = accuracy\_score(y\_test, y\_pred) print(f"\nModel Accuracy: {accuracy:.2f}\n")

**# Display correct and incorrect predictions**

print("Correct Predictions:")

for i in range(len(y\_test)):

if y\_pred[i] == y\_test[i]:

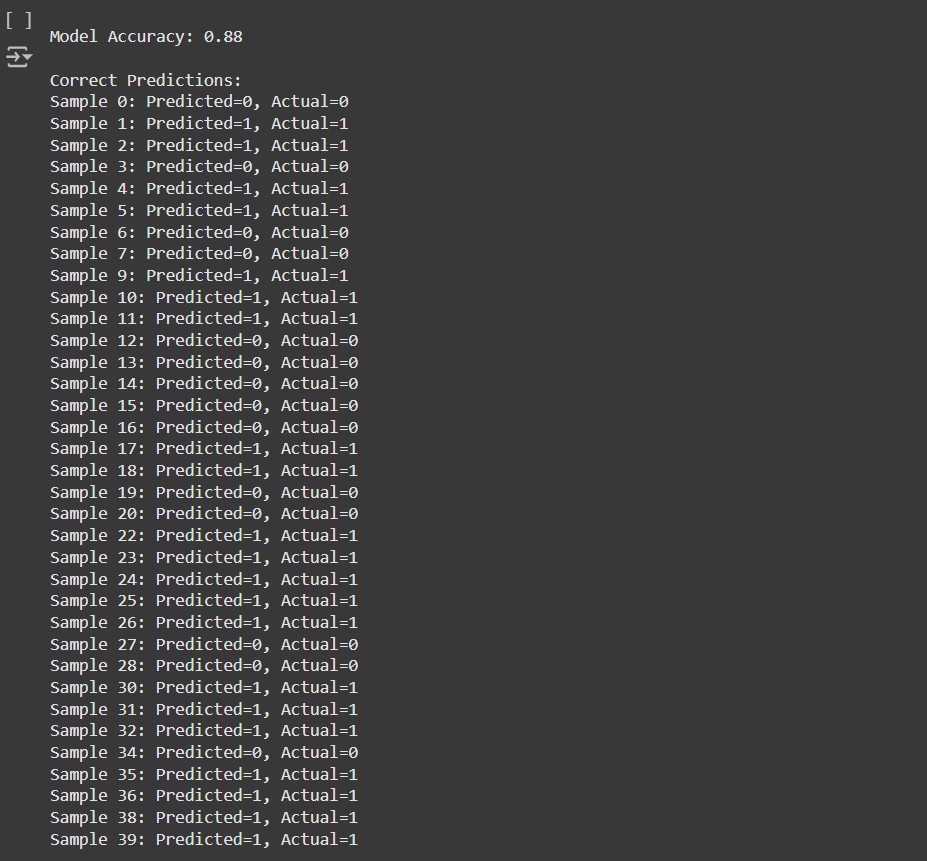
print(f"Sample {i}: Predicted={y\_pred[i]}, Actual={y\_test[i]}") print("\nIncorrect Predictions:")

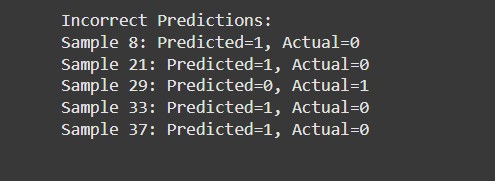
for i in range(len(y\_test)):

if y\_pred[i] != y\_test[i]:

print(f"Sample {i}: Predicted={y\_pred[i]}, Actual={y\_test[i]}")

**Output :**





**4c. Build a decision tree classifier or regressor. Control hyperparameters like tree depth to avoid overfitting. Visualize the tree.**

**Code :**

**Step 1: Import Required Libraries**

**# Import necessary libraries**

import pandas as pd

import numpy as np

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

from sklearn.tree import DecisionTreeClassifier, DecisionTreeRegressor, plot\_tree

from sklearn.metrics import accuracy\_score, mean\_squared\_error

import matplotlib.pyplot as plt

from google.colab import files

**Step 2: Create or Upload the CSV File**

**# Check if the user wants to upload a file or generate one**

print("Do you have a CSV file to upload? (yes/no)") response = input().lower()

if response == "yes":

**# Upload the CSV file**

uploaded = files.upload()

filename = list(uploaded.keys())[0]

else:

**# Generate synthetic data (classification or regression)**

from sklearn.datasets import make\_classification, make\_regression

print("Choose a task: (1) Classification (2) Regression")

task = int(input())

if task == 1:

**# Generate synthetic classification data**

X, y = make\_classification(n\_samples=200, n\_features=5, random\_state=42)

task\_type = "classification"

else:

**# Generate synthetic regression data**

X, y = make\_regression(n\_samples=200, n\_features=5, random\_state=42)

task\_type = "regression"

**# Combine features and target into a single DataFrame**

data = pd.DataFrame(X, columns=[f"Feature\_{i}"

for i in range(X.shape[1])]) data['Target'] = y

**# Save the dataset to a CSV file**

filename = "synthetic\_data.csv"

data.to\_csv(filename, index=False)

print(f"Synthetic {task\_type} dataset saved as {filename}.")

**Step 3: Load the Dataset**

**# Load the dataset**

data =pd.read\_csv(filename)

**# Display the first few rows of the dataset** print("Dataset Preview:")

print(data.head())

**Step 4: Preprocess the Data**

**# Separate features and target**

X = data.iloc[:, :-1].values # All columns except the last one y = data.iloc[:, -1].values # Last column as the target

**# Split data into training and testing sets**

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

**Step 5: Build the Decision Tree**

**# Define the tree depth to avoid overfitting** max\_depth = 3

**# Initialize the model**

if task\_type =="classification":

model = DecisionTreeClassifier(max\_depth=max\_depth, random\_state=42)

else:

model = DecisionTreeRegressor(max\_depth=max\_depth, random\_state=42)

**# Train the model** model.fit(X\_train, y\_train)

**Step 6: Make Predictions**

**# Predict on the test set**

y\_pred = model.predict(X\_test)

**# Evaluate the model**

if task\_type == "classification":

accuracy = accuracy\_score(y\_test, y\_pred)

print(f"Accuracy: {accuracy:.2f}")

else:

mse = mean\_squared\_error(y\_test, y\_pred) print(f"Mean Squared Error: {mse:.2f}")

**Step 7: Visualize the Tree**

**# Visualize the decision tree**

plt.figure(figsize=(12, 8))

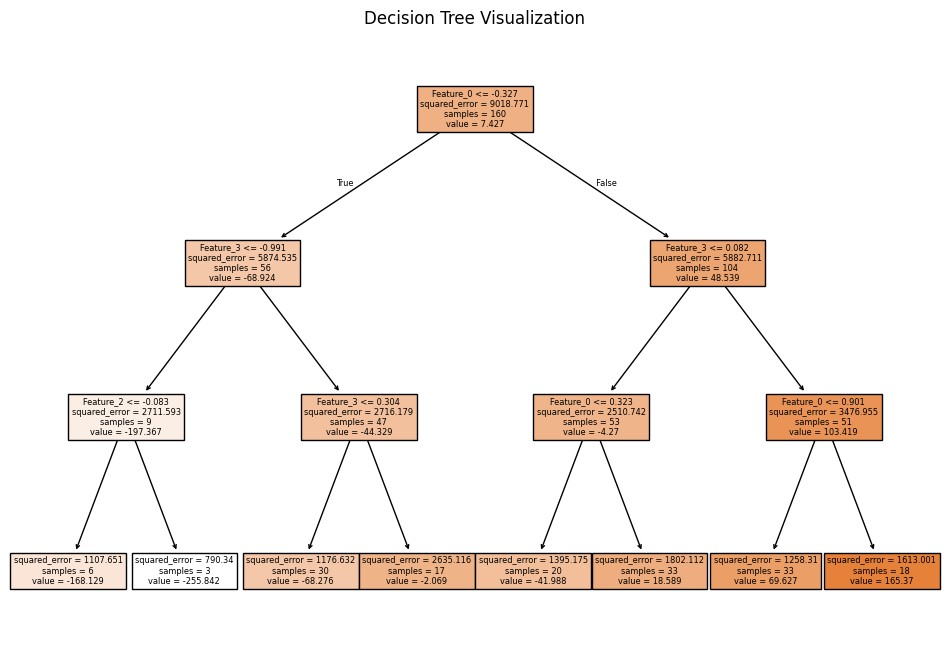
plot\_tree(model, feature\_names=data.columns[:-1], class\_names=str(np.unique(y))

if task\_type == "classification" else None, filled=True)

plt.title("Decision Tree Visualization")

plt.show()

**Output :**



**4d. Implement a Support Vector Machine for any relevant dataset.**

**Code:**

**Step 1: Import Required Libraries**

**# Import necessary libraries**

import pandas as pd

import numpy as np

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

from sklearn.svm import SVC

from sklearn.metrics import accuracy\_score, classification\_report

from google.colab import files

**Step 2: Create or Upload a Dataset**

**# Check if the user wants to upload a file or generate one**

print("Do you have a CSV file to upload? (yes/no)") response = input().lower()

if response == "yes":

**# Upload the CSV file**

uploaded = files.upload()

filename = list(uploaded.keys())[0]

else:

**# Generate synthetic classification data**

from sklearn.datasets import make\_classification

X, y = make\_classification(n\_samples=200, n\_features=5, n\_classes=2, random\_state=42)

**# Combine features and target into a DataFrame**

data = pd.DataFrame(X, columns=[f"Feature\_{i}"

for i in range(X.shape[1])])

data['Target'] = y

**#Save the synthetic dataset to a CSV file**

filename="synthetic\_data.csv" data.to\_csv(filename,index=False) print(f"Synthetic dataset saved as {filename}.")

