INDEX

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Sr.**  **No.** | **Name of Experiment** | | **Date of Experiment** | | **Date of Submission** | | **Signature** | |
| **1** | 1. Data Pre-processing and Exploration a. Load a CSV dataset. Handle missing values, inconsistent formatting, and outliers. b. 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. c. Create or Explore datasets to use all pre-processing routines like label  encoding, scaling, and binarization | |  | |  | |  | |
| **2** | Testing Hypothesis a. 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) | |  | |  | |  | |
| **3** | Linear Models  a. 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 b. Multiple Linear Regression Extend linear regression to  multiple features. Handle feature selection and potential multicollinearity.  c. Regularized Linear Models (Ridge, Lasso, ElasticNet) Implement regression variants like LASSO and Ridge on any  generated dataset. | |  | |  | |  | |
| **4** | Discriminative Models a Logistic Regression Perform binary  classification using logistic regression. Calculate accuracy, precision, recall, and understand the ROC curve. b. 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. c. Build a decision tree classifier or regressor.  Control hyperparameters like tree depth to avoid overfitting.  Visualize the tree. d. Implement a Support Vector Machine for any relevant dataset. e. Train a random forest ensemble.  Experiment with the number of trees and feature sampling.  Compare performance to a single decision tree. f. Implement a gradient boosting machine (e.g., XGBoost). Tune hyperparameters  and explore feature importance. |  | |  | |  | |  |
| **5** | . Generative Models a. Implement and demonstrate the working of a Naive Bayesian classifier using a sample data set. Build the model to classify a test sample. b. Implement  Hidden Markov Models using hmmlearn |  | |  | |  | |  |
| **6** | Probabilistic Models a. Implement Bayesian Linear Regression to explore prior and posterior distribution. b. Implement Gaussian Mixture Models for density estimation and  unsupervised clustering |  | |  | |  | |  |
| **7** | Model Evaluation and Hyperparameter Tuning a. Implement cross-validation techniques (k-fold, stratified, etc.) for robust model evaluation b. Systematically explore combinations of hyperparameters to optimize model performance.(use grid  and randomized search) |  | |  | |  | |  |
| **8** | Bayesian Learning a. Implement Bayesian  Learning using inferences |  | |  | |  | |  |

# Practical:-1

**Aim: - 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.

Solution:-

## Step 1: Load the Iris Dataset

import pandas as pd

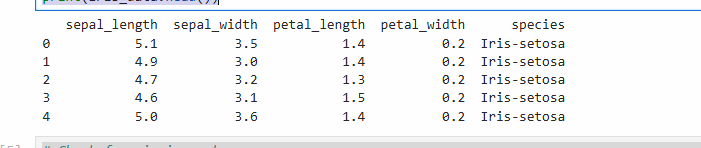
import seaborn as sns

import matplotlib.pyplot as plt # Load the Iris dataset

url = "https://archive.ics.uci.edu/ml/machine-learning-databases/iris/iris.data" column\_names = ['sepal\_length', 'sepal\_width', 'petal\_length', 'petal\_width', 'species'] iris\_data = pd.read\_csv(url, header=None, names=column\_names)

# Display the first few rows of the dataset print(iris\_data.head())

Output:-



## Step 2: Handle Missing Values and Inconsistent Formatting

# Check for missing values

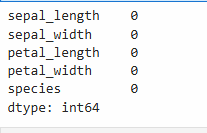
print(iris\_data.isnull().sum())

# If there were missing values, we could handle them like this:

# iris\_data.fillna(iris\_data.mean(), inplace=True) # For numerical columns

# iris\_data['species'].fillna(iris\_data['species'].mode()[0], inplace=True) # For categorical columns

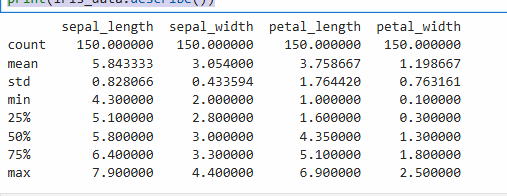
Output:-



## Step 3: Calculate Descriptive Summary Statistics

# Descriptive statistics

print(iris\_data.describe()) Output:-

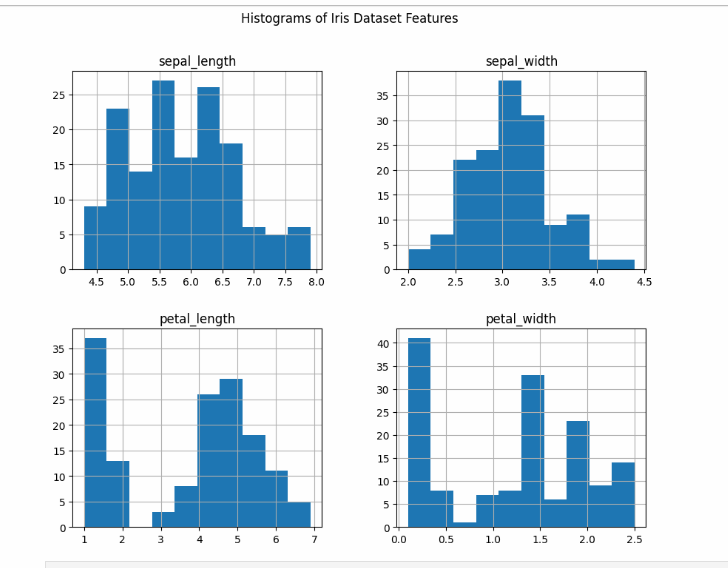


## Step 4: Create Visualizations

# Univariate analysis: Histograms

iris\_data.hist(bins=10, figsize=(10, 8))

plt.suptitle('Histograms of Iris Dataset Features') plt.show()



## Univariate Analysis

# Bivariate analysis: Pairplot

sns.pairplot(iris\_data, hue='species') plt.title('Pairplot of Iris Dataset')

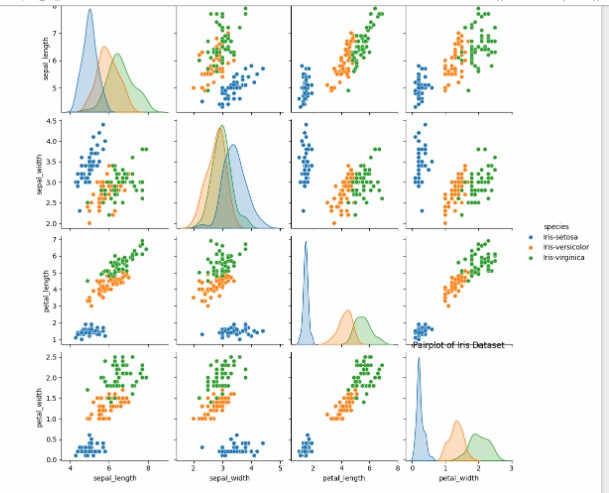
plt.show()

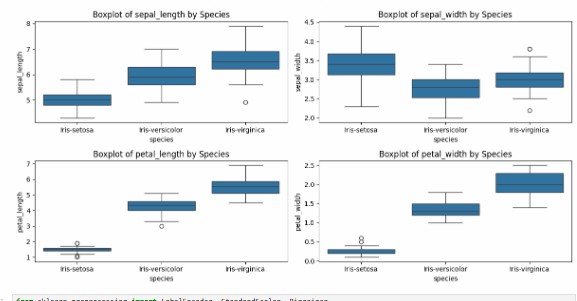
# Boxplot for each feature by species plt.figure(figsize=(12, 6))

for i, feature in enumerate(column\_names[:-1]): plt.subplot(2, 2, i + 1)

sns.boxplot(x='species', y=feature, data=iris\_data) plt.title(f'Boxplot of {feature} by Species')

plt.tight\_layout() plt.show()





## Step 5: Identify Potential Features and Target Variables

In the Iris dataset:

* Features: sepal\_length, sepal\_width, petal\_length, petal\_width
* Target Variable: species

## Step 6: Pre-processing Routines

Now, we will apply label encoding, scaling, and binarization.

from sklearn.preprocessing import LabelEncoder, StandardScaler, Binarizer # Label Encoding

label\_encoder = LabelEncoder()

iris\_data['species'] = label\_encoder.fit\_transform(iris\_data['species']) # Scaling

scaler = StandardScaler()

scaled\_features = scaler.fit\_transform(iris\_data.iloc[:, :-1]) # Exclude target variable

scaled\_iris\_data = pd.DataFrame(scaled\_features, columns=column\_names[:-1]) scaled\_iris\_data['species'] = iris\_data['species']

# Binarization

binarizer = Binarizer(threshold=0.5)

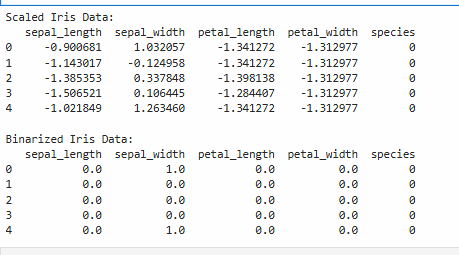
binarized\_features = binarizer.fit\_transform(scaled\_iris\_data.iloc[:, :-1])

binarized\_iris\_data = pd.DataFrame(binarized\_features, columns=column\_names[:-1]) binarized\_iris\_data['species'] = scaled\_iris\_data['species']

# Display the processed datasets print("Scaled Iris Data:")

print(scaled\_iris\_data.head()) print("\nBinarized Iris Data:")

print(binarized\_iris\_data.head()) Output:-



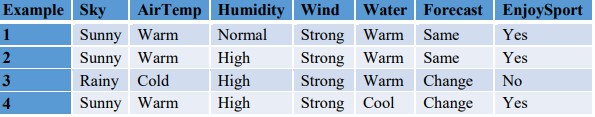
# Practical:-2

**Aim:- 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 a. CSV file and generate the final specific hypothesis. (Create your dataset)

Solution:- FIND-S Algorithm

1. Initialize h to the most specific hypothesis in H
2. For each positive training instance x For each attribute constraint ai in h If the constraint ai is satisfied by x Then do nothing Else replace ai in h by the next more general constraint that is satisfied by x
3. Output hypothesis h Training Examples:



Program:

import csv num\_attributes = 6 a = []

print("\n The Given Training Data Set \n")

with open('enjoysport.csv', 'r') as csvfile: reader = csv.reader(csvfile) for row in reader: a.append (row)

print(row)

print("\n The initial value of hypothesis: ") hypothesis = ['0'] \* num\_attributes print(hypothesis)

for j in range(0,num\_attributes):

hypothesis[j] = a[0][j];

print("\n Find S: Finding a Maximally Specific Hypothesis\n") for i in range(0,len(a)):

if a[i][num\_attributes]=='yes': for j in range(0,num\_attributes): if a[i][j]!=hypothesis[j]: hypothesis[j]='?'

else :

hypothesis[j]= a[i][j]

print(" For Training instance No:{0} the hypothesis is ".format(i),hypothesis) print("\n The Maximally Specific Hypothesis for a given Training Examples :\n") print(hypothesis)

Output:

The Given Training Data Set

['sunny', 'warm', 'normal', 'strong', 'warm', 'same', 'yes']

['sunny', 'warm', 'high', 'strong', 'warm', 'same', 'yes']

['rainy', 'cold', 'high', 'strong', 'warm', 'change', 'no']

['sunny', 'warm', 'high', 'strong', 'cool', 'change', 'yes' The initial value of hypothesis:

['0', '0', '0', '0', '0', '0']

Find S: Finding a Maximally Specific Hypothesis For Training Example No:0 the hypothesis is ['sunny', 'warm', 'normal', 'strong', 'warm', 'same'] For Training Example No:1 the hypothesis is ['sunny', 'warm', '?', 'strong', 'warm', 'same']

For Training Example No:2 the hypothesis is [ 'sunny', 'warm', '?', 'strong', 'warm', 'same'] For Training Example No:3 the hypothesis is ['sunny', 'warm', '?', 'strong', '?', '?']

The Maximally Specific Hypothesis for a given Training Examples: ['sunny', 'warm', '?', 'strong', '?', '?']

# Practical no:- 3

**Aim:- Linear Models**

a. 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

1. Multiple Linear Regression Extend linear regression to multiple features. Handle feature selection and potential multicollinearity.
2. Regularized Linear Models (Ridge, Lasso, ElasticNet) Implement regression variants like LASSO and Ridge on any generated dataset.
3. **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**

Solution:-

## Step 1: Import 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

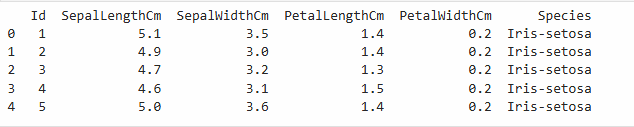
## Step 2: Load the Dataset

# Load the dataset

data = pd.read\_csv('your\_dataset.csv')

# Display the first few rows of the dataset print(data.head())

Output:-



## Step 3: Prepare the Data

Assuming we have a dataset with one independent variable (X) and one dependent variable (y), we need to separate them.

# Define the independent variable (X) and dependent variable (y) X = data[[' SepalLengthCm']]

y = data['SepalWidthCm ']

# Split the 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) # Create a linear regression model

model = LinearRegression() # Fit the model model.fit(X\_train, y\_train) # Get the coefficients intercept = model.intercept\_ slope = model.coef\_[0]

print(f'Intercept: {intercept}') print(f'Slope: {slope}')

## Output:

Intercept: 3.3634418476987857

Slope: -0.0526653589497265

# Make predictions

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

# Calculate R-squared

r\_squared = r2\_score(y\_test, y\_pred)

# Calculate Mean Squared Error

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

print(f'R-squared: {r\_squared}') print(f'Mean Squared Error: {mse}')

output:-

R-squared: 0.02356223365172383

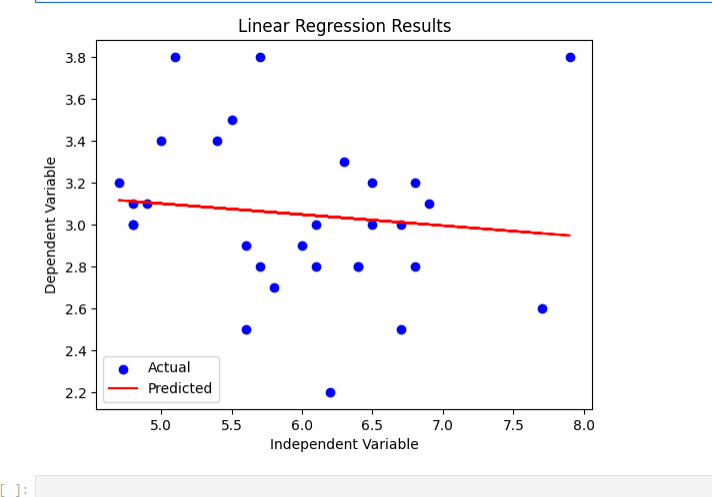
Mean Squared Error: 0.13969569643889335

# Plotting the results

plt.scatter(X\_test, y\_test, color='blue', label='Actual') plt.plot(X\_test, y\_pred, color='red', label='Predicted') plt.xlabel('Independent Variable') plt.ylabel('Dependent Variable')

plt.title('Linear Regression Results') plt.legend()

plt.show()



1. **Multiple Linear Regression Extend linear regression to multiple features. Handle feature selection and potential multicollinearity**.

## Step 1: Import Libraries

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

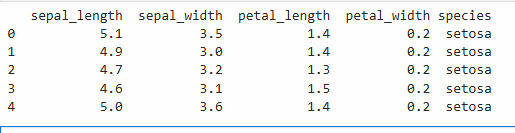
from statsmodels.stats.outliers\_influence import variance\_inflation\_factor import statsmodels.api as sm

## Step 2: Load the Dataset

# Load the Iris dataset

iris = sns.load\_dataset('iris')

# Display the first few rows of the dataset print(iris.head())



# Define the independent variables (X) and dependent variable (y)

X = iris[['sepal\_length', 'sepal\_width', 'petal\_width']] # Using sepal length, sepal width, and petal width

y = iris['petal\_length']

## Step 3: Check for Multicollinearity

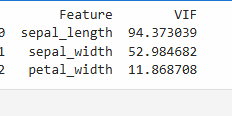
Before fitting the model, we should check for multicollinearity using the Variance Inflation Factor (VIF).

# Calculate VIF for each feature def calculate\_vif(X):

vif\_data = pd.DataFrame() vif\_data["Feature"] = X.columns

vif\_data["VIF"] = [variance\_inflation\_factor(X.values, i) for i in range(X.shape[1])] return vif\_data

vif\_data = calculate\_vif(X) print(vif\_data)



## Step 4: Split the Data

Split the data into training and testing sets. # Split the 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: Fit the Multiple Linear Regression Model

# Create a linear regression model model = LinearRegression()

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

## Step 6: Interpret Coefficients

After fitting the model, we can interpret the coefficients. # Get the coefficients

intercept = model.intercept\_ coefficients = model.coef\_ print(f'Intercept: {intercept}')

for feature, coef in zip(X.columns, coefficients): print(f'Coefficient for {feature}: {coef}')

## output:

Intercept: -0.2621959025887044

Coefficient for sepal\_length: 0.7228146259066678 Coefficient for sepal\_width: -0.6358164939643198 Coefficient for petal\_width: 1.4675240315042082

## Step 7: Make Predictions

Make predictions on the test set. # Make predictions

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

## Step 8: Evaluate Performance

Evaluate the model's performance using R-squared and Mean Squared Error (MSE). # Calculate R-squared

r\_squared = r2\_score(y\_test, y\_pred) # Calculate Mean Squared Error

mse = mean\_squared\_error(y\_test, y\_pred) print(f'R-squared: {r\_squared}') print(f'Mean Squared Error: {mse}')

## Output:-

R-squared: 0.9603293155857664

Mean Squared Error: 0.13001626031382688

## Step 9: Visualize the Results

You can visualize the results by plotting the predicted values against the actual values. # Plotting predicted vs actual values

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

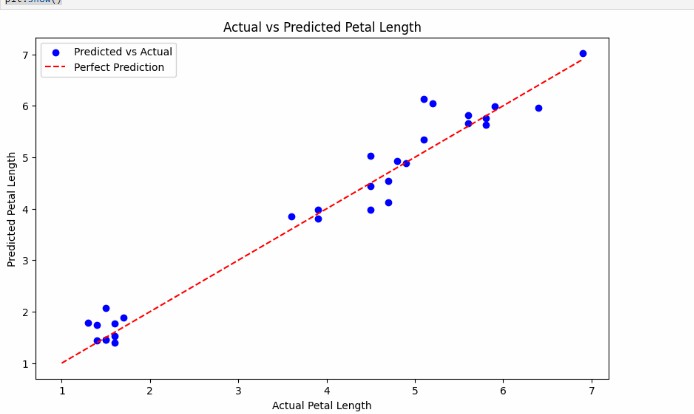
plt.scatter(y\_test, y\_pred, color='blue', label='Predicted vs Actual')

plt.plot([y.min(), y.max()], [y.min(), y.max()], color='red', linestyle='--', label='Perfect Prediction')

plt.xlabel('Actual Petal Length') plt.ylabel('Predicted Petal Length') plt.title('Actual vs Predicted Petal Length') plt.legend()

plt.show()

## Output:-

****

# Regularized Linear Models (Ridge, Lasso, ElasticNet) Implement regression variants like LASSO and Ridge on any generated dataset.

## Step 1: Import Libraries

First, we need to import the necessary libraries.

import pandas as pd import numpy as np

import seaborn as sns

import matplotlib.pyplot as plt

from sklearn.model\_selection import train\_test\_split from sklearn.linear\_model import Ridge, Lasso

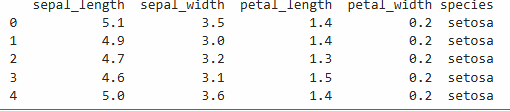
from sklearn.metrics import mean\_squared\_error, r2\_score

## Step 2: Load the Iris Dataset

# Load the Iris dataset

iris = sns.load\_dataset('iris')

# Display the first few rows of the dataset print(iris.head())



## Step 3: Prepare the Data

# Define the independent variables (X) and dependent variable (y)

X = iris[['sepal\_length', 'sepal\_width', 'petal\_width']] # Using sepal length, sepal width, and petal width

y = iris['petal\_length'] # Target variable

## Step 4: Split the Data

Split the data into training and testing sets. # Split the 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: Implement Ridge Regression

# Create a Ridge regression model

ridge\_model = Ridge(alpha=1.0) # You can adjust the alpha parameter for regularization strength

# Fit the model ridge\_model.fit(X\_train, y\_train) # Make predictions

y\_pred\_ridge = ridge\_model.predict(X\_test) # Evaluate performance

ridge\_mse = mean\_squared\_error(y\_test, y\_pred\_ridge) ridge\_r2 = r2\_score(y\_test, y\_pred\_ridge)

print(f'Ridge Regression - Mean Squared Error: {ridge\_mse}') print(f'Ridge Regression - R-squared: {ridge\_r2}')

## Output:-

Ridge Regression - Mean Squared Error: 0.12874617381071274 Ridge Regression - R-squared: 0.9607168455818007

## Step 6: Implement Lasso Regression

Next, we can fit a Lasso regression model using the training data. # Create a Lasso regression model

lasso\_model = Lasso(alpha=0.1) # You can adjust the alpha parameter for regularization strength

# Fit the model

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

# Make predictions

y\_pred\_lasso = lasso\_model.predict(X\_test)

# Evaluate performance

lasso\_mse = mean\_squared\_error(y\_test, y\_pred\_lasso) lasso\_r2 = r2\_score(y\_test, y\_pred\_lasso)

print(f'Lasso Regression - Mean Squared Error: {lasso\_mse}') print(f'Lasso Regression - R-squared: {lasso\_r2}')

## Output:-

Lasso Regression - Mean Squared Error: 0.17275238696285125 Lasso Regression - R-squared: 0.947289628170608

## Step 7: Compare the Results

You can compare the performance of Ridge and Lasso regression.

# Print comparison of results

print("\nComparison of Ridge and Lasso Regression:") print(f"Ridge MSE: {ridge\_mse}, R-squared: {ridge\_r2}") print(f"Lasso MSE: {lasso\_mse}, R-squared: {lasso\_r2}")

Output:-

Comparison of Ridge and Lasso Regression:

Ridge MSE: 0.12874617381071274, R-squared: 0.9607168455818007

Lasso MSE: 0.17275238696285125, R-squared: 0.947289628170608

## Step 8: Visualize the Results

You can visualize the predicted values against the actual values for both models.

# Plotting predicted vs actual values for Ridge plt.figure(figsize=(12, 6))

plt.subplot(1, 2, 1)

plt.scatter(y\_test, y\_pred\_ridge, color='blue', label='Predicted vs Actual')

plt.plot([y.min(), y.max()], [y.min(), y.max()], color='red', linestyle='--', label='Perfect Prediction') plt.xlabel('Actual Petal Length')

plt.ylabel('Predicted Petal Length') plt.title('Ridge Regression: Actual vs Predicted') plt.legend()

# Plotting predicted vs actual values for Lasso plt.subplot(1, 2, 2)

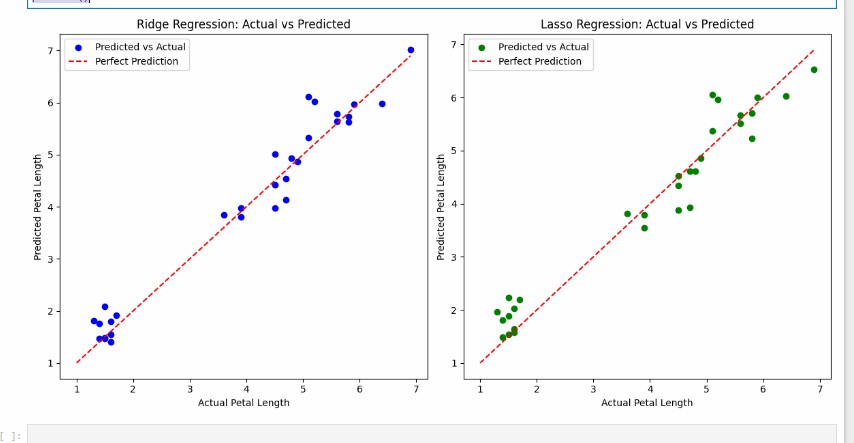
plt.scatter(y\_test, y\_pred\_lasso, color='green', label='Predicted vs Actual')

plt.plot([y.min(), y.max()], [y.min(), y.max()], color='red', linestyle='--', label='Perfect Prediction') plt.xlabel('Actual Petal Length')

plt.ylabel('Predicted Petal Length') plt.title('Lasso Regression: Actual vs Predicted') plt.legend()

plt.tight\_layout() plt.show()

Output:-



# Practical no-4

**Aim:- Logistic Regression**

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

1. 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.

1. Build a decision tree classifier or regressor. Control hyperparameters like tree depth to avoid overfitting. Visualize the tree.
2. Implement a Support Vector Machine for any relevant dataset.
3. Train a random forest ensemble. Experiment with the number of trees and feature sampling. Compare performance to a single decision tree.
4. Implement a gradient boosting machine (e.g., XGBoost). Tune hyperparameters and explore feature importance.

Solution **a)**

import pandas as pd import numpy as np

from sklearn.datasets import load\_iris

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

# Load the Iris dataset iris = load\_iris()

X = iris.data y = iris.target

# For binary classification, we will classify Setosa (0) vs. Not Setosa (1)

# Create a binary target variable

y\_binary = np.where(y == 0, 0, 1) # Setosa = 0, Not Setosa = 1

# Split the dataset into training and testing sets

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y\_binary, test\_size=0.2, random\_state=42) # Train the logistic regression model

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

# Make predictions

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

y\_pred\_proba = model.predict\_proba(X\_test)[:, 1] # Probabilities for the positive class # Calculate evaluation 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}') print(f'Precision: {precision:.2f}') print(f'Recall: {recall:.2f}')

# ROC Curve

fpr, tpr, thresholds = roc\_curve(y\_test, y\_pred\_proba) roc\_auc = auc(fpr, tpr)

# Plotting the ROC curve plt.figure()

plt.plot(fpr, tpr, color='blue', label='ROC curve (area = %0.2f)' % roc\_auc) plt.plot([0, 1], [0, 1], color='red', linestyle='--')

plt.xlim([0.0, 1.0])

plt.ylim([0.0, 1.05]) plt.xlabel('False Positive Rate') plt.ylabel('True Positive Rate')

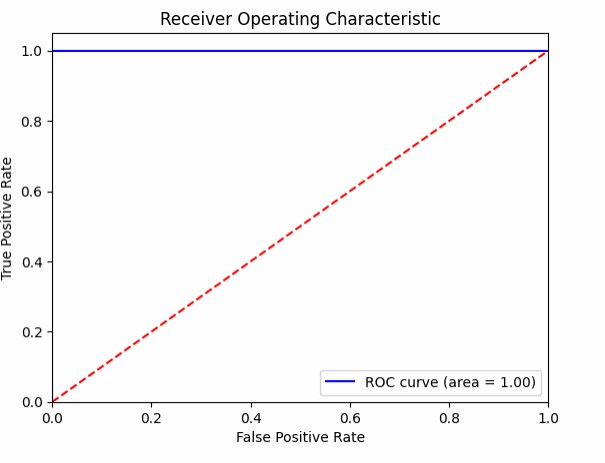
plt.title('Receiver Operating Characteristic') plt.legend(loc='lower right')

plt.show() **Output:-**

Accuracy: 1.00

Precision: 1.00

Recall: 1.00



**b)**

import pandas as pd

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

from sklearn.metrics import accuracy\_score # Load the dataset from a CSV file

# Make sure to adjust the path to where your data.csv file is located data = pd.read\_csv('f:/data.csv')

# Display the first few rows of the dataset print(data.head())

# Assume the last column is the target variable and the rest are features X = data.iloc[:, :-1] # Features

y = data.iloc[:, -1] # Target variable

# Split the dataset 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) # Create and train the KNN model

k = 3 # You can choose the value of k

knn = KNeighborsClassifier(n\_neighbors=k) knn.fit(X\_train, y\_train)

# Make predictions on the test set y\_pred = knn.predict(X\_test)

# Calculate accuracy

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

# Print correct and wrong predictions print("\nCorrect Predictions:")

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

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

print(f"Predicted: {y\_pred[i]}, Actual: {y\_test.iloc[i]}") print("\nWrong Predictions:")

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

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

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

**Output:-**

Feature1 Feature2 Feature3 Target

0 4.9 3.5 1.4 0

1 4.7 3.0 1.4 0

2 4.6 3.2 1.3 0

3 5.0 3.1 1.5 0

4 5.4 3.6 1.4 0

Accuracy: 1.00

Correct Predictions:

Predicted: 1, Actual: 1

Predicted: 0, Actual: 0

Predicted: 0, Actual: 0

Predicted: 1, Actual: 1

Predicted: 0, Actual: 0

Predicted: 0, Actual: 0

**c)**

import pandas as pd

from sklearn.datasets import load\_iris

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

from sklearn.tree import DecisionTreeClassifier, plot\_tree import matplotlib.pyplot as plt

# Load the Iris dataset iris = load\_iris()

X = iris.data y = iris.target

# Split the dataset 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) # Create and train the Decision Tree Classifier

# Control the max depth to avoid overfitting max\_depth = 3

dt\_classifier = DecisionTreeClassifier(max\_depth=max\_depth, random\_state=42) dt\_classifier.fit(X\_train, y\_train)

# Evaluate the model

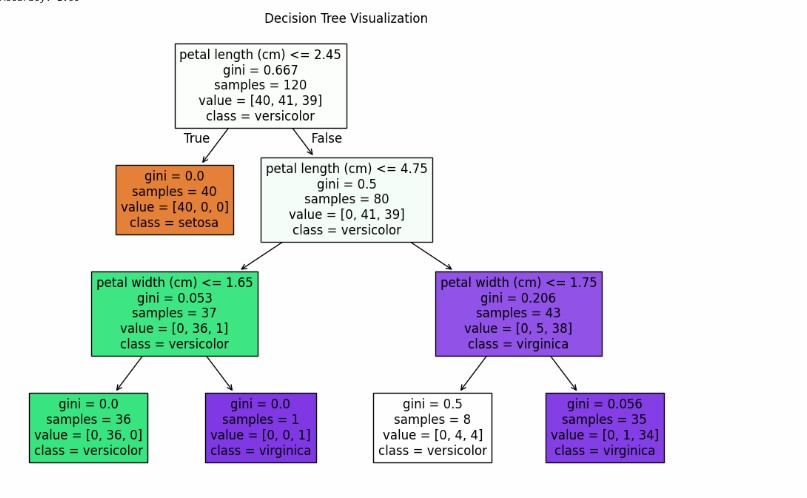
accuracy = dt\_classifier.score(X\_test, y\_test) print(f'Accuracy: {accuracy:.2f}')

# Visualize the Decision Tree plt.figure(figsize=(12, 8))

plot\_tree(dt\_classifier, filled=True, feature\_names=iris.feature\_names, class\_names=iris.target\_names)

plt.title('Decision Tree Visualization') plt.show()

**Output:-**



**d)**

import numpy as np import pandas as pd

import matplotlib.pyplot as plt from sklearn import datasets

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

from sklearn.metrics import accuracy\_score, classification\_report, confusion\_matrix # Load the Iris dataset

iris = datasets.load\_iris()

X = iris.data[:, :2] # Use only the first two features for visualization y = iris.target

# Split the dataset 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) # Create and train the Support Vector Machine Classifier

svm\_classifier = SVC(kernel='linear', random\_state=42) # You can also try 'rbf', 'poly', etc. svm\_classifier.fit(X\_train, y\_train)

# Make predictions

y\_pred = svm\_classifier.predict(X\_test) # Evaluate the model

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

print("\nClassification Report:")

print(classification\_report(y\_test, y\_pred)) print("\nConfusion Matrix:")

print(confusion\_matrix(y\_test, y\_pred)) # Visualize the decision boundary

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

# Create a mesh grid for plotting

x\_min, x\_max = X[:, 0].min() - 1, X[:, 0].max() + 1

y\_min, y\_max = X[:, 1].min() - 1, X[:, 1].max() + 1

xx, yy = np.meshgrid(np.arange(x\_min, x\_max, 0.01), np.arange(y\_min, y\_max, 0.01))

# Predict the class for each point in the mesh grid

Z = svm\_classifier.predict(np.c\_[xx.ravel(), yy.ravel()]) Z = Z.reshape(xx.shape)

# Plot the decision boundary and the training points plt.contourf(xx, yy, Z, alpha=0.8)

plt.scatter(X\_train[:, 0], X\_train[:, 1], c=y\_train, edgecolors='k', marker='o', label='Train') plt.scatter(X\_test[:, 0], X\_test[:, 1], c=y\_test, edgecolors='k', marker='s', label='Test')

plt.xlabel(iris.feature\_names[0]) plt.ylabel(iris.feature\_names[1])

plt.title('SVM Decision Boundary with Iris Dataset') plt.legend()

plt.show()

**Output:**

Accuracy: 0.90

Classification Report:

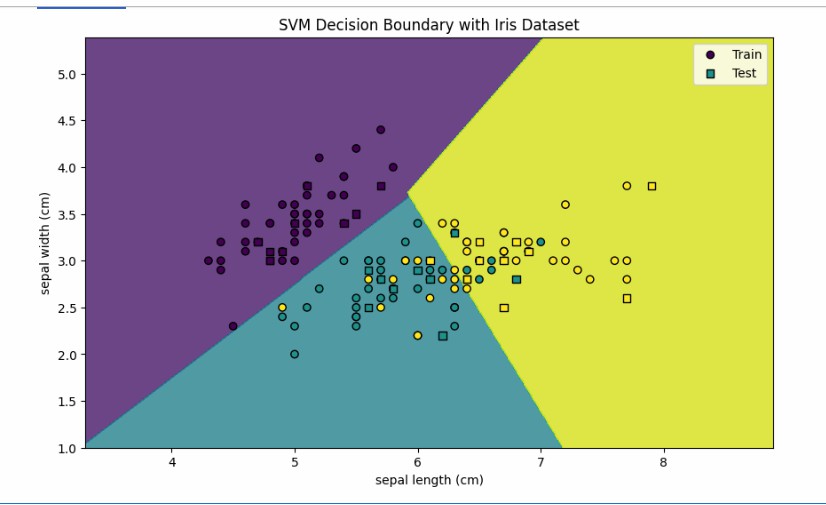
precision recall f1-score support

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 0 | 1.00 | 1.00 | 1.00 | 10 | |
| 1 | 0.88 | 0.78 | 0.82 | 9 | |
| 2 | 0.83 | 0.91 | 0.87 | 11 | |
| accuracy | | 0.90 | | 30 |  |
| macro avg | | 0.90 0.90 | | 0.90 | 30 |
| weighted avg | | 0.90 0.90 | | 0.90 | 30 |

Confusion Matrix: [[10 0 0]

[ 0 7 2]

[ 0 1 10]]



**e)**

import numpy as np import pandas as pd

import matplotlib.pyplot as plt from sklearn import datasets

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, confusion\_matrix # Load the Iris dataset

iris = datasets.load\_iris()

X = iris.data y = iris.target

# Split the dataset 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)

# Train a single Decision Tree model

dt\_classifier = DecisionTreeClassifier(random\_state=42) dt\_classifier.fit(X\_train, y\_train)

# Make predictions with the Decision Tree y\_pred\_dt = dt\_classifier.predict(X\_test)

# Evaluate the Decision Tree model

accuracy\_dt = accuracy\_score(y\_test, y\_pred\_dt) print("Decision Tree Performance:")

print(f'Accuracy: {accuracy\_dt:.2f}') print("\nClassification Report:")

print(classification\_report(y\_test, y\_pred\_dt)) print("\nConfusion Matrix:")

print(confusion\_matrix(y\_test, y\_pred\_dt))

# Train a Random Forest model with different numbers of trees n\_trees = [10, 50, 100]

accuracies\_rf = [] for n in n\_trees:

rf\_classifier = RandomForestClassifier(n\_estimators=n, random\_state=42) rf\_classifier.fit(X\_train, y\_train)

y\_pred\_rf = rf\_classifier.predict(X\_test) accuracy\_rf = accuracy\_score(y\_test, y\_pred\_rf) accuracies\_rf.append(accuracy\_rf)

print(f"\nRandom Forest with {n} trees Performance:") print(f'Accuracy: {accuracy\_rf:.2f}')

print("\nClassification Report:")

print(classification\_report(y\_test, y\_pred\_rf)) print("\nConfusion Matrix:")

print(confusion\_matrix(y\_test, y\_pred\_rf)) # Plotting the performance comparison

plt.figure(figsize=(10, 5))

plt.bar(['Decision Tree'] + [f'Random Forest ({n})' for n in n\_trees],

[accuracy\_dt] + accuracies\_rf, color=['blue', 'green', 'orange', 'red']) plt.ylabel('Accuracy')

plt.title('Model Performance Comparison') plt.ylim(0, 1)

plt.show() **Output:-**

Decision Tree Performance:

Accuracy: 1.00

Classification Report:

precision recall f1-score support

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 0 | 1.00 | 1.00 | 1.00 | 10 | |
| 1 | 1.00 | 1.00 | 1.00 | 9 | |
| 2 | 1.00 | 1.00 | 1.00 | 11 | |
| accuracy | | 1.00 | | 30 |  |
| macro avg | | 1.00 1.00 | | 1.00 | 30 |
| weighted avg | | 1.00 1.00 | | 1.00 | 30 |

Confusion Matrix: [[10 0 0]

[ 0 9 0]

[ 0 0 11]]

Random Forest with 10 trees Performance: Accuracy: 1.00

Classification Report:

precision recall f1-score support

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 0 | 1.00 | 1.00 | 1.00 | 10 | |
| 1 | 1.00 | 1.00 | 1.00 | 9 | |
| 2 | 1.00 | 1.00 | 1.00 | 11 | |
| accuracy | | 1.00 | | 30 |  |
| macro avg | | 1.00 1.00 | | 1.00 | 30 |
| weighted avg | | 1.00 1.00 | | 1.00 | 30 |

Confusion Matrix:

[[10 0 0]

[ 0 9 0]

[ 0 0 11]]

Random Forest with 50 trees Performance: Accuracy: 1.00

Classification Report:

precision recall f1-score support

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 0 | 1.00 | 1.00 | 1.00 | 10 | |
| 1 | 1.00 | 1.00 | 1.00 | 9 | |
| 2 | 1.00 | 1.00 | 1.00 | 11 | |
| accuracy | | 1.00 | | 30 |  |
| macro avg | | 1.00 1.00 | | 1.00 | 30 |
| weighted avg | | 1.00 1.00 | | 1.00 | 30 |

Confusion Matrix: [[10 0 0]

[ 0 9 0]

[ 0 0 11]]

Random Forest with 100 trees Performance: Accuracy: 1.00

Classification Report:

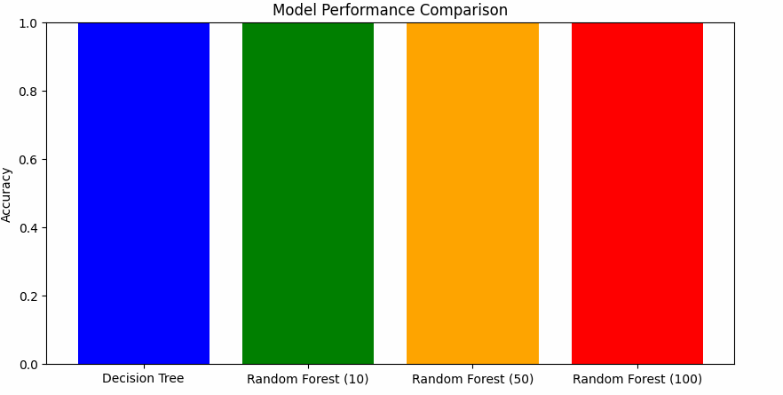
precision recall f1-score support

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 0 | 1.00 | 1.00 | 1.00 | 10 | |
| 1 | 1.00 | 1.00 | 1.00 | 9 | |
| 2 | 1.00 | 1.00 | 1.00 | 11 | |
| accuracy | | 1.00 | | 30 |  |
| macro avg | | 1.00 1.00 | | 1.00 | 30 |
| weighted avg | | 1.00 1.00 | | 1.00 | 30 |

Confusion Matrix: [[10 0 0]

[ 0 9 0]

[ 0 0 11]]



# Practical no-5

**Aim:- Generative Models OC2,OC6**

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

b. Implement Hidden Markov Models using hmmlearn Solution:-

import numpy as np import pandas as pd

from sklearn import datasets

from sklearn.model\_selection import train\_test\_split from sklearn.naive\_bayes import GaussianNB

from sklearn.metrics import accuracy\_score, classification\_report, confusion\_matrix # Load the Iris dataset

iris = datasets.load\_iris()

X = iris.data y = iris.target

# Split the dataset 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) # Create the Naive Bayes classifier

nb\_classifier = GaussianNB() # Train the model

nb\_classifier.fit(X\_train, y\_train) # Make predictions on the test set

y\_pred = nb\_classifier.predict(X\_test) # Evaluate the model

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

print("\nClassification Report:") print(classification\_report(y\_test, y\_pred)) print("\nConfusion Matrix:") print(confusion\_matrix(y\_test, y\_pred))

# Classify a new test sample

# Example: Classifying a new sample with features [5.0, 3.5, 1.5, 0.2]

new\_sample = np.array([[5.0, 3.5, 1.5, 0.2]]) predicted\_class = nb\_classifier.predict(new\_sample)

predicted\_class\_name = iris.target\_names[predicted\_class][0]

print(f"\nPredicted class for the new sample {new\_sample[0]}: {predicted\_class\_name}") **Output:-**

Accuracy: 1.00

Classification Report:

precision recall f1-score support

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 0 | 1.00 | 1.00 | 1.00 | 10 |
| 1 | 1.00 | 1.00 | 1.00 | 9 |
| 2 | 1.00 | 1.00 | 1.00 | 11 |
| accuracy | |  | 1.00 | 30 |
| macro avg | | 1.00 | 1.00 | 1.00 30 |
| weighted avg | | 1.00 | 1.00 | 1.00 30 |

Confusion Matrix:

[[10 0 0]

[ 0 9 0]

[ 0 0 11]]

Predicted class for the new sample [5. 3.5 1.5 0.2]: setosa

**b)**

import numpy as np

from hmmlearn import hmm

# Define the states and observations

states = ["Rainy", "Sunny"] n\_states = len(states)

observations = ["Walk", "Shop", "Clean"] n\_observations = len(observations)

# Create a mapping from observations to integers obs\_map = {obs: i for i, obs in enumerate(observations)} # Sample data: sequences of observations

# Let's say we have the following sequences of observations # Each sequence corresponds to a day of activities

# For example, [Walk, Shop, Clean] corresponds to [0, 1, 2]

X = np.array([[obs\_map["Walk"], obs\_map["Shop"], obs\_map["Clean"]],

[obs\_map["Walk"], obs\_map["Walk"], obs\_map["Shop"]],

[obs\_map["Clean"], obs\_map["Walk"], obs\_map["Walk"]],

[obs\_map["Shop"], obs\_map["Clean"], obs\_map["Walk"]]]) # Reshape the data for HMM

X = np.concatenate([X[i].reshape(-1, 1) for i in range(X.shape[0])]) # Define the model

model = hmm.MultinomialHMM(n\_components=n\_states, n\_iter=100, random\_state=42) # Set the initial state probabilities

model.startprob\_ = np.array([0.6, 0.4]) # Initial probabilities for Rainy and Sunny # Set the transition probabilities

model.transmat\_ = np.array([[0.7, 0.3], # From Rainy to Rainy and Sunny [0.4, 0.6]]) # From Sunny to Rainy and Sunny

# Set the emission probabilities

model.emissionprob\_ = np.array([[0.1, 0.4, 0.5], # Emission probabilities for Rainy [0.6, 0.3, 0.1]]) # Emission probabilities for Sunny

# Fit the model to the data model.fit(X)

# Predict the hidden states for the observed data hidden\_states = model.predict(X)

# Print the results print("Observed Activities:")

print([observations[i] for i in X.flatten()]) print("\nPredicted Hidden States:") print([states[i] for i in hidden\_states])

**Output:-**

Observed Activities:

['Walk', 'Shop', 'Clean', 'Walk', 'Walk', 'Shop', 'Clean', 'Walk', 'Walk', 'Shop', 'Clean', 'Walk']

Predicted Hidden States:

['Sunny', 'Rainy', 'Rainy', 'Rainy', 'Rainy', 'Rainy', 'Rainy', 'Rainy', 'Rainy', 'Rainy', 'Rainy', 'Rainy']

# Practical -6

**Aim:- Probabilistic Models**

1. Implement Bayesian Linear Regression to explore prior and posterior distribution.
2. Implement Gaussian Mixture Models for density estimation and unsupervised clustering Solution:-

**a)**

import numpy as np

import matplotlib.pyplot as plt # Generate synthetic data

np.random.seed(42)

N = 100 # Number of data points D = 1 # Number of features

X = np.random.randn(N, D)

true\_theta = np.array([2.0]) # True parameter sigma = 1.0 # Noise standard deviation

y = X @ true\_theta + sigma \* np.random.randn(N) # Prior parameters

mu\_0 = np.zeros(D) # Prior mean

Sigma\_0 = np.eye(D) \* 10 # Prior covariance # Likelihood parameters

sigma\_sq = sigma\*\*2 # Noise variance # Posterior parameters

Sigma\_p = np.linalg.inv((1 / sigma\_sq) \* X.T @ X + np.linalg.inv(Sigma\_0))

mu\_p = Sigma\_p @ ((1 / sigma\_sq) \* X.T @ y + np.linalg.inv(Sigma\_0) @ mu\_0) # Sample from prior and posterior

prior\_samples = np.random.multivariate\_normal(mu\_0, Sigma\_0, 1000) posterior\_samples = np.random.multivariate\_normal(mu\_p.flatten(), Sigma\_p, 1000) # Plot prior and posterior distributions

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

plt.hist(prior\_samples, bins=50, alpha=0.5, label="Prior", density=True)

plt.hist(posterior\_samples, bins=50, alpha=0.5, label="Posterior", density=True) plt.axvline(true\_theta, color='red', linestyle='--', label="True Theta")

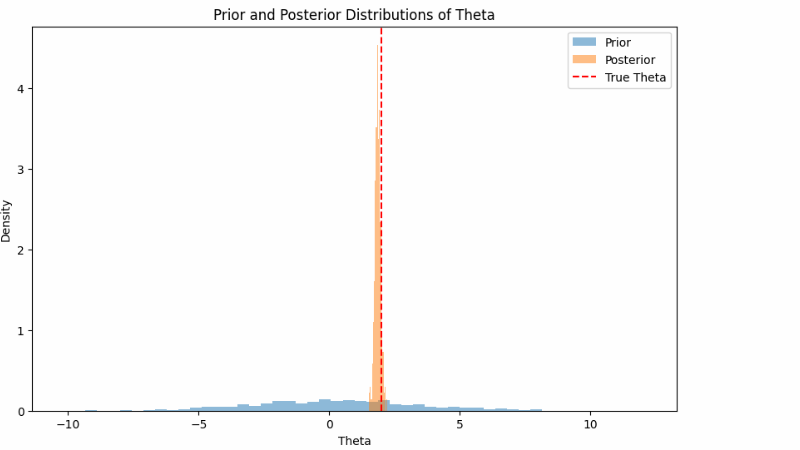
plt.xlabel("Theta") plt.ylabel("Density")

plt.title("Prior and Posterior Distributions of Theta") plt.legend()

plt.show()

# Print posterior mean and covariance print("Posterior Mean:", mu\_p) print("Posterior Covariance:", Sigma\_p)

Solution:-



**b)**

import numpy as np

import matplotlib.pyplot as plt

from scipy.stats import multivariate\_normal

class GaussianMixtureModel:

def init (self, n\_components, max\_iter=100, tol=1e-6): self.n\_components = n\_components # Number of Gaussian components self.max\_iter = max\_iter # Maximum number of iterations

self.tol = tol # Convergence tolerance self.weights = None # Mixing coefficients self.means = None # Means of the Gaussians

self.covariances = None # Covariances of the Gaussians self.responsibilities = None # Responsibilities

def fit(self, X):

n\_samples, n\_features = X.shape # Initialize parameters

self.weights = np.ones(self.n\_components) / self.n\_components

self.means = X[np.random.choice(n\_samples, self.n\_components, replace=False)] self.covariances = [np.eye(n\_features) for \_ in range(self.n\_components)]

log\_likelihood = 0

for iteration in range(self.max\_iter): # E-step: Compute responsibilities

responsibilities = np.zeros((n\_samples, self.n\_components)) for k in range(self.n\_components):

responsibilities[:, k] = self.weights[k] \* multivariate\_normal.pdf( X, mean=self.means[k], cov=self.covariances[k]

)

responsibilities /= responsibilities.sum(axis=1, keepdims=True) # M-step: Update parameters

Nk = responsibilities.sum(axis=0) self.weights = Nk / n\_samples

self.means = np.dot(responsibilities.T, X) / Nk[:, np.newaxis] for k in range(self.n\_components):

diff = X - self.means[k]

self.covariances[k] = np.dot(responsibilities[:, k] \* diff.T, diff) / Nk[k] # Compute log-likelihood

new\_log\_likelihood = 0

for k in range(self.n\_components):

new\_log\_likelihood += self.weights[k] \* multivariate\_normal.pdf(

X, mean=self.means[k], cov=self.covariances[k]

)

new\_log\_likelihood = np.log(new\_log\_likelihood).sum() # Check for convergence

if np.abs(new\_log\_likelihood - log\_likelihood) < self.tol: break

log\_likelihood = new\_log\_likelihood self.responsibilities = responsibilities

def predict(self, X):

# Predict the cluster for each data point

responsibilities = np.zeros((X.shape[0], self.n\_components)) for k in range(self.n\_components):

responsibilities[:, k] = self.weights[k] \* multivariate\_normal.pdf( X, mean=self.means[k], cov=self.covariances[k]

)

return np.argmax(responsibilities, axis=1) # Generate synthetic data

np.random.seed(42) n\_samples = 500

X1 = np.random.multivariate\_normal(mean=[0, 0], cov=[[1, 0], [0, 1]], size=n\_samples)

X2 = np.random.multivariate\_normal(mean=[5, 5], cov=[[1, 0], [0, 1]], size=n\_samples) X = np.vstack((X1, X2))

# Fit GMM

gmm = GaussianMixtureModel(n\_components=2)

gmm.fit(X)

# Predict clusters

labels = gmm.predict(X) # Plot results

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

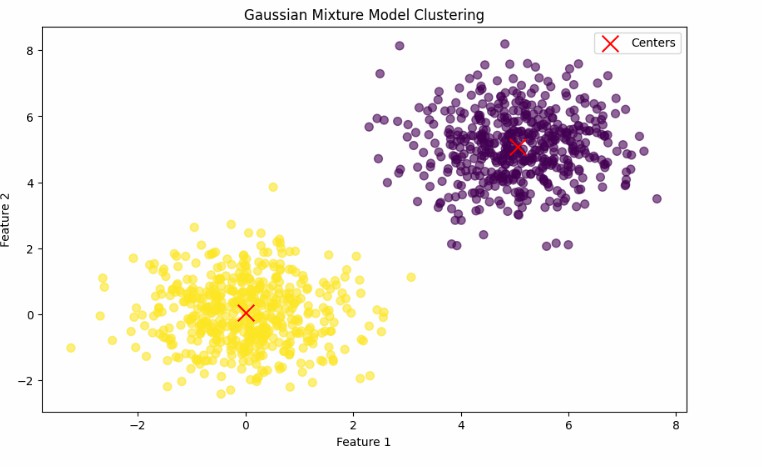
plt.scatter(X[:, 0], X[:, 1], c=labels, cmap='viridis', s=50, alpha=0.6)

plt.scatter(gmm.means[:, 0], gmm.means[:, 1], c='red', marker='x', s=200, label="Centers") plt.title("Gaussian Mixture Model Clustering")

plt.xlabel("Feature 1")

plt.ylabel("Feature 2") plt.legend()

plt.show() **Output:-**



# Practical no-7

**Aim:- Model Evaluation and Hyper parameter Tuning OC3,OC4,OC5**

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

b. Systematically explore combinations of hyper parameters to optimize model performance.(use grid and randomized search)

Solution:-

## K-Fold Cross-Validation

import numpy as np

from sklearn.model\_selection import KFold from sklearn.datasets import load\_iris

from sklearn.ensemble import RandomForestClassifier from sklearn.metrics import accuracy\_score

# Load dataset

data = load\_iris()

X, y = data.data, data.target # Initialize model

model = RandomForestClassifier() # K-Fold Cross-Validation

k = 5

kf = KFold(n\_splits=k, shuffle=True, random\_state=42) accuracies = []

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

X\_train, X\_test = X[train\_index], X[test\_index] y\_train, y\_test = y[train\_index], y[test\_index]

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

predictions = model.predict(X\_test)

accuracy = accuracy\_score(y\_test, predictions) accuracies.append(accuracy)

print(f'K-Fold Cross-Validation Accuracies: {accuracies}') print(f'Mean Accuracy: {np.mean(accuracies)}')

Output:-

K-Fold Cross-Validation Accuracies: [1.0, 0.9666666666666667, 0.9333333333333333,

0.9333333333333333, 0.9666666666666667]

Mean Accuracy: 0.9600000000000002

## Stratified K-Fold Cross-Validation

from sklearn.model\_selection import StratifiedKFold # Stratified K-Fold Cross-Validation

skf = StratifiedKFold(n\_splits=k, shuffle=True, random\_state=42) stratified\_accuracies = []

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

X\_train, X\_test = X[train\_index], X[test\_index] y\_train, y\_test = y[train\_index], y[test\_index]

model.fit(X\_train, y\_train) predictions = model.predict(X\_test)

accuracy = accuracy\_score(y\_test, predictions) stratified\_accuracies.append(accuracy)

print(f'Stratified K-Fold Cross-Validation Accuracies: {stratified\_accuracies}') print(f'Mean Accuracy: {np.mean(stratified\_accuracies)}')

Output:-

Stratified K-Fold Cross-Validation Accuracies: [1.0, 0.9666666666666667, 0.9333333333333333,

0.9666666666666667, 0.9]

Mean Accuracy: 0.9533333333333335

## Hyperparameter Tuning with Cross-Validation

from sklearn.model\_selection import GridSearchCV # Define the parameter grid

param\_grid = {

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

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

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

}

# Initialize GridSearchCV

grid\_search = GridSearchCV(estimator=model, param\_grid=param\_grid, scoring='accuracy', cv=skf, n\_jobs=-1)

# Fit the model

grid\_search.fit(X, y)

# Best parameters and best score

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

print(f'Best Cross-Validation Accuracy: {grid\_search.best\_score\_}') Output:-

Best Parameters: {'max\_depth': 20, 'min\_samples\_split': 10, 'n\_estimators': 50}

Best Cross-Validation Accuracy: 0.9666666666666668

## b) Implementation of Grid Search

import numpy as np

from sklearn.datasets import load\_iris

from sklearn.ensemble import RandomForestClassifier from sklearn.model\_selection import GridSearchCV

# Load dataset

data = load\_iris()

X, y = data.data, data.target # Initialize model

model = RandomForestClassifier() # Define the parameter grid

param\_grid = {

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

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

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

}

# Initialize GridSearchCV

grid\_search = GridSearchCV(estimator=model, param\_grid=param\_grid, scoring='accuracy', cv=5, n\_jobs=-1)

# Fit the model

grid\_search.fit(X, y)

# Best parameters and best score

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

print(f'Best Cross-Validation Accuracy: {grid\_search.best\_score\_}') Output:-

Best Parameters: {'max\_depth': None, 'min\_samples\_split': 2, 'n\_estimators': 200} Best Cross-Validation Accuracy: 0.9666666666666668

## Implementation of Randomized Search

from sklearn.model\_selection import RandomizedSearchCV from scipy.stats import randint

# Define the parameter distribution param\_dist = {

'n\_estimators': randint(50, 300), # Randomly choose between 50 and 300 'max\_depth': [None, 10, 20, 30],

'min\_samples\_split': randint(2, 20) # Randomly choose between 2 and 20

}

# Initialize RandomizedSearchCV

random\_search = RandomizedSearchCV(estimator=model, param\_distributions=param\_dist, n\_iter=100, scoring='accuracy', cv=5, n\_jobs=-1, random\_state=42)

# Fit the model

random\_search.fit(X, y)

# Best parameters and best score

print(f'Best Parameters: {random\_search.best\_params\_}')

print(f'Best Cross-Validation Accuracy: {random\_search.best\_score\_}')

Output:-

Best Parameters: {'max\_depth': 20, 'min\_samples\_split': 12, 'n\_estimators': 252} Best Cross-Validation Accuracy: 0.9666666666666668

# Practical no-8

**Aim:- Implement Bayesian Learning using** inferences. Solution:-

pip install scikit-learn

pip install scikit-optimize pip install matplotlib

pip install bayesian-optimization

from sklearn.model\_selection import cross\_val\_score from sklearn.ensemble import RandomForestClassifier

def objective\_function(n\_estimators, max\_depth):

model = RandomForestClassifier(n\_estimators=int(n\_estimators), max\_depth=int(max\_depth))

scores = cross\_val\_score(model, X\_train, y\_train, cv=5, scoring='accuracy') return scores.mean()

from bayes\_opt import BayesianOptimization

# Define the bounds for hyperparameters pbounds = {

'n\_estimators': (10, 200),

'max\_depth': (1, 30)

}

# Initialize Bayesian Optimization optimizer = BayesianOptimization(

f=objective\_function, pbounds=pbounds,

random\_state=1

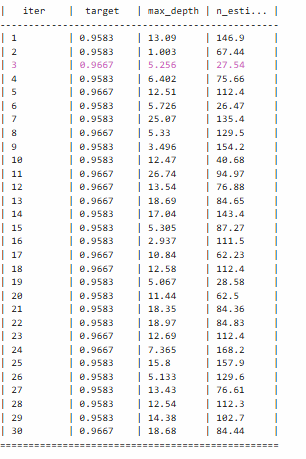
)

optimizer.maximize(

init\_points=5, # Number of random initial points n\_iter=25 # Number of optimization iterations

)

Output:-



## Retrieve the Best Hyperparameters

best\_params = optimizer.max['params']

print(f'Best Hyperparameters: {best\_params}')

Output:-

Best Hyperparameters: {'max\_depth': 5.255920833696278, 'n\_estimators': 27.544333006071582}

[ ]: