SONOPANT DANDEKAR ARTS, V.S. APTE COMMERCE & M.H. MEHTA SCIENCE COLLEGE, PALGHAR, 401404, MAHARASHTRA

DEPARTMENT OF INFORMATION TECHNOLOGY

This is to certify that Mr. / Miss. _

Of M.Sc.(IT) Part-II Semester III, Seat No._ has successfully completed the practicals in the subject of MACHINE LEARNING as per the requirement of University Of Mumbai in part fulfilment for the completion of Degree of Master of Science (Information Technology). It is also to certify that this is the original work of the candidate done during theacademic year 2023-2024.

Internal Examiner **External Examiner**

H.O.D

DEPARTMENT OF I.T

DATE OF SUBMISSION : **COLLEGE SEAL**

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.

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.

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:-

```
   sepal_length  sepal_width  petal_length  petal_width      species
0           5.1          3.5           1.4          0.2  Iris-setosa
1           4.9          3.0           1.4          0.2  Iris-setosa
2           4.7          3.2           1.3          0.2  Iris-setosa
3           4.6          3.1           1.5          0.2  Iris-setosa
4           5.0          3.6           1.4          0.2  Iris-setosa
```

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:-

```
sepal_length    0
sepal_width     0
petal_length    0
petal_width     0
species         0
dtype: int64
```

Step 3: Calculate Descriptive Summary Statistics

Descriptive statistics

print(iris_data.describe())

Output:-

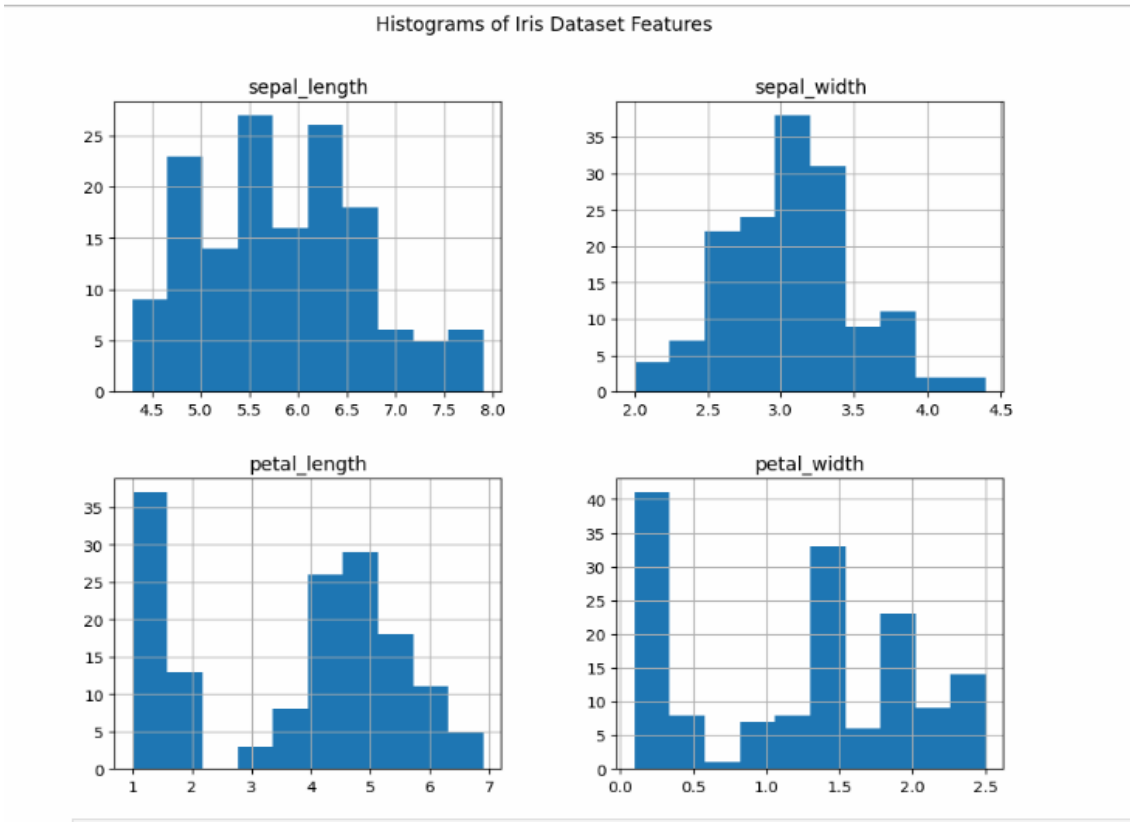
```
       sepal_length  sepal_width  petal_length  petal_width
count    150.000000   150.000000    150.000000   150.000000
mean       5.843333     3.054000      3.758667     1.198667
std        0.828066     0.433594      1.764420     0.763161
min        4.300000     2.000000      1.000000     0.100000
25%        5.100000     2.800000      1.600000     0.300000
50%        5.800000     3.000000      4.350000     1.300000
75%        6.400000     3.300000      5.100000     1.800000
max        7.900000     4.400000      6.900000     2.500000
```

Step 4: Create Visualizations

Univariate analysis: Histograms

iris_data.hist(bins=10, figsize=(10, 8))

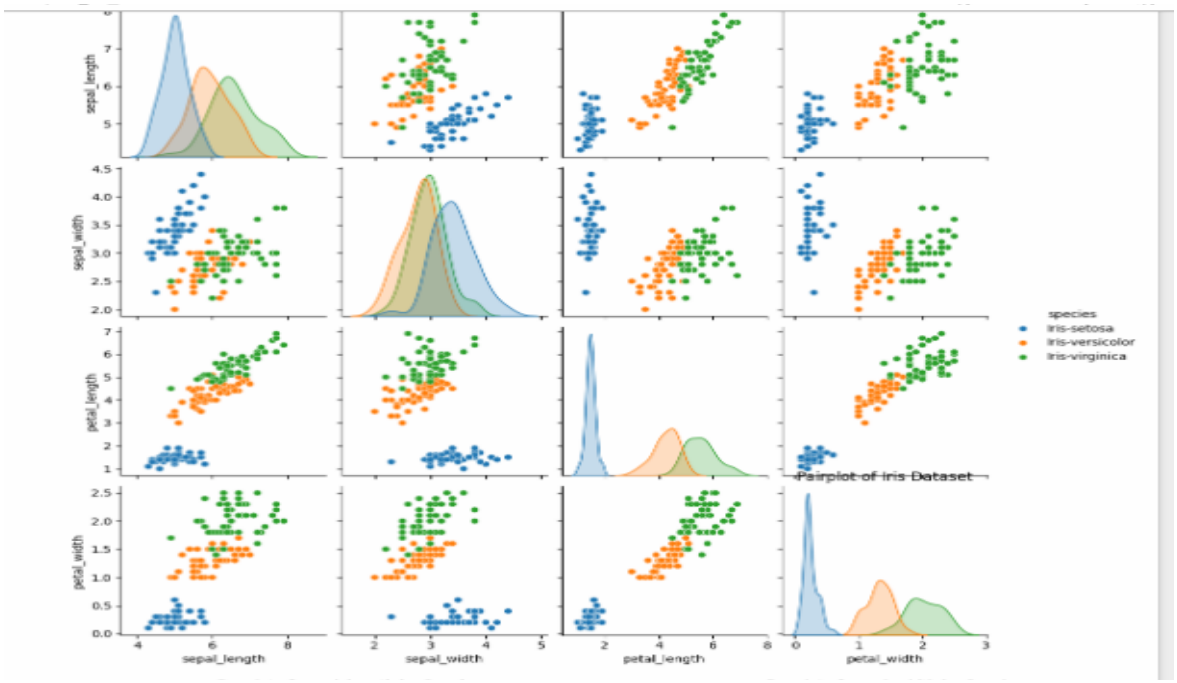
```
plt.suptitle('Histograms of Iris Dataset Features')

plt.show()
```