**Step 3: Load the Dataset**

**# Load the dataset into a DataFrame** data = pd.read\_csv(filename)

**# Display the first few rows of the dataset**

print("Dataset Preview:")

print(data.head())

**Step 4: Preprocess the Data**

**# Separate features (X) and target (y)**

X = data.iloc[:, :-1].values # All columns except the last one y = data.iloc[:, -1].values # Last column as the target

**# Split the dataset into training (80%) and testing (20%) sets**

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

**Step 5: Train the Support Vector Machine**

**# Initialize the SVM model (use RBF kernel as default)**

svm\_model = SVC(kernel='rbf', C=1.0, gamma='scale', random\_state=42)

**# Train the SVM model on the training data** svm\_model.fit(X\_train, y\_train)

**Step 6: Make Predictions**

**# Predict the labels for the test set** y\_pred = svm\_model.predict(X\_test)

**Step 7: Evaluate the Model**

**# Calculate and print the accuracy**

accuracy = accuracy\_score(y\_test, y\_pred) print(f"Model Accuracy: {accuracy:.2f}")

**# Print a detailed classification report** print("\nClassification Report:") print(classification\_report(y\_test, y\_pred))

**Step 8: Visualize the Decision Boundary (Optional for 2D Data)**

import matplotlib.pyplot as plt

**# Generate 2D synthetic data**

from sklearn.datasets import make\_blobs

X, y = make\_blobs(n\_samples=100, centers=2, random\_state=42, cluster\_std=1.5)

**# Fit the SVM on this data** svm\_model.fit(X, y)

**#Plot the decision boundary**

plt.figure(figsize=(8, 6))

plt.scatter(X[:, 0], X[:, 1], c=y, cmap='coolwarm', edgecolor='k')

**# Create a grid to evaluate the model**

xx, yy = np.meshgrid (np.linspace(X[:, 0].min(), X[:, 0].max(), 100), np.linspace(X[:, 1].min(), X[:, 1].max(), 100))

Z = svm\_model.decision\_function(np.c\_[xx.ravel(), yy.ravel()])

Z = Z.reshape(xx.shape)

**# Plot the decision boundary and margins**

plt.contour(xx, yy, Z, levels=[-1, 0, 1], linestyles=['--', '-', '--'], colors='k')

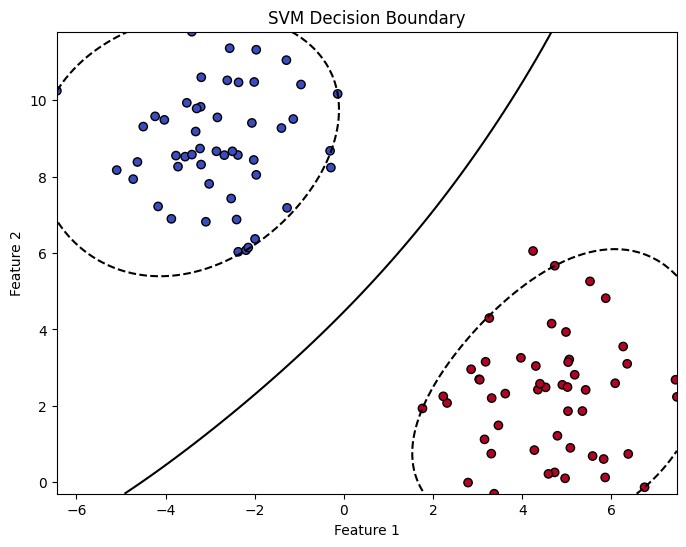
plt.title("SVM Decision Boundary")

plt.xlabel("Feature 1")

plt.ylabel("Feature 2")

plt.show()

**Output :**



**4e. Train a random forest ensemble. Experiment with the number of trees and feature sampling. Compare performance to a single decision tree.**

**Code :**

**Step 1: Import Required Libraries # Import necessarylibraries**

import pandas as pd

import numpy as np

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

from sklearn.tree import DecisionTreeClassifier

from sklearn.ensemble import RandomForestClassifier

from sklearn.metrics import accuracy\_score, classification\_report

from google.colab import files

**Step 2: Create or Upload a Dataset**

**# Check if the user wants to upload a file or generate one**

print("Do you have a CSV file to upload? (yes/no)")

response = input().lower()

if response == "yes":

**# Upload the CSV file**

uploaded = files.upload()

filename = list(uploaded.keys())[0]

else:

**# Generate synthetic classification data**

from sklearn.datasets import make\_classification

X, y = make\_classification(n\_samples=300, n\_features=10, n\_classes=2, random\_state=42)

**# Combine features and target into a DataFrame**

data = pd.DataFrame(X, columns=[f"Feature\_{i}"

for i in range(X.shape[1])])

data['Target'] = y

**# Save the synthetic dataset to a CSV file**

filename = "synthetic\_data.csv"

data.to\_csv(filename, index=False)

print(f"Synthetic dataset saved as {filename}.")

**Step 3: Load the Dataset**

**# Load the dataset**

data =pd.read\_csv(filename)

**# Display the first few rows of the dataset**

print("Dataset Preview:") print(data.head())

**Step 4: Preprocess the Data**

**# Separate features (X) and target (y)**

X = data.iloc[:, :-1].values # All columns except the last one y = data.iloc[:, -1].values # Last column as the target

**# Split the dataset into training (80%) and testing (20%) sets**

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

**Step 5: Train a Single Decision Tree Classifier**

**# Initialize and train the Decision Tree model**

decision\_tree =DecisionTreeClassifier(random\_state=42)

decision\_tree.fit(X\_train, y\_train)

**# Predict and evaluate**

y\_pred\_tree = decision\_tree.predict(X\_test)

accuracy\_tree = accuracy\_score(y\_test, y\_pred\_tree)

print(f"Decision Tree Accuracy: {accuracy\_tree:.2f}")

**Step 6: Train a Random Forest Classifier**

**# Initialize the Random Forest model with hyperparameter tuning**

random\_forest = RandomForestClassifier(n\_estimators=100, max\_features='sqrt', random\_state=42)

**# Train the model** random\_forest.fit(X\_train, y\_train)

**# Predict and evaluate**

y\_pred\_rf = random\_forest.predict(X\_test)

accuracy\_rf = accuracy\_score(y\_test, y\_pred\_rf)

print(f"Random Forest Accuracy (100 trees, sqrt features): {accuracy\_rf:.2f}")

**Step 7: Experiment with Random Forest Hyperparameters**

**# Experiment with fewer trees and different feature sampling**

rf\_experiment = RandomForestClassifier(n\_estimators=50, max\_features=3, random\_state=42)

rf\_experiment.fit(X\_train, y\_train)

**# Predict and evaluate**

y\_pred\_rf\_exp = rf\_experiment.predict(X\_test)

accuracy\_rf\_exp = accuracy\_score(y\_test, y\_pred\_rf\_exp)

print(f"Random Forest Accuracy (50 trees, max\_features=3): {accuracy\_rf\_exp:.2f}")

**Step 8: Compare the Models**

print("\nModel Comparison:")

print(f"Decision Tree Accuracy: {accuracy\_tree:.2f}")

print(f"Random Forest Accuracy (100 trees): {accuracy\_rf:.2f}")

print(f"Random Forest Accuracy (50 trees, max\_features=3): {accuracy\_rf\_exp:.2f}")

**Step 9: Visualize Feature Importance (Optional)** import matplotlib.pyplot as plt

**# Extract feature importance from the Random Forest model** feature\_importances = random\_forest.feature\_importances\_

**# Plot the feature importance**

plt.figure(figsize=(10, 6))

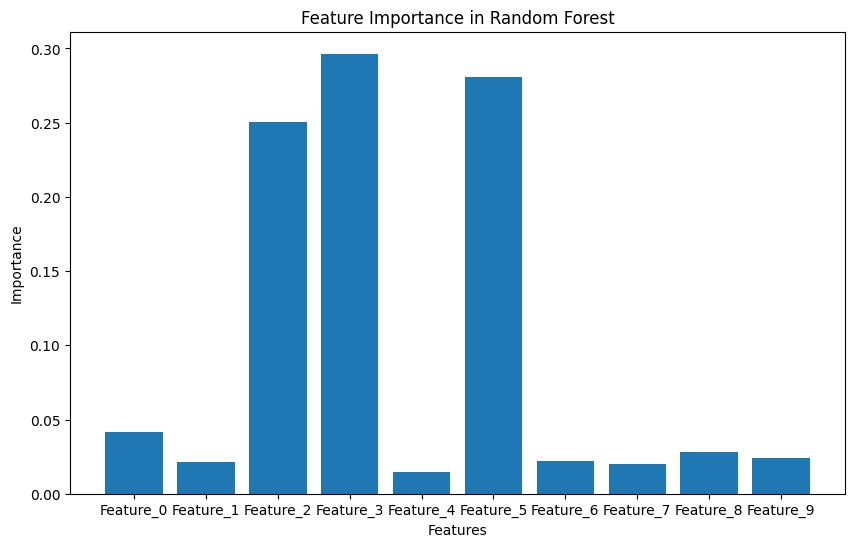
plt.bar(range(len(feature\_importances)), feature\_importances, tick\_label=data.columns[:-1]) plt.title("Feature Importance in Random Forest")

plt.xlabel("Features")

plt.ylabel("Importance")

plt.show()

**Output :**



**4f. Implement a gradient boosting machine (e.g., XGBoost). Tune hyperparameters and explore feature importance.**

**Code :**

**Step 1: Import Required Libraries**

**# Import necessary libraries**

import pandas as pd

import numpy as np

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

from sklearn.metrics import accuracy\_score, classification\_report from xgboost import XGBClassifier, plot\_importance

import matplotlib.pyplot as plt

from google.colab import files

**Step 2: Create or Upload a Dataset**

**# Check if the user wants to upload a file or generate one** print("Do you have a CSV file to upload? (yes/no)") response = input().lower()

if response == "yes":

# Upload the CSV file

uploaded=files.upload()

filename = list(uploaded.keys())[0]

else:

**# Generate synthetic classification data**

from sklearn.datasets import make\_classification

X, y = make\_classification(n\_samples=300, n\_features=10, n\_classes=2, random\_state=42)

**# Combine features and target into a DataFrame**

data = pd.DataFrame(X, columns=[f"Feature\_{i}"

for i in range(X.shape[1])])data['Target'] = y

**# Save the synthetic dataset to a CSV file**

filename="synthetic\_data.csv" data.to\_csv(filename,index=False) print(f"Synthetic dataset saved as {filename}.")

**Step 3: Load the Dataset**

**# Load the dataset**

data = pd.read\_csv(filename)

**# Display the first few rows of the dataset** print("Dataset Preview:") print(data.head())

**Step 4: Preprocess the Data**

**# Separate features (X) and target (y)**

X = data.iloc[:, :-1].values # All columns except the last one y = data.iloc[:, -1].values # Last column as the target

**# Split the dataset into training (80%) and testing (20%) sets**

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

**Step 5: Train a Basic XGBoost Model**

**# Initialize and train the XGBoost model with default parameters** xgb = XGBClassifier(random\_state=42)

xgb.fit(X\_train, y\_train)

**# Predict and evaluate the model**

y\_pred = xgb.predict(X\_test)

accuracy = accuracy\_score(y\_test, y\_pred)

print(f"XGBoost Accuracy (Default Parameters): {accuracy:.2f}")

**Step 6: Tune Hyperparameters with GridSearchCV**

**# Define a grid of hyperparameters**

param\_grid = { 'n\_estimators': [50, 100, 150],'learning\_rate': [0.01, 0.1, 0.2], 'max\_depth': [3, 5, 7] }

**# Initialize GridSearchCV**

grid\_search = GridSearchCV(estimator=XGBClassifier(random\_state=42),param\_grid=param\_grid, scoring='accuracy', cv=3,verbose=1)

**# Fit the model with grid search** grid\_search.fit(X\_train, y\_train)

**# Best parameters from GridSearch**

print(f"Best Parameters: {grid\_search.best\_params\_}")

**# Train the final model with the best parameters**

best\_xgb = grid\_search.best\_estimator\_

**# Predict and evaluate**

y\_pred\_best = best\_xgb.predict(X\_test)

accuracy\_best = accuracy\_score(y\_test, y\_pred\_best)

print(f"XGBoost Accuracy (Tuned Parameters): {accuracy\_best:.2f}")

**Step 7: Explore Feature Importance**

**# Plot feature importance for the tuned model**

plt.figure(figsize=(10, 6))

plot\_importance(best\_xgb,importance\_type='weight', xlabel="Importance", ylabel="Features")

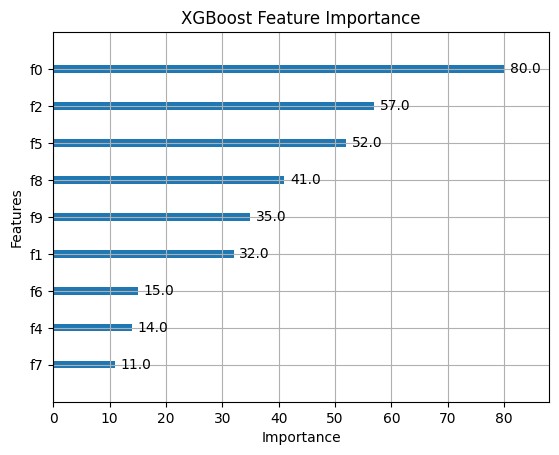
plt.title("XGBoost Feature Importance")

plt.show()

**Step 8: Evaluate the Model**

**# Print a detailed classification report** print("Classification Report:") print(classification\_report(y\_test, y\_pred\_best))

**Output :**



**Practical 5**

**5a. Implement and demonstrate the working of a Naive Bayesian classifier using a sample data set. Build the model to classify a test sample.**

**Step 1: Import Required Libraries**

**# Import necessary libraries**

import pandas as pd

import numpy as np

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

from sklearn.metrics import accuracy\_score,classification\_report from sklearn.naive\_bayes import GaussianNB

from google.colab import files

**Step 2: Create or Upload a Dataset**

**# Ask if the user wants to upload a file or generate one**

print("Do you have a CSV file to upload? (yes/no)")

response = input().lower()

if response == "yes":

**# Upload the CSV file**

uploaded =files.upload()

filename = list(uploaded.keys())[0]

else:

**# Generate synthetic classification data**

from sklearn.datasets import make\_classification

X, y = make\_classification(n\_samples=300, n\_features=8, n\_classes=2, random\_state=42)

**# Combine features and target into a DataFrame**

data = pd.DataFrame(X, columns=[f"Feature\_{i}" for i in range(X.shape[1])])

data['Target'] = y

**# Save the synthetic dataset to a CSV file**  filename="synthetic\_naive\_bayes\_data.csv" data.to\_csv(filename,index=False) print(f"Synthetic dataset saved as {filename}.")

**Step 3: Load the Dataset**

**# Load the dataset**

data =pd.read\_csv(filename)

**# Display the first few rows of the dataset**

print("Dataset Preview:")

print(data.head())

**Step 4: Preprocess the Data**

**# Separate features (X) and target (y)**

X = data.iloc[:, :-1].values # All columns except the last one y = data.iloc[:, -1].values # Last column as the target

**# Split the dataset into training (80%) and testing (20%) sets**

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

**Step 5: Train a Naive Bayes Classifier**

**# Initialize the Gaussian Naive Bayes classifier**

naive\_bayes = GaussianNB()

**# Train the model**

naive\_bayes.fit(X\_train, y\_train)

**Step 6: Make Predictions and Evaluate**

**# Predict on the test set**

y\_pred =naive\_bayes.predict(X\_test)

**# Evaluate the model**

accuracy = accuracy\_score(y\_test, y\_pred) print(f"Naive Bayes Accuracy: {accuracy:.2f}")

**# Detailed classification report**

print("Classification Report:")

print(classification\_report(y\_test, y\_pred))

**Step 7: Test the Model with a Custom Sample**

**# Define a sample test input (replace with meaningful values based on your dataset)**

test\_sample = [X\_test[0]]

**# Taking the first test sample for demonstration**

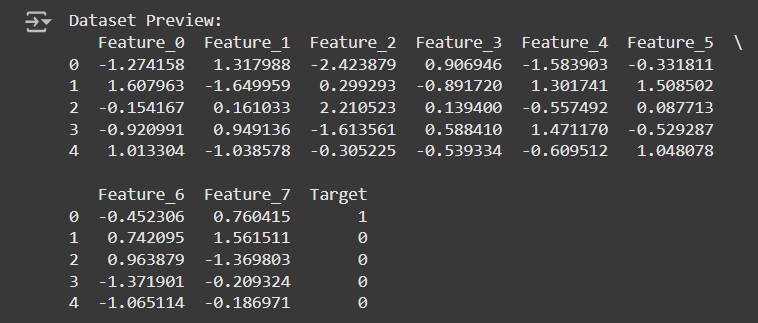
**# Predict the class for the test sample**

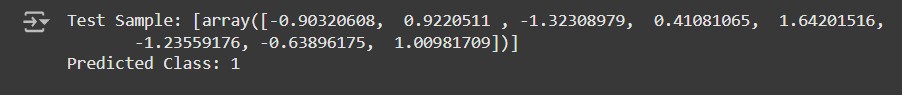
predicted\_class = naive\_bayes.predict(test\_sample)

print(f"Test Sample: {test\_sample}")

print(f"Predicted Class: {predicted\_class[0]}")

**Output :**





**5b. Implement Hidden Markov Models using hmmlearn**

**Code :**

**Step 1: Install Required Libraries**

**# Install hmmlearn**

!pip install hmmlearn

**Step 2: Import Required Libraries**

**# Import necessary libraries** import numpy as np

import pandas as pd

from hmmlearn import hmm

import matplotlib.pyplot as plt

**Step 3: Create or Load a Dataset**

**# Generate synthetic observable data**

np.random.seed(42)

**# Create a sequence of observations and hidden states**

observations = np.random.choice(['A', 'B', 'C'], size=100, p=[0.5, 0.3, 0.2])

hidden\_states = np.random.choice(['X', 'Y'], size=100, p=[0.6, 0.**4])**

**# Save the data in a DataFrame for analysis**

data = pd.DataFrame({'Observations': observations, 'Hidden States': hidden\_states}) print("Generated Data:")

print(data.head())

**Step 4: Encode Observations**

**# Encode the observations into integers**

observation\_mapping = {obs: idx for idx, obs in enumerate(np.unique(observations))} encoded\_observations = np.array([observation\_mapping[obs] for obs in observations]) **# Print the mapping**

print("Observation Encoding:")

print(observation\_mapping)

**Step 5: Initialize and Configure the HMM**

**# Initialize the HMM model**

n\_states = 2 **# Number of hidden states**

n\_observations = len(observation\_mapping)

**# Number of unique observations**

model = hmm.MultinomialHMM(n\_components=n\_states, random\_state=42, n\_iter=100, tol=0.01)

**# Define start probabilities (initial distribution of states)** start\_probs = np.array([0.6, 0.4]) # Assumed probabilities model.startprob\_ = start\_probs

**# Define transition probabilities between states** trans\_probs = np.array([

[0.7, 0.3], # From state X

[0.4, 0.6], # From state Y])

model.transmat\_ = trans\_probs

**# Define emission probabilities (probability of observations given states)** emission\_probs = np.array([

[0.5, 0.4, 0.1], # State X emits A, B, C

[0.2, 0.3, 0.5], # State Y emits A, B, C

])

model.emissionprob\_ = emission\_probs

**# Print the configured model parameters** print("Start Probabilities:", model.startprob\_) print("Transition Matrix:", model.transmat\_) print("Emission Probabilities:", model.emissionprob\_)

**Step 6: Train the Model**

**# Reshape the data for HMM (requires 2D array)** encoded\_observations = encoded\_observations.reshape(-1, 1)

**# Fit the model** model.fit(encoded\_observations)

**# Predict hidden states for the observations** predicted\_states = model.predict(encoded\_observations)

**# Print the predicted states** print("Predicted States:") print(predicted\_states)

**Step 7: Visualize the Results**

**# Map predicted states back to their original labels**

state\_mapping = {0: 'X', 1: 'Y'}

predicted\_state\_labels = [state\_mapping[state] for state in predicted\_states]

**# Add predicted states to the DataFrame** data['Predicted States'] = predicted\_state\_labels

**# Display the first few rows with predicted states**

print("Data with Predicted States:")

print(data.head())

**# Plot the observations and predicted states**

plt.figure(figsize=(12, 6))

plt.plot(data['Observations'], label='Observations', marker='o', linestyle='-', alpha=0.7) plt.plot(data['Predicted States'], label='Predicted States', marker='x', linestyle='--', alpha=0.7) plt.legend()

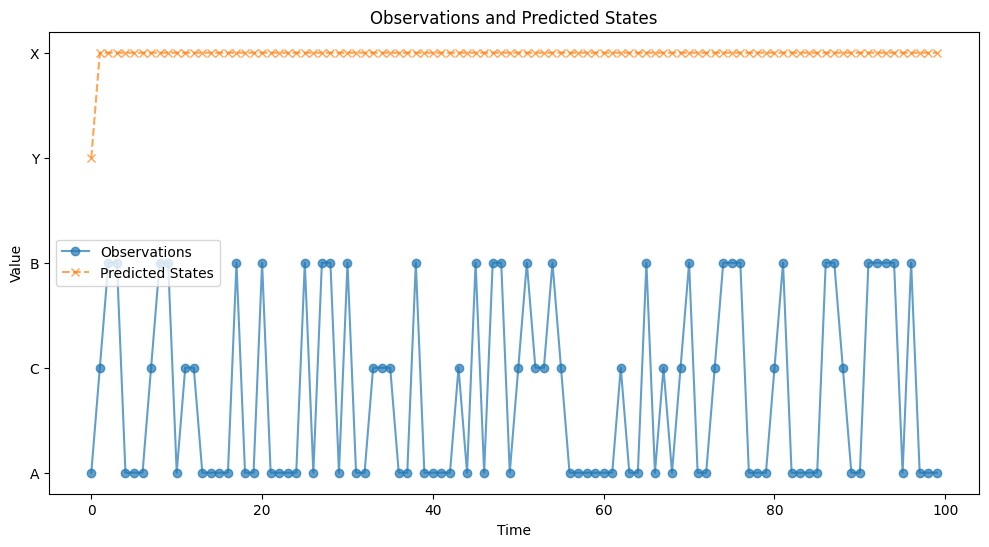
plt.title("Observations and Predicted States")

plt.xlabel("Time")

plt.ylabel("Value")

plt.show()

**Output :**



**Practical 6 : Probabilistic Model**

**6a. Implement Bayesian Linear Regression to explore prior and posterior distribution.**

Bayesian Linear Regression is a probabilistic approach to linear regression that incorporates uncertainty in the model parameters. Instead of estimating point values for parameters (as in traditional linear regression), we estimate distributions over the parameters.

**Code :**

**Step 1: Install Required Libraries**

**# Install necessary libraries**

!pip install matplotlib seaborn scikit-learn

**Step 2: Import Required Libraries**

**# Import necessary libraries**

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

import seaborn as sns

from sklearn.linear\_model import BayesianRidge from sklearn.model\_selection import train\_test\_split from sklearn.metrics import mean\_squared\_error from google.colab import files

**Step 3: Create or Upload a Dataset**

**# Upload a CSV file if you have one**

print("Do you have a CSV file to upload? (yes/no)")

response = input().lower()

if response == "yes":

# Upload the CSV file

uploaded = files.upload()

filename = list(uploaded.keys())[0] else:

**# Generate synthetic data for demonstration** np.random.seed(42)

X = np.random.rand(100, 1) \* 10

**# Random data between 0 and 10**

y = 2 \* X + 1 + np.random.randn(100, 1) \* 2

**# y = 2x + 1 with some noise**

**# Convert to a DataFrame**

data = pd.DataFrame(np.hstack((X, y)), columns=["X", "y"])

**# Save to CSV for convenience**

filename="synthetic\_data.csv" data.to\_csv(filename,index=False) print(f"Synthetic dataset saved as {filename}.")

**Step 4: Load and Explore the Data**

**# Load the dataset (for CSV file)**

data = pd.read\_csv(filename)

**# Display first few rows**

print("Dataset Preview:")

print(data.head())

**Step 5: Preprocess the Data**

**# Separate features (X) and target (y)**

X = data["X"].values.reshape(-1, 1) # Feature matrix y = data["y"].values # Target vector

**# Split the dataset into training (80%) and testing (20%) sets**

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

**Step 6: Implement Bayesian Linear Regression Model**

**# Initialize the BayesianRidge model (which implements Bayesian Linear Regression)** bayesian\_regressor = BayesianRidge()

**# Fit the model on the training data**

bayesian\_regressor.fit(X\_train, y\_train)

**# Predict on the test data**

y\_pred = bayesian\_regressor.predict(X\_test)

**Step 7: Visualize the Prior and Posterior Distributions**

**# Plot the prior and posterior distributions of the parameters**

fig, ax = plt.subplots(1, 2, figsize=(12, 6))

**# Plot prior distribution (assuming the model starts with a standard prior)** ax[0].set\_title("Prior Distribution (Assumed)") ax[0].hist(np.random.normal(0, 1, 1000), bins=50, alpha=0.7, color='blue', label="Prior") ax[0].legend()

**# Plot posterior distribution (after model fitting)**

ax[1].set\_title("Posterior Distribution (After Fitting)") ax[1].hist(bayesian\_regressor.coef\_, bins=50, alpha=0.7, color='green', label="Posterior") ax[1].legend()

plt.show()

**Step 8: Evaluate the Model Performance # Calculate the Mean Squared Error (MSE)** mse = mean\_squared\_error(y\_test, y\_pred) print(f"Mean Squared Error (MSE): {mse:.2f}")

**Step 9: Visualize the Fit of the Model**

**# Plot the true values and the predicted values**

plt.figure(figsize=(8, 6))

plt.scatter(X\_test, y\_test, color="blue", label="True values") plt.plot(X\_test, y\_pred, color="red", label="Predicted values", linewidth=2)

plt.title("Bayesian Linear Regression: True vs Predicted Values") plt.xlabel("X")

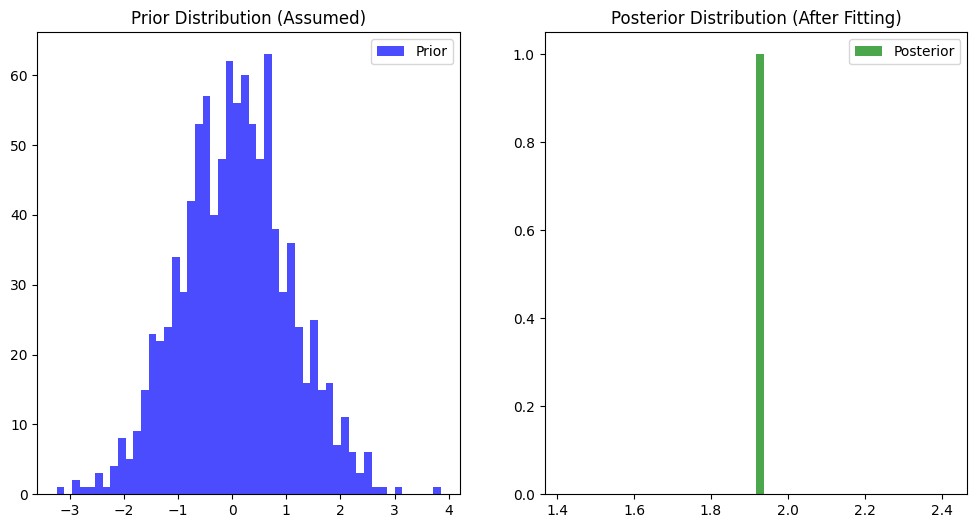
plt.ylabel("y")

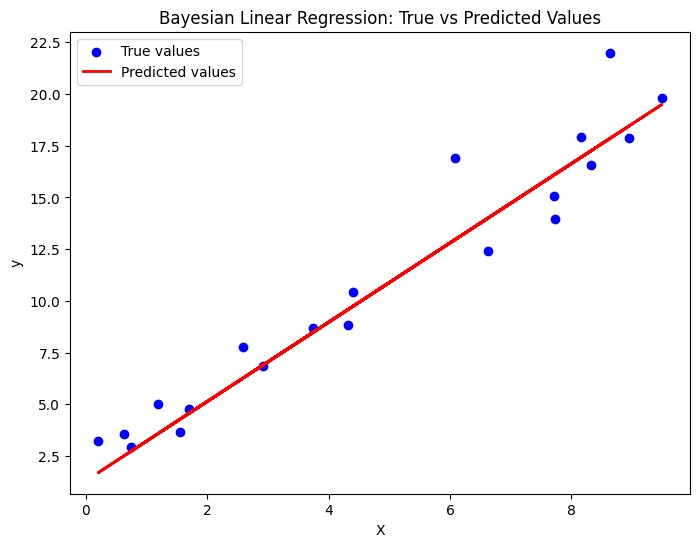
plt.legend()

plt.show()

**Mean Squared Error (MSE): 3.9**

**Output :**





**6b. Implement Gaussian Mixture Models for density estimation and unsupervised clustering.**

**Code :**

**Step 1: Install Required Libraries**

**# Install required libraries**

!pip install matplotlib seaborn scikit-learn

**Step 2: Import Required Libraries**

**# Import necessary libraries**

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

import seaborn as sns

from sklearn.mixture import GaussianMixture

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

from google.colab import files

**Step 3: Create or Upload a Dataset**

**#Ask if the user has a CSV file to upload**

print("Do you have a CSV file to upload? (yes/no)")

response = input().lower()

if response == "yes":

**# Upload the CSV file**

uploaded = files.upload()

filename = list(uploaded.keys())[0]

else:

**# Generate synthetic 2D data with two clusters for demonstration**

np.random.seed(42)

**# Generate data for two Gaussian distributions**

X1 = np.random.normal(loc=0, scale=1, size=(300, 2)) # Cluster 1: mean = 0, std = 1

X2 = np.random.normal(loc=5, scale=1, size=(300, 2)) # Cluster 2: mean = 5, std = 1  **# Stack the data to create a dataset**

X = np.vstack([X1, X2])

**# Create DataFrame to simulate the CSV file for consistency**

data = pd.DataFrame(X, columns=["Feature\_1", "Feature\_2"])

filename = "synthetic\_data.csv"

data.to\_csv(filename, index=False)

print(f"Synthetic dataset saved as {filename}.")

**Step 4: Load and Explore the Dataset**

**# Load the dataset (if CSV file is uploaded)**

data = pd.read\_csv(filename)

**# Display the first few rows**

print("Dataset Preview:")

print(data.head())

**# Plot the data to visualize its structure** sns.scatterplot(data=data, x="Feature\_1", y="Feature\_2") plt.title("Synthetic Data")

plt.xlabel("Feature 1")

plt.ylabel("Feature 2")

plt.show()

**Step 5: Fit a Gaussian Mixture Model (GMM)**

**# Define the GMM model**

n\_components = 2 **# Number of Gaussian distributions (clusters)**

gmm = GaussianMixture(n\_components=n\_components, covariance\_type='full', random\_state=42)

**# Fit the GMM model to the data** gmm.fit(data)

**# Predict the cluster labels for each data point**

labels = gmm.predict(data)

**# Add the cluster labels to the dataset for visualization**

data['Cluster'] = labels

**# Plot the clustered data**

sns.scatterplot(data=data, x="Feature\_1", y="Feature\_2", hue="Cluster", palette="viridis", marker="o")

plt.title("Gaussian Mixture Model Clustering")

plt.xlabel("Feature 1")

plt.ylabel("Feature 2")

plt.legend()

plt.show()

**Step 6: Visualize the Gaussian Mixture Model (GMM) Components**

**# Extract the means and covariances of the Gaussian components**

means = gmm.means\_

covariances = gmm.covariances\_

**# Plot the GMM components on top of the data**

plt.figure(figsize=(8, 6)) **# Plot data points**

sns.scatterplot(data=data, x="Feature\_1", y="Feature\_2", hue="Cluster", palette="viridis", marker="o", s=60, alpha=0.7)

**# Plot the GMM ellipses** for mean, covar in zip(means, covariances):

**# Plot the Gaussian components as ellipses**

v, w = np.linalg.eigh(covar)

v = 2.0 \* np.sqrt(2.0) \* np.sqrt(v)

**# Scaling factor for the ellipse**

u = w[0] / np.linalg.norm(w[0])

**# Normalize the eigenvector**

angle = np.arctan(u[1] / u[0])

**# Create the ellipse**

angle = angle \* 180.0 / np.pi **# Convert to degrees**   
ellipse = plt.matplotlib.patches.Ellipse(mean, v[0], v[1], angle=angle, color='red', alpha=0.3) plt.gca().add\_patch(ellipse)

plt.title("GMM Clustering with Gaussian Components")

plt.xlabel("Feature 1")

plt.ylabel("Feature 2")

plt.legend()

plt.show()

**Step 7: Model Evaluation (Optional)**

**# Compute the log-likelihood of the data under the fitted GMM model**

log\_likelihood = gmm.score(data)

print(f"Log-Likelihood of the data: {log\_likelihood:.2f}")

**Step 8: Predict New Data Points**

**# Example of predicting the cluster for new data points** new\_data = np.array([[1.5, 2.5], [4.5, 5.5], [7.0, 8.0]])

new\_labels = gmm.predict(new\_data)

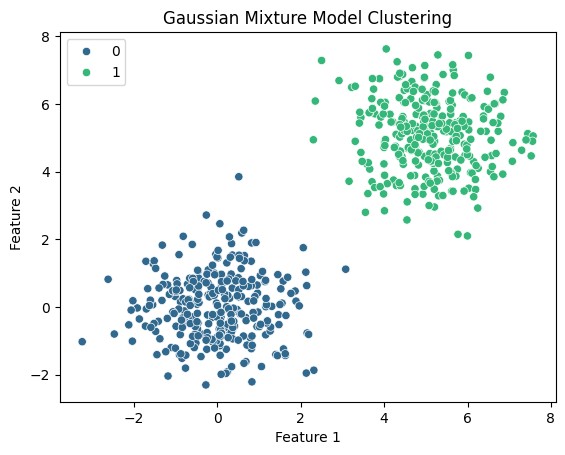
**# Print the predicted clusters for the new data points**

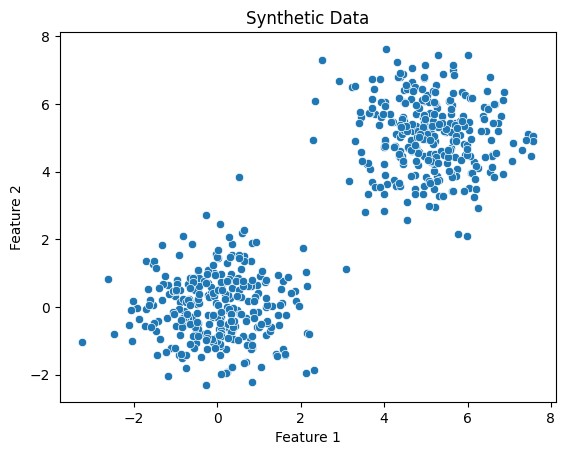
print("Predicted Clusters for New Data Points:")

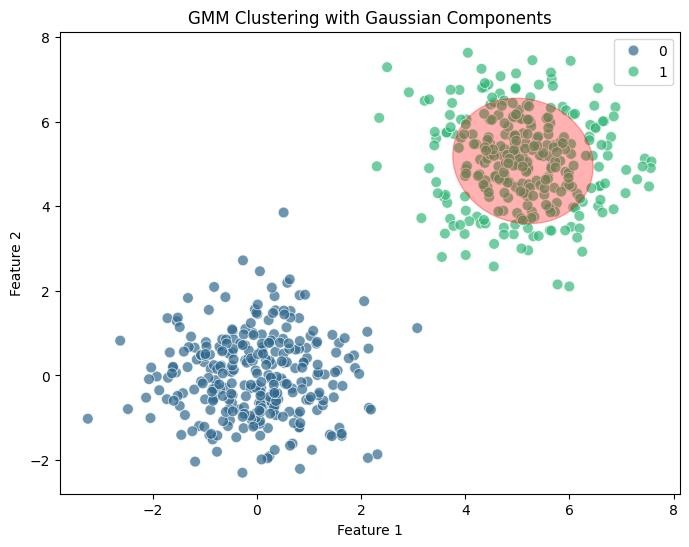
for i, label in enumerate(new\_labels):

print(f"Data point {new\_data[i]} is in Cluster {label}")

**Output :**







**Practical 7 : Model Evaluation and Hyperparameter Tuning**

**7a. Implement cross-validation techniques (k-fold, stratified, etc.) for robust model evaluation**

**Code :**

**1. Import Necessary Libraries**

import numpy as np

import pandas as pd

from sklearn.datasets import make\_classification

from sklearn.model\_selection import train\_test\_split, KFold, StratifiedKFold, GridSearchCV

from sklearn.ensemble import RandomForestClassifier

from sklearn.metrics import accuracy\_score, classification\_report, confusion\_matrix

import matplotlib.pyplot as plt

import seaborn as sns

**2. Generate a Synthetic Dataset**

**# Create a synthetic dataset with 2 classes**

X, y = make\_classification(

n\_samples=1000, n\_features=10, n\_informative=8, n\_redundant=2,

n\_clusters\_per\_class=1, random\_state=42

)

**# Convert to a DataFrame for visualization**

df = pd.DataFrame(X, columns=[f'Feature\_{i}' for i in range(1, 11)])

df['Target'] = y

**# Display the first few rows**

print(df.head())

**3. Split Data into Train and Test Sets**

**# Split data into 80% training and 20% testing**

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42, stratify=y)

**4. Define k-Fold Cross-Validation**

kf = KFold(n\_splits=5, shuffle=True, random\_state=42)

print("k-Fold Cross-Validation:")

for train\_index, val\_index in kf.split(X\_train):

print("TRAIN:", train\_index, "VALIDATION:", val\_index)

**5. Define Stratified k-Fold Cross-Validation**

skf = StratifiedKFold(n\_splits=5, shuffle=True, random\_state=42)

print("\nStratified k-Fold Cross-Validation:")

for train\_index, val\_index in skf.split(X\_train, y\_train):

print("TRAIN:", train\_index, "VALIDATION:", val\_index)

**6. Train and Evaluate Using k-Fold Cross-Validation**

**# Initialize model**

model = RandomForestClassifier(random\_state=42)

**# Perform k-Fold Cross-Validation**

accuracies = []

for train\_index, val\_index in kf.split(X\_train):

X\_kf\_train, X\_kf\_val = X\_train[train\_index], X\_train[val\_index]

y\_kf\_train, y\_kf\_val = y\_train[train\_index], y\_train[val\_index]

**# Train model**

model.fit(X\_kf\_train, y\_kf\_train)

**# Validate model**

y\_pred = model.predict(X\_kf\_val)

accuracy = accuracy\_score(y\_kf\_val, y\_pred)

accuracies.append(accuracy)

print(f"Average Accuracy from k-Fold: {np.mean(accuracies):.2f}")

**7. Hyperparameter Tuning Using GridSearchCV**

**# Define parameter grid**

param\_grid = {

'n\_estimators': [50, 100, 200],

'max\_depth': [None, 10, 20, 30],

'min\_samples\_split': [2, 5, 10],

}

**# Perform GridSearchCV with Stratified k-Fold**

grid\_search = GridSearchCV(

estimator=RandomForestClassifier(random\_state=42),

param\_grid=param\_grid,

cv=StratifiedKFold(n\_splits=5, shuffle=True, random\_state=42),

scoring='accuracy',

n\_jobs=-1,

verbose=1

)

**# Fit to training data**

grid\_search.fit(X\_train, y\_train)

print("Best Parameters:", grid\_search.best\_params\_)

print("Best Cross-Validation Accuracy:", grid\_search.best\_score\_)

**8. Evaluate the Final Model**

**# Use the best model for evaluation**

best\_model = grid\_search.best\_estimator\_

**# Predict on test dat**a

y\_test\_pred = best\_model.predict(X\_test)

**# Evaluate performance**

print("\nTest Accuracy:", accuracy\_score(y\_test, y\_test\_pred))

print("\nClassification Report:\n", classification\_report(y\_test, y\_test\_pred))

**# Confusion matrix**

conf\_matrix = confusion\_matrix(y\_test, y\_test\_pred)

sns.heatmap(conf\_matrix, annot=True, fmt='d', cmap='Blues', xticklabels=['Class 0', 'Class 1'], yticklabels=['Class 0', 'Class 1'])

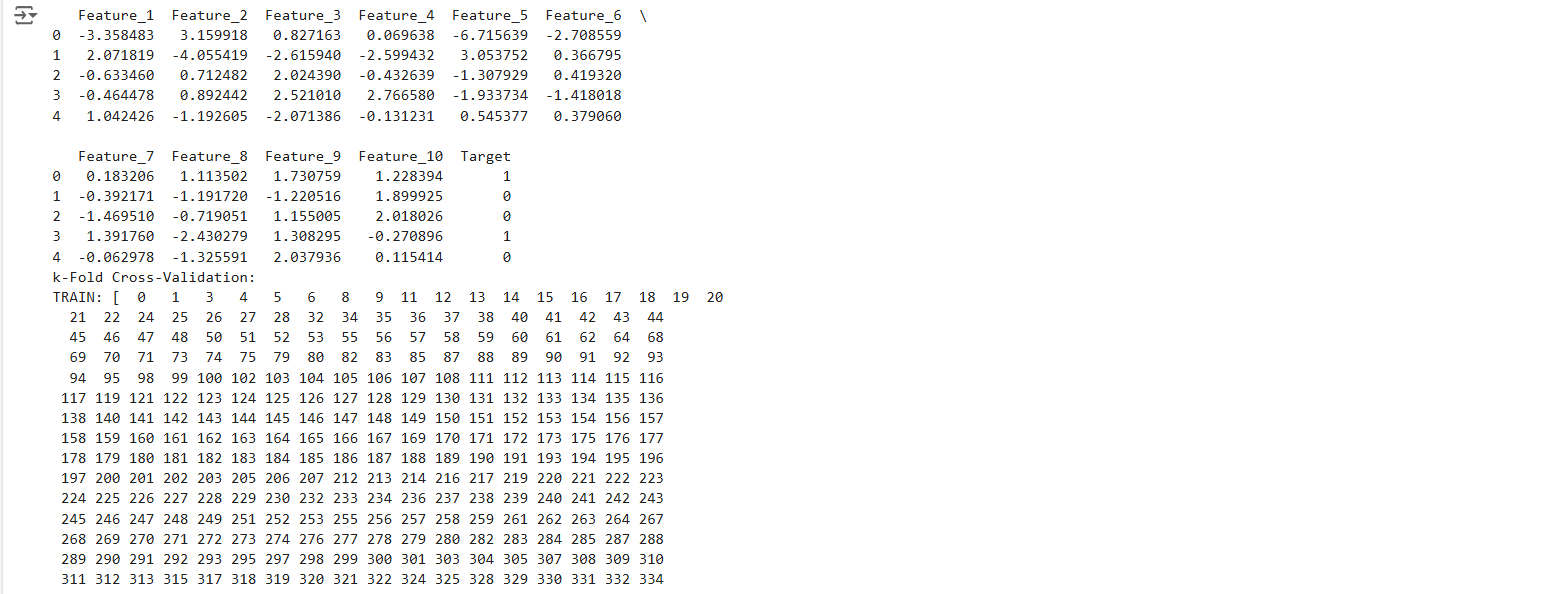
plt.xlabel('Predicted')

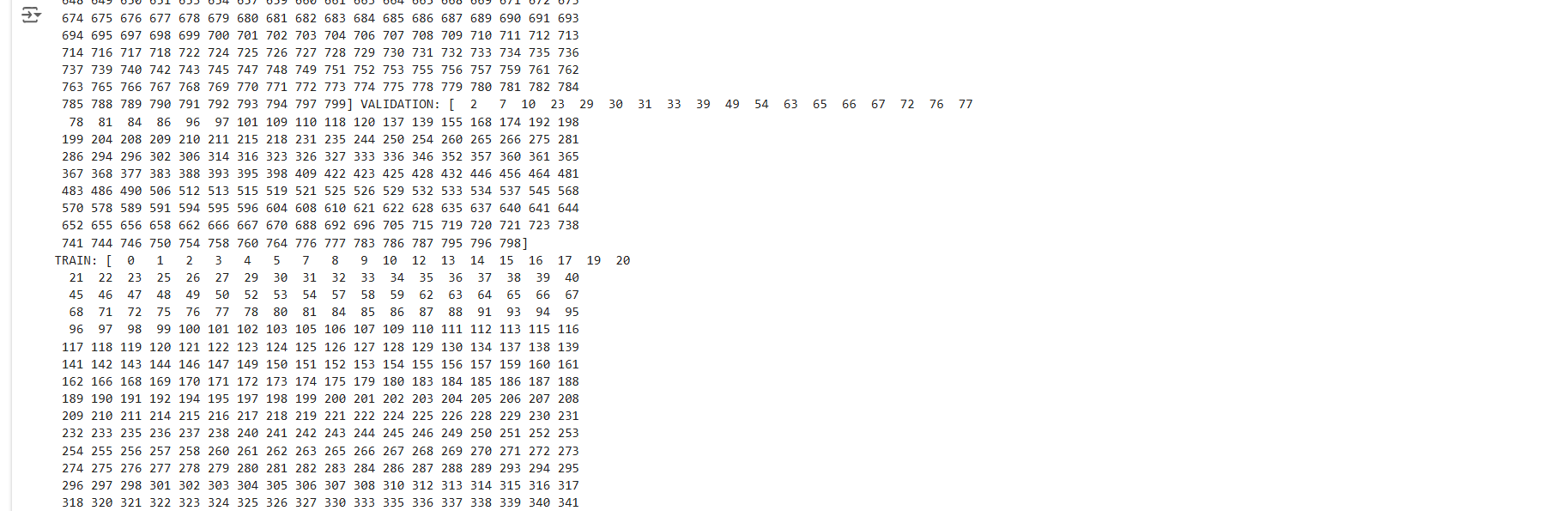
plt.ylabel('Actual')

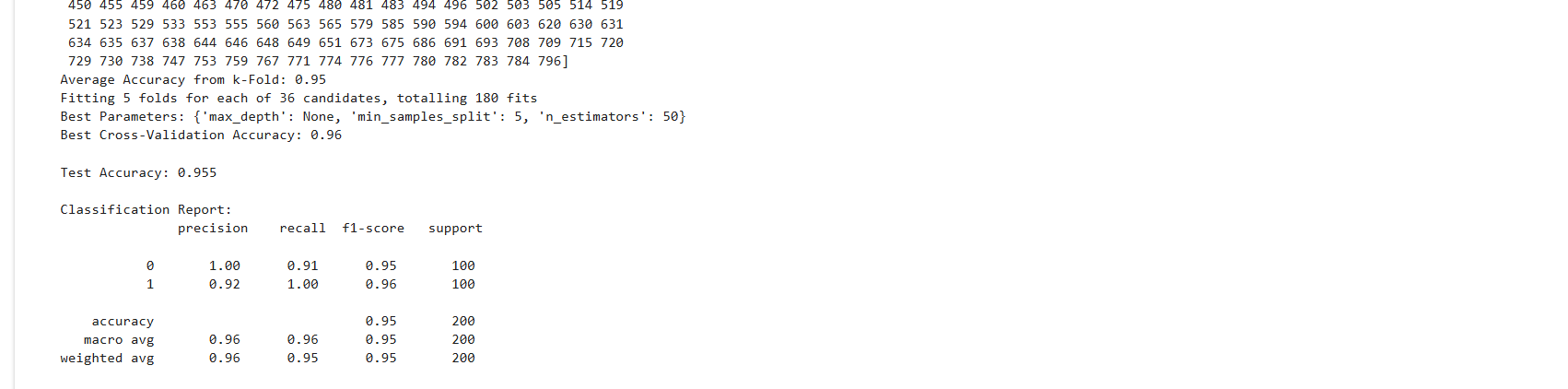
plt.title('Confusion Matrix')

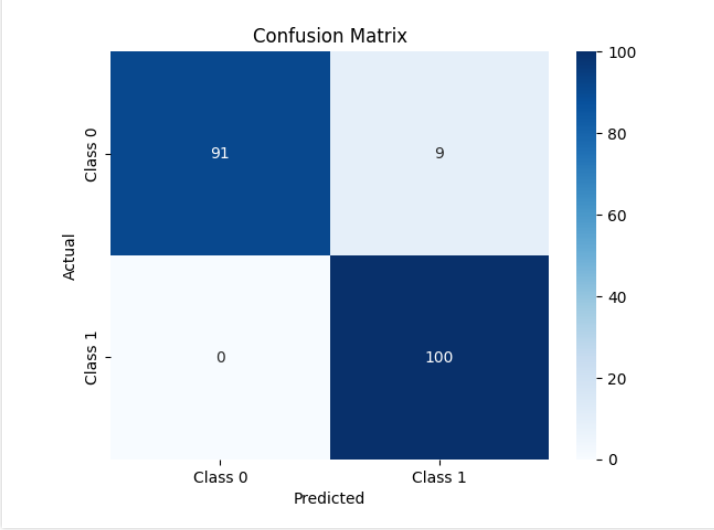
plt.show()

**Output :**

****







**7b. Systematically explore combinations of hyperparameters to optimize model performance.(use grid and randomized search)**

**Code :**

**1. Import Necessary Libraries**

import numpy as np

import pandas as pd

from sklearn.datasets import make\_classification

from sklearn.model\_selection import train\_test\_split, GridSearchCV, RandomizedSearchCV, StratifiedKFold

from sklearn.ensemble import RandomForestClassifier

from sklearn.metrics import accuracy\_score, classification\_report, confusion\_matrix

import matplotlib.pyplot as plt

import seaborn as sns

**2. Generate a Synthetic Dataset**

**# Generate a binary classification dataset**

X, y = make\_classification(

n\_samples=1000, n\_features=12, n\_informative=8, n\_redundant=2,

n\_clusters\_per\_class=1, flip\_y=0.03, random\_state=42

)

**# Convert to a DataFrame for visualization**

df = pd.DataFrame(X, columns=[f'Feature\_{i}' for i in range(1, 13)])

df['Target'] = y

**# Display the first few rows**

print(df.head())

**3. Split Data into Train and Test Sets**

**# Split data into training and testing sets**

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42, stratify=y)

**4. Define the Model**

**# Initialize a Random Forest classifier**

model = RandomForestClassifier(random\_state=42)

**5. Hyperparameter Tuning Using Grid Search**

# Define a parameter grid for Grid Search

param\_grid = {

'n\_estimators': [50, 100, 200],

'max\_depth': [None, 10, 20],

'min\_samples\_split': [2, 5, 10],

'min\_samples\_leaf': [1, 2, 4]

}

**# GridSearchCV with 5-fold cross-validation**

grid\_search = GridSearchCV(

estimator=model,

param\_grid=param\_grid,

scoring='accuracy',

cv=StratifiedKFold(n\_splits=5, shuffle=True, random\_state=42),

verbose=1,

n\_jobs=-1

)

**# Fit the model**

grid\_search.fit(X\_train, y\_train)

**# Best parameters and score from Grid Search**

print("Best Parameters from Grid Search:", grid\_search.best\_params\_)

print("Best Cross-Validation Accuracy from Grid Search:", grid\_search.best\_score\_)

**6. Hyperparameter Tuning Using Randomized Search**

from scipy.stats import randint

**# Define a parameter distribution for Randomized Search**

param\_dist = {

'n\_estimators': randint(50, 300),

'max\_depth': [None, 10, 20, 30],

'min\_samples\_split': randint(2, 15),

'min\_samples\_leaf': randint(1, 10)

}

**# RandomizedSearchCV with 5-fold cross-validation**

random\_search = RandomizedSearchCV(

estimator=model,

param\_distributions=param\_dist,

n\_iter=50, # Number of random combinations to try

scoring='accuracy',

cv=StratifiedKFold(n\_splits=5, shuffle=True, random\_state=42),

verbose=1,

n\_jobs=-1,

random\_state=42

)

**# Fit the model**

random\_search.fit(X\_train, y\_train)

**# Best parameters and score from Randomized Search**

print("Best Parameters from Randomized Search:", random\_search.best\_params\_)

print("Best Cross-Validation Accuracy from Randomized Search:", random\_search.best\_score\_)

**7. Evaluate the Best Model**

# Select the best model from Grid Search and Randomized Search

best\_model = random\_search.best\_estimator\_ # Or use grid\_search.best\_estimator\_

**# Predict on test data**

y\_test\_pred = best\_model.predict(X\_test)

**# Evaluate the performance**

print("\nTest Accuracy:", accuracy\_score(y\_test, y\_test\_pred))

print("\nClassification Report:\n", classification\_report(y\_test, y\_test\_pred))

**# Confusion Matrix**

conf\_matrix = confusion\_matrix(y\_test, y\_test\_pred)

sns.heatmap(conf\_matrix, annot=True, fmt='d', cmap='Blues', xticklabels=['Class 0', 'Class 1'], yticklabels=['Class 0', 'Class 1'])

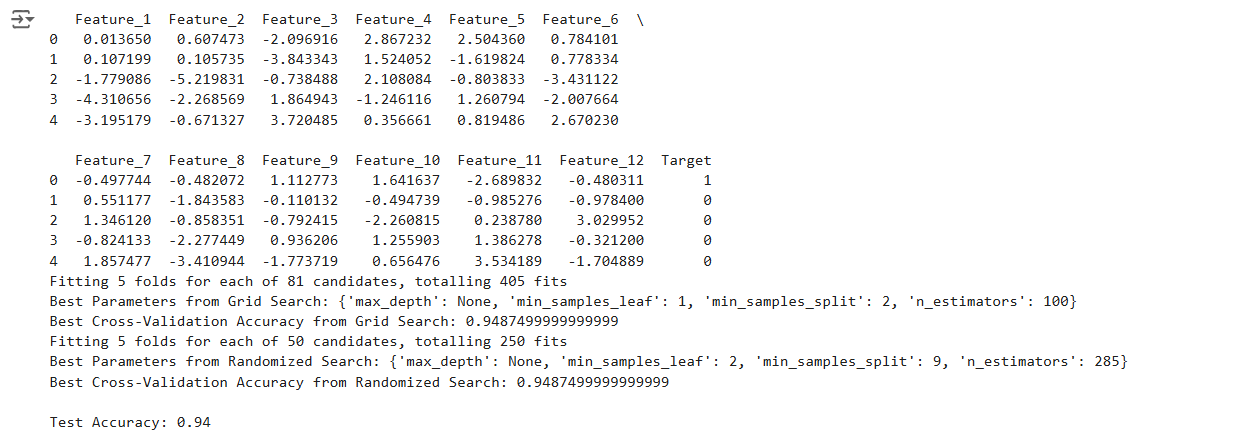
plt.xlabel('Predicted')

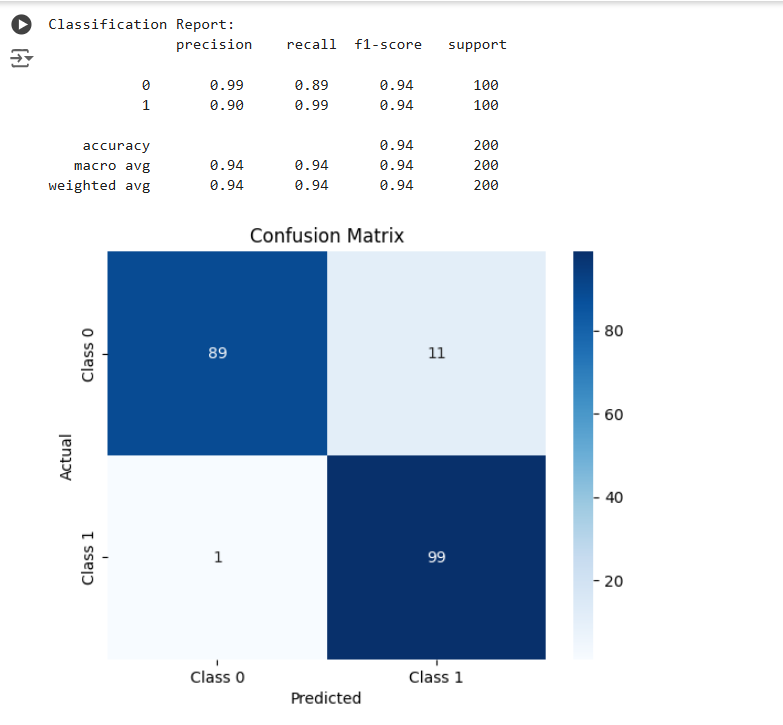
plt.ylabel('Actual')

plt.title('Confusion Matrix')

plt.show()

**Output :**

****

****

**Practical 8 : Bayesian Learning**

**Implement Bayesian Learning using inferences**

**Code :**

**1. Import Necessary Libraries**

import numpy as np

import pandas as pd

from sklearn.datasets import make\_classification

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

from sklearn.naive\_bayes import GaussianNB

from sklearn.metrics import accuracy\_score, classification\_report, confusion\_matrix

import seaborn as sns

import matplotlib.pyplot as plt

**2. Generate a Synthetic Dataset**

We create a dataset suitable for classification problems.

**# Generate a dataset with 2 classes**

X, y = make\_classification(

n\_samples=1000, n\_features=8, n\_informative=6, n\_redundant=2,

n\_classes=2, random\_state=42)

**# Convert to DataFrame for visualization**

df = pd.DataFrame(X, columns=[f'Feature\_{i}' for i in range(1, 9)])

df['Target'] = y

**# Display the first few rows**

print(df.head())

**3. Split the Dataset**

Divide the data into training and testing sets.

**# Split data into 80% training and 20% testing**

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42, stratify=y)

**4. Bayesian Learning with Naive Bayes**

Here, we implement Bayesian Learning using the Gaussian Naive Bayes classifier.

**# Initialize the Gaussian Naive Bayes model**

model = GaussianNB()

**# Fit the model to the training data**

model.fit(X\_train, y\_train)

**# Predict on the test data**

y\_pred = model.predict(X\_test)

**5. Evaluate the Model**

We evaluate the model's performance using accuracy, classification report, and confusion matrix.

**# Calculate accuracy**

accuracy = accuracy\_score(y\_test, y\_pred)

print(f"Test Accuracy: {accuracy:.2f}")

**# Print classification report**

print("\nClassification Report:\n", classification\_report(y\_test, y\_pred))

**# Generate and plot confusion matrix**

conf\_matrix = confusion\_matrix(y\_test, y\_pred)

sns.heatmap(conf\_matrix, annot=True, fmt='d', cmap='Blues', xticklabels=['Class 0', 'Class 1'], yticklabels=['Class 0', 'Class 1'])

plt.xlabel('Predicted')

plt.ylabel('Actual')

plt.title('Confusion Matrix')

plt.show()

**6. Understanding Bayesian Inference**

In Bayesian Learning, the model predicts based on the probabilities:

* **Prior Probability (P(C)P(C)P(C)):** The likelihood of each class based on historical data.
* **Likelihood (P(X∣C)P(X|C)P(X∣C)):** The probability of the data given a class.
* **Posterior Probability (P(C∣X)P(C|X)P(C∣X)):** Calculated using Bayes' theorem: P(C∣X)=P(X∣C)⋅P(C)P(X)P(C|X) = \frac{P(X|C) \cdot P(C)}{P(X)}P(C∣X)=P(X)P(X∣C)⋅P(C)​

**# Example: Compute posterior probabilities for the first test sample**

sample = X\_test[0].reshape(1, -1)

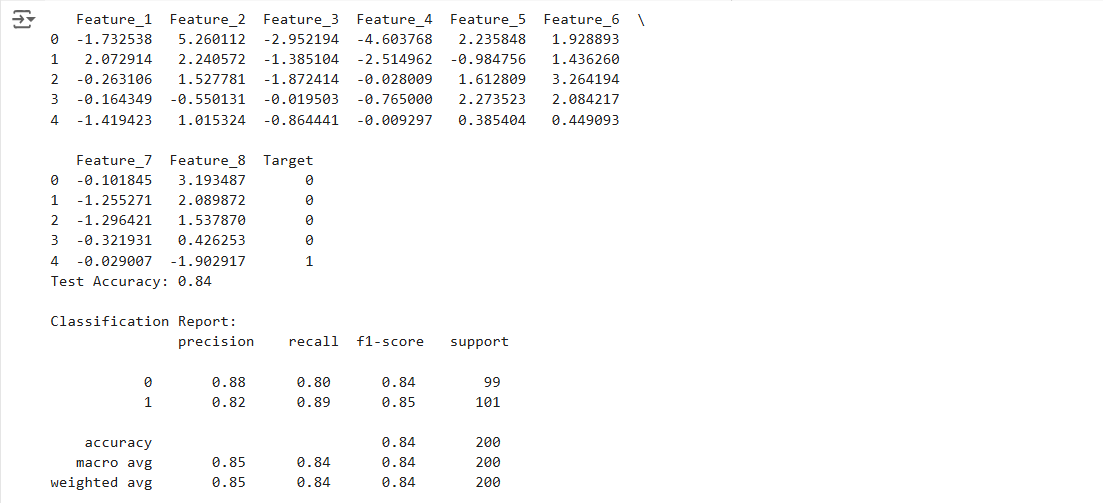
posterior\_probs = model.predict\_proba(sample)

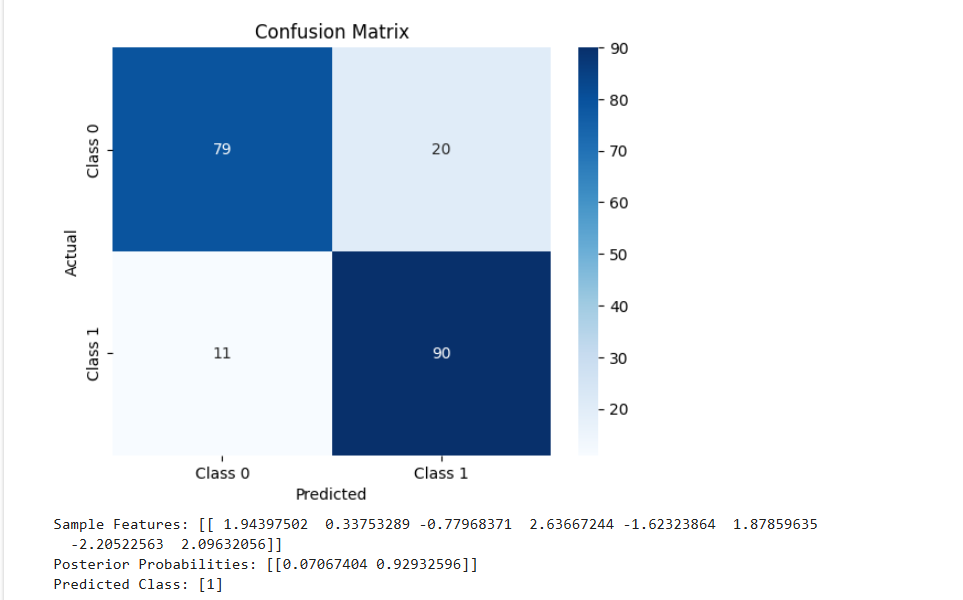
print(f"Sample Features: {sample}")

print(f"Posterior Probabilities: {posterior\_probs}")

print(f"Predicted Class: {model.predict(sample)}")

**Output :**

****

****