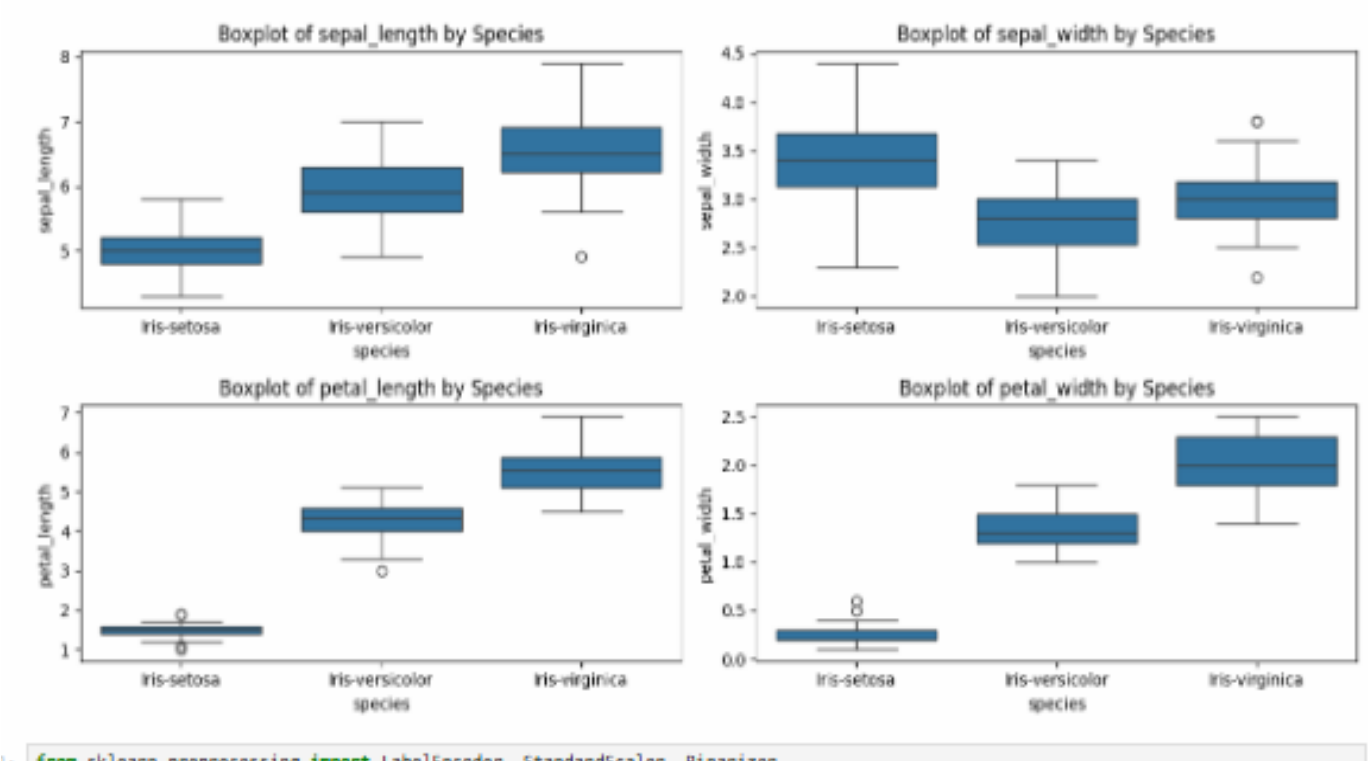


Univariate Analysis

```
# 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:-

```
Scaled Iris Data:
   sepal_length  sepal_width  petal_length  petal_width  species
0     -0.900681     1.032057     -1.341272    -1.312977        0
1     -1.143017    -0.124958     -1.341272    -1.312977        0
2     -1.385353     0.337848     -1.398138    -1.312977        0
3     -1.506521     0.106445     -1.284407    -1.312977        0
4     -1.021849     1.263460     -1.341272    -1.312977        0

Binarized Iris Data:
   sepal_length  sepal_width  petal_length  petal_width  species
0           0.0          1.0           0.0          0.0        0
1           0.0          0.0           0.0          0.0        0
2           0.0          0.0           0.0          0.0        0
3           0.0          0.0           0.0          0.0        0
4           0.0          1.0           0.0          0.0        0
```

Practical:-2

Aim:- 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)

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:

Example	Sky	AirTemp	Humidity	Wind	Water	Forecast	EnjoySport
1	Sunny	Warm	Normal	Strong	Warm	Same	Yes
2	Sunny	Warm	High	Strong	Warm	Same	Yes
3	Rainy	Cold	High	Strong	Warm	Change	No
4	Sunny	Warm	High	Strong	Cool	Change	Yes

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

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.

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:-

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

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}')
```

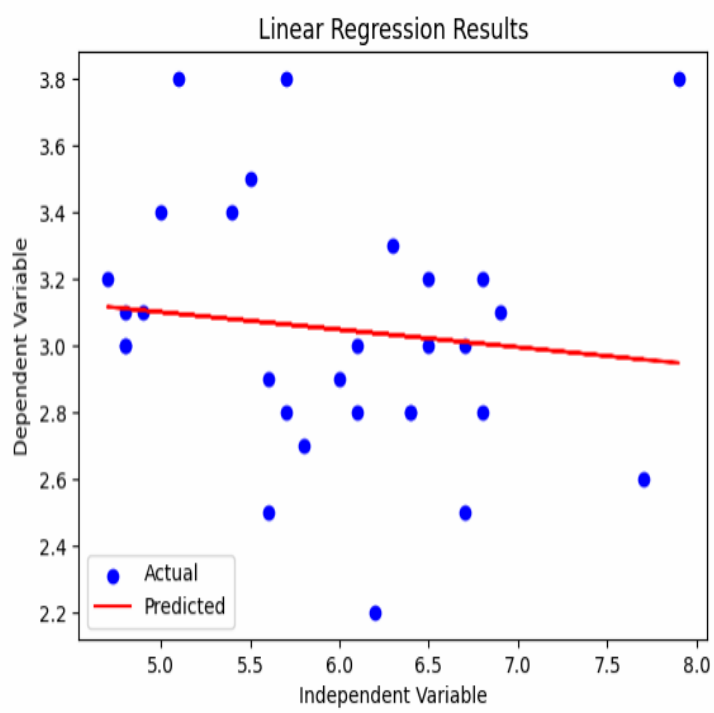
output:

```
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()
```



[ ]:

b) Multiple Linear Regression

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())
```

```
   sepal_length  sepal_width  petal_length  petal_width species
0           5.1          3.5           1.4          0.2  setosa
1           4.9          3.0           1.4          0.2  setosa
2           4.7          3.2           1.3          0.2  setosa
3           4.6          3.1           1.5          0.2  setosa
4           5.0          3.6           1.4          0.2  setosa
```

```
# 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 4: 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)
```

```
        Feature        VIF
0  sepal_length  94.373039
1   sepal_width  52.984682
2   petal_width  11.868708
```

Step 5: 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 6: Fit the Multiple Linear Regression Model

```
# Create a linear regression model

model = LinearRegression()

# Fit the model

model.fit(X_train, y_train)
```

Step 7: 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 8: Make Predictions

Make predictions on the test set.

```
# Make predictions

y_pred = model.predict(X_test)
```

Step 9: 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:-

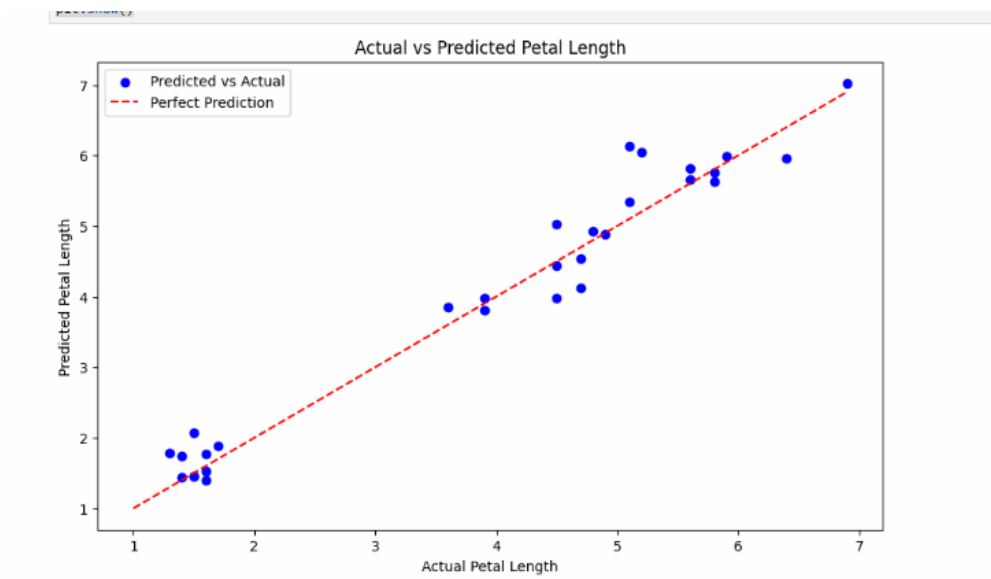
```
R-squared: 0.9603293155857664
Mean Squared Error: 0.13001626031382688
```

Step 10: 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:-



c. Regularized Linear Models (Ridge, Lasso, ElasticNet)

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())
```

	sepal_length	sepal_width	petal_length	petal_width	species
0	5.1	3.5	1.4	0.2	setosa
1	4.9	3.0	1.4	0.2	setosa
2	4.7	3.2	1.3	0.2	setosa
3	4.6	3.1	1.5	0.2	setosa
4	5.0	3.6	1.4	0.2	setosa

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

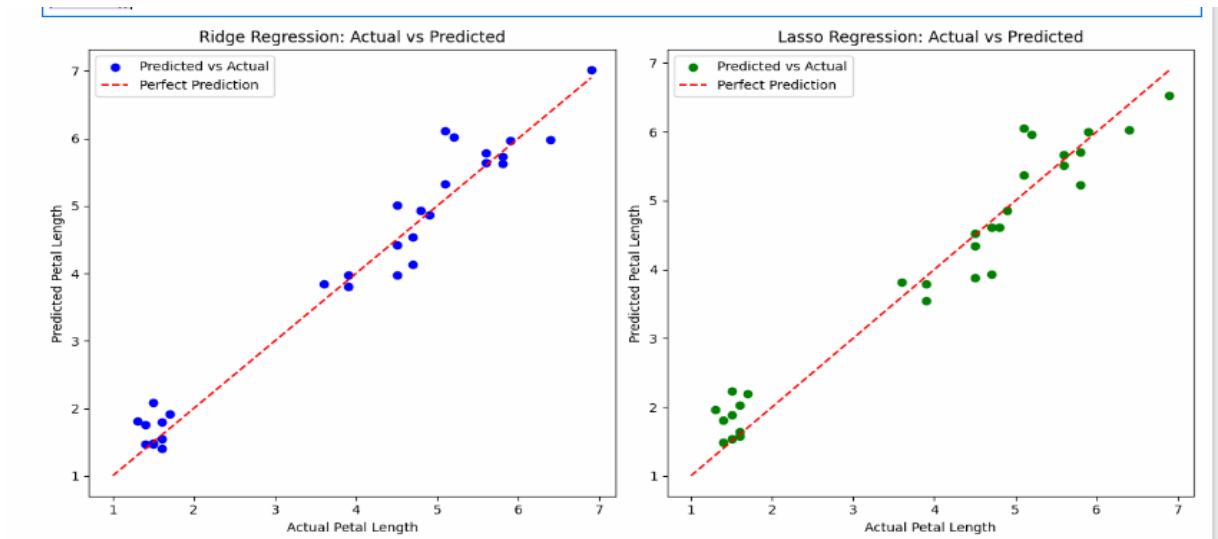
Step 8: Visualize the Results

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.

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.

Solution

a)

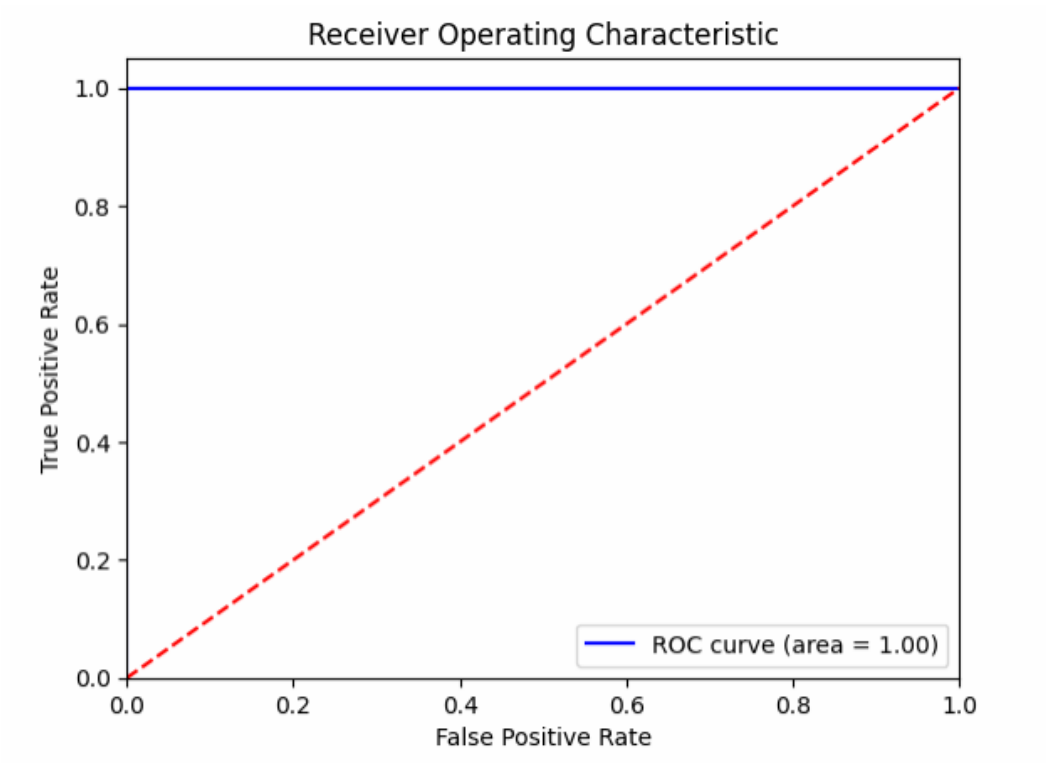
```
import pandas as pd

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:-

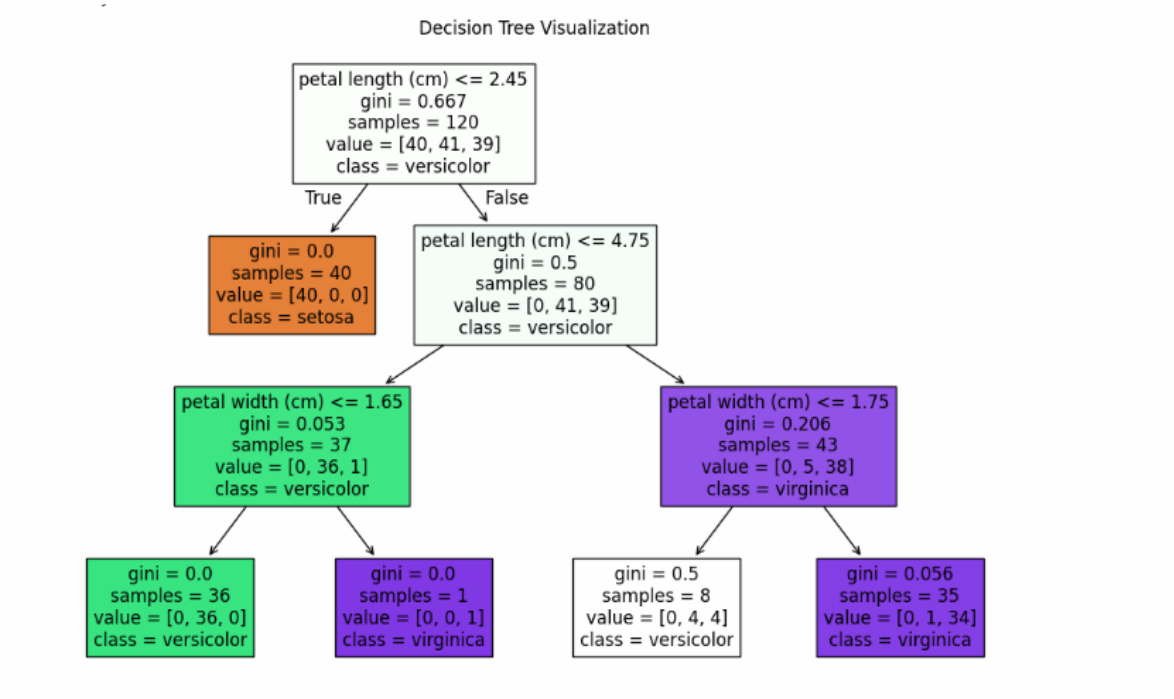
```
Feature1  Feature2  Feature3  Target
0      4.9      3.5      1.4      0
1      4.7      3.0      1.4      0
2      4.6      3.2      1.3      0
3      5.0      3.1      1.5      0
4      5.4      3.6      1.4      0
Accuracy: 1.00

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)

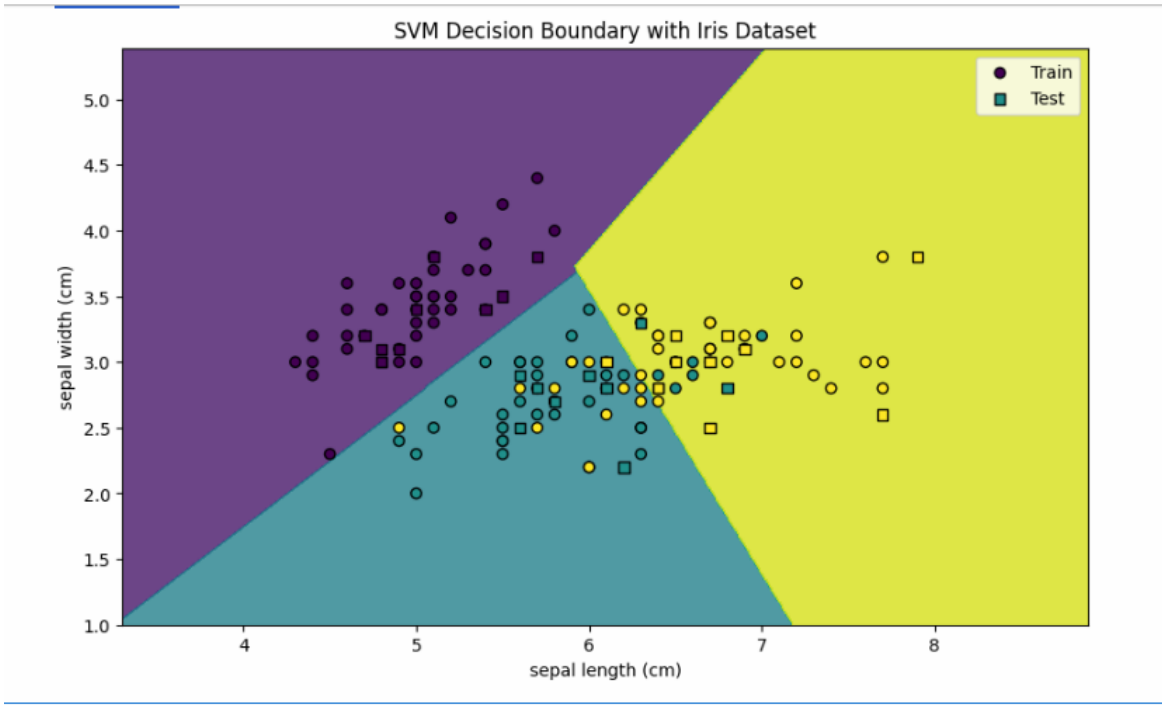
```
import numpy as np

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

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

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

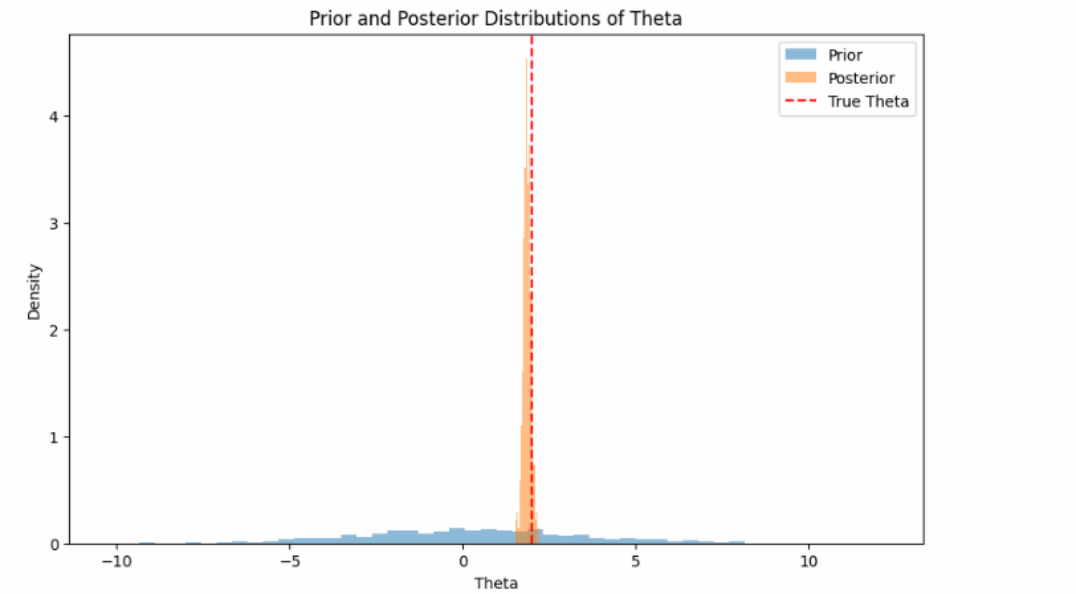
Solution:-

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)

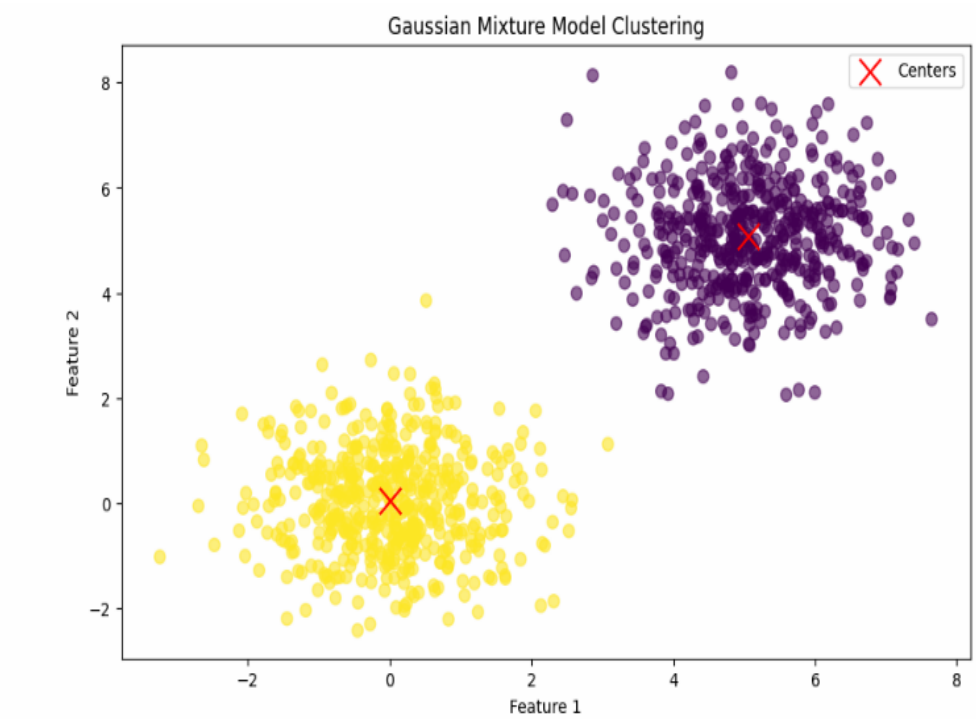
```
import numpy as np

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:-

1. 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

2. 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

3. 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:-

iter	target	max_depth	n_esti...
1	0.9583	13.09	146.9
2	0.9583	1.003	67.44
3	0.9667	5.256	27.54
4	0.9583	6.402	75.66
5	0.9667	12.51	112.4
6	0.9583	5.726	26.47
7	0.9583	25.07	135.4
8	0.9667	5.33	129.5
9	0.9583	3.496	154.2
10	0.9583	12.47	40.68
11	0.9667	26.74	94.97
12	0.9667	13.54	76.88
13	0.9667	18.69	84.65
14	0.9583	17.04	143.4
15	0.9583	5.305	87.27
16	0.9583	2.937	111.5
17	0.9667	10.84	62.23
18	0.9667	12.58	112.4
19	0.9583	5.067	28.58
20	0.9583	11.44	62.5
21	0.9583	18.35	84.36
22	0.9583	18.97	84.83
23	0.9667	12.69	112.4
24	0.9667	7.365	168.2
25	0.9583	15.8	157.9
26	0.9583	5.133	129.6
27	0.9583	13.43	76.61
28	0.9583	12.54	112.3
29	0.9583	14.38	102.7
30	0.9667	18.68	84.44

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}

[ ]: