PRACTICAL – 10

AIM : Write and test a program to provide total numbers of objects present on the page.

```
packagetestscripts;
import
com.thoughtworks.selenium.*;
import org.openqa.selenium.server.*;
import org.testng.annotations.*;
public class exp8 {
   public Selenium selenium;
   public SeleniumServer
   seleniumserver; @BeforeClass
   public void setUp() throws Exception {

      RemoteControlConfiguration rc = new
      RemoteControlConfiguration(); seleniumserver = new
      SeleniumServer(rc);
      selenium = new DefaultSelenium("localhost", 4444, "*firefox", "http://");
      seleniumserver.start();
      selenium.start();
   }
   @Test
   public void testDefaultTNG() throws Exception {
      selenium.open("http://www.google.co.in/");
      selenium.windowMaximize();
      String lc[] = selenium.getAllLinks();
      System.out.println("TOTAL NO OF LINKS=" + lc.length);

      String bc[] = selenium.getAllButtons();
      System.out.println("TOTAL NO OF BUTTONS=" +
      bc.length); String fc[] = selenium.getAllFields();
      System.out.println("TOTAL NO OF INPUT FIELDS=" + fc.length);
   }
   @AfterClass
   public void tearDown() throws Exception
         { selenium.stop();

}}
```

OUTPUT:

TOTAL NO OF LINKS = 41

TOTAL NO OF BUTTONS = 1

TOTAL NO OF INPUT FIELDS=3

PRACTICAL-11

Aim: Write and test a program to update 10 student records into table into Excel file.

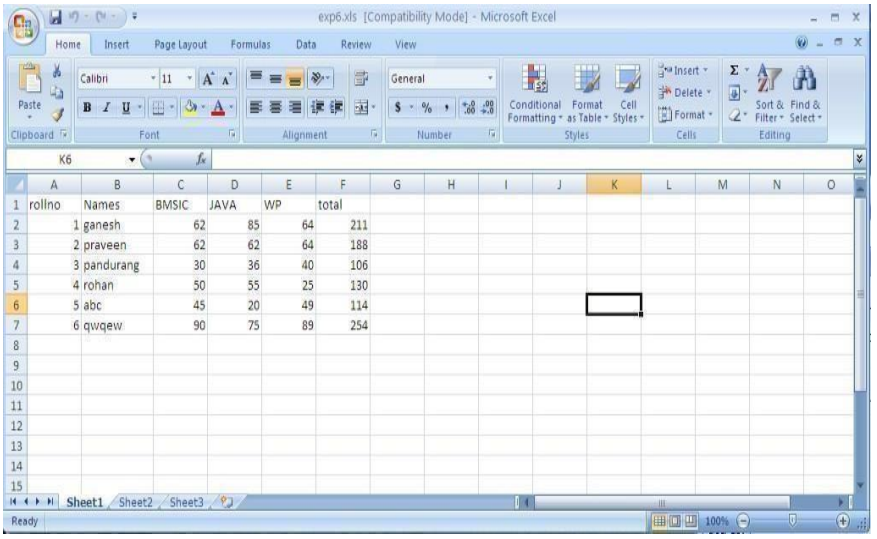
CODE: FileInputStream("D:\\exp6.xls");

```
Workbook w = Workbook.getWorkbook(fi);
Sheet s = w.getSheet(0);
String a[][] = new String[s.getRows()][s.getColumns()];
FileOutputStream fo = new
FileOutputStream("D://exp6Result.xls"); WritableWorkbook wwb
```

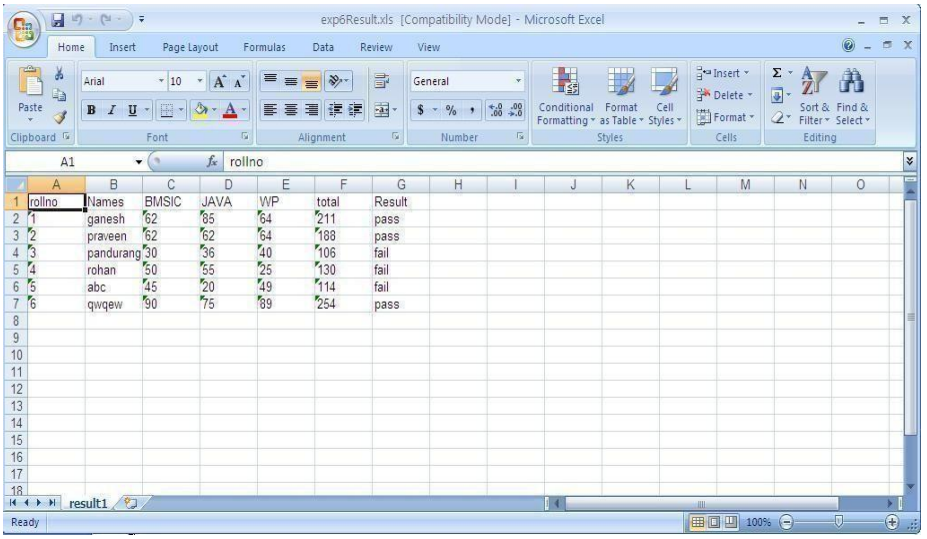
```
import java.io.FileInputStream;
import
java.io.FileOutputStream;
import jxl.Sheet;
import jxl.Workbook;
import
jxl.write.Label;
import  jxl.write.WritableSheet;
importjxl.write.WritableWorok;
import org.testng.annotations.*;
public class newj {
   @BeforeClass
   public void setUp() throws Exception
   {} @Test
   public void testImportexport1() throws Exception {
      FileInputStream fi = new
      = Workbook.createWorkbook(fo); WritableSheet ws =
      wwb.createSheet("result1", 0);
      for (int i = 0; i < s.getRows(); i++)
         for (int j = 0; j < s.getColumns(); j++) {
            a[i][j] = s.getCell(j, i).getContents();
            Label l2 = new Label(j, i, a[i][j]);
            ws.addCell(l2);
            Label l1 = new Label(6, 0, "Result");
```

```
            ws.addCell(l1);

        }

    for (int i = 1; i < s.getRows(); i++) {

        for (int j = 2; j < s.getColumns(); j++) {
            a[i][j] = s.getCell(j, i).getContents();
            intx = Integer.parseInt(a[i][j]);
            if (x > 35) {

                Label l1 = new Label(6, i, "pass");
                ws.addCell(l1);
            }

            Else

            {

                Label l1 = new Label(6, i, "fail");
                ws.addCell(l1);
                break;

            }}}

    wwb.write();

    wwb.close();

}}
```

INPUT:



OUTPUT:



rollno	Names	BMSIC	JAVA	WP	total	Result
1	ganesh	62	85	64	211	pass
2	praveen	62	62	64	188	pass
3	pandurang	30	36	40	106	fail
4	rohan	50	55	25	130	fail
5	abc	45	20	49	114	fail
6	qwqew	90	75	89	254	pass

PRACTICAL – 10

AIM : Write and test a program to provide total numbers of objects present on the page.

```
packagetestscripts;
import
com.thoughtworks.selenium.*;
import org.openqa.selenium.server.*;
import org.testng.annotations.*;
public class exp8 {
   public Selenium selenium;
   public SeleniumServer
   seleniumserver; @BeforeClass
   public void setUp() throws Exception {

      RemoteControlConfiguration rc = new
      RemoteControlConfiguration(); seleniumserver = new
      SeleniumServer(rc);
      selenium = new DefaultSelenium("localhost", 4444, "*firefox", "http://");
      seleniumserver.start();
      selenium.start();
   }
   @Test
   public void testDefaultTNG() throws Exception {
      selenium.open("http://www.google.co.in/");
      selenium.windowMaximize();
      String lc[] = selenium.getAllLinks();
      System.out.println("TOTAL NO OF LINKS=" +lc.length);

      String bc[] = selenium.getAllButtons();
      System.out.println("TOTAL NO OF BUTTONS=" +
      bc.length); String fc[] = selenium.getAllFields();
      System.out.println("TOTAL NO OF INPUT FIELDS=" + fc.length);
   }
   @AfterClass
   public void tearDown() throws Exception
         { selenium.stop();

}}
```

OUTPUT:

TOTAL NO OF LINKS = 41

TOTAL NO OF BUTTONS = 1

TOTALNO OF INPUT FIELDS=3

PRACTICAL-11

Aim: Write and test a program to update 10 student records into table into Excel file.

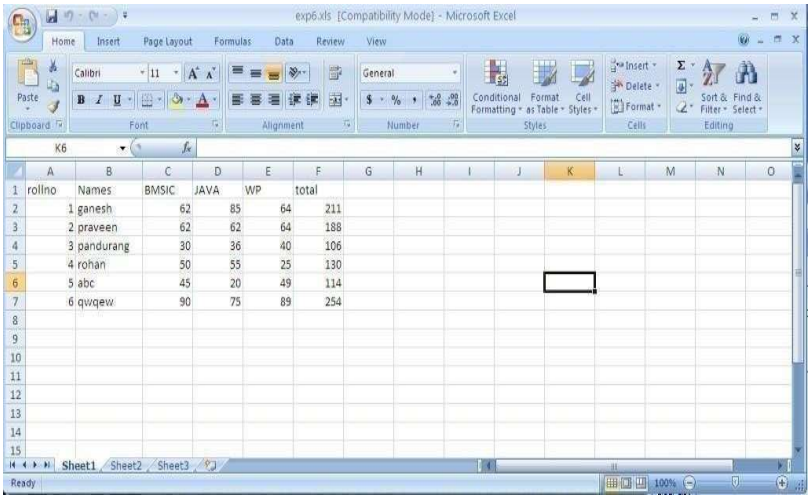
CODE: FileInputStream("D:\\exp6.xls");

```
Workbook w = Workbook.getWorkbook(fi);
Sheet s = w.getSheet(0);
String a[][] = new String[s.getRows()][s.getColumns()];
FileOutputStream fo = new
FileOutputStream("D://exp6Result.xls"); WritableWorkbook wwb
```

```
import java.io.FileInputStream;
import
java.io.FileOutputStream;
import jxl.Sheet;
import jxl.Workbook;
import
jxl.write.Label;
import  jxl.write.WritableSheet;
importjxl.write.WritableWorok;
import org.testng.annotations.*;
public class newj {
   @BeforeClass
   public void setUp() throws Exception
   {} @Test
   public void testImportexport1() throws Exception {
      FileInputStream fi = new
      = Workbook.createWorkbook(fo); WritableSheet ws =
      wwb.createSheet("result1", 0);
      for (int i = 0; i < s.getRows(); i++)
         for (int j = 0; j < s.getColumns(); j++) {
            a[i][j] = s.getCell(j, i).getContents();
            Label l2 = new Label(j, i, a[i][j]);
            ws.addCell(l2);
            Label l1 = new Label(6, 0, "Result");
```

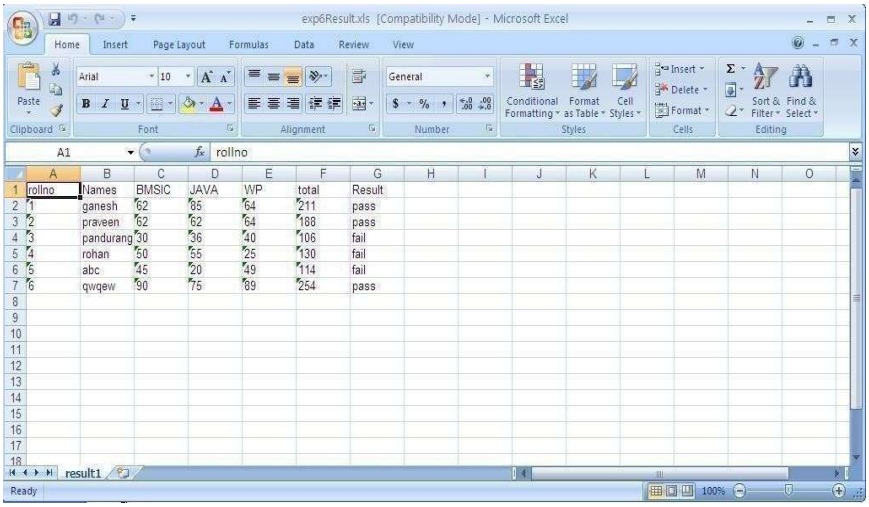
```
            ws.addCell(l1);

        }

    for(int i = 1; i < s.getRows(); i++) {

        for (int j = 2; j < s.getColumns(); j++) {
            a[i][j] = s.getCell(j, i).getContents();
            intx = Integer.parseInt(a[i][j]);
            if(x > 35) {

                Label l1 = new Label(6, i, "pass");
                ws.addCell(l1);
            }

            Else

            {

                Label l1 = new Label(6, i, "fail");
                ws.addCell(l1);
                break;

            }}}

    wwb.write();

    wwb.close();

}}
```

INPUT:



rollno	Names	BMSIC	JAVA	WP	total
1	ganesh	62	85	64	211
2	praveen	62	62	64	188
3	pandurang	30	36	40	106
4	rohan	50	55	25	130
5	abc	45	20	49	114
6	qwqew	90	75	89	254

OUTPUT:



rollno	Names	BMSIC	JAVA	WP	total	Result
1	ganesh	62	85	64	211	pass
2	praveen	62	62	64	188	pass
3	pandurang	30	36	40	106	fail
4	rohan	50	55	25	130	fail
5	abc	45	20	49	114	fail
6	qwqew	90	75	89	254	pass

PRACTICAL – 10

AIM : Write and test a program to provide total numbers of objects present on the page.

```
packagetestscripts;
import
com.thoughtworks.selenium.*;
import org.openqa.selenium.server.*;
import org.testng.annotations.*;
public class exp8 {
   public Selenium selenium;
   public SeleniumServer
   seleniumserver; @BeforeClass
   public void setUp() throws Exception {

      RemoteControlConfiguration rc = new
      RemoteControlConfiguration(); seleniumserver = new
      SeleniumServer(rc);
      selenium = new DefaultSelenium("localhost", 4444, "*firefox", "http://");
      seleniumserver.start();
      selenium.start();
   }
   @Test
   public void testDefaultTNG() throws Exception {
      selenium.open("http://www.google.co.in/");
      selenium.windowMaximize();
      String lc[] = selenium.getAllLinks();
      System.out.println("TOTAL NO OF LINKS=" +lc.length);

      String bc[] = selenium.getAllButtons();
      System.out.println("TOTAL NO OF BUTTONS=" +
      bc.length); String fc[] = selenium.getAllFields();
      System.out.println("TOTAL NO OF INPUT FIELDS=" + fc.length);
   }
   @AfterClass
   public void tearDown() throws Exception
         { selenium.stop();

}}
```

OUTPUT:

TOTAL NO OF LINKS = 41

TOTAL NO OF BUTTONS = 1

TOTALNO OF INPUT FIELDS=3

PRACTICAL-11

Aim: Write and test a program to update 10 student records into table into Excel file.

CODE: FileInputStream("D:\\exp6.xls");

Workbook w = Workbook.getWorkbook(fi);

Sheet s = w.getSheet(0);

String a[][] = new String[s.getRows()][s.getColumns()];

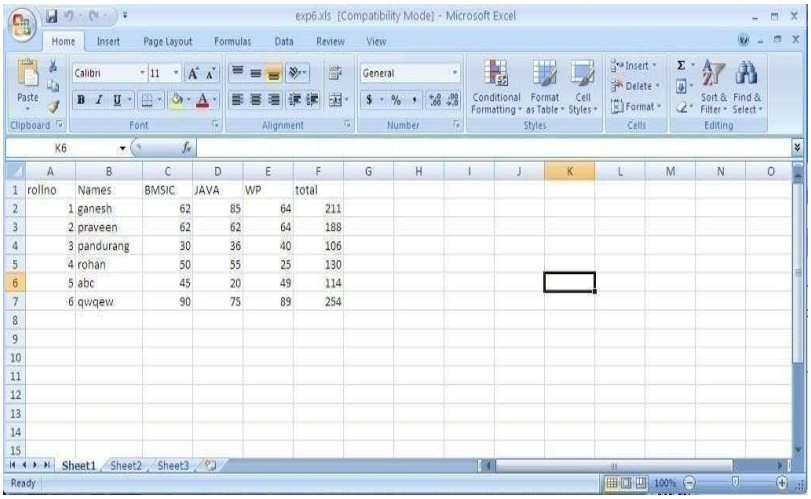
FileOutputStream fo = new

FileOutputStream("D://exp6Result.xls"); WritableWorkbook wwb

```
import java.io.FileInputStream;
import
java.io.FileOutputStream;
import jxl.Sheet;
import jxl.Workbook;
import
jxl.write.Label;
import  jxl.write.WritableSheet;
importjxl.write.WritableWorok;
import org.testng.annotations.*;
public class newj {
   @BeforeClass
   public void setUp() throws Exception
   {} @Test
   public void testImportexport1() throws Exception {
      FileInputStream fi = new
      = Workbook.createWorkbook(fo); WritableSheet ws =
      wwb.createSheet("result1", 0);
      for (int i = 0; i < s.getRows(); i++)
         for (int j = 0; j < s.getColumns(); j++) {
            a[i][j] = s.getCell(j, i).getContents();
            Label l2 = new Label(j, i, a[i][j]);
            ws.addCell(l2);
            Label l1 = new Label(6, 0, "Result");
```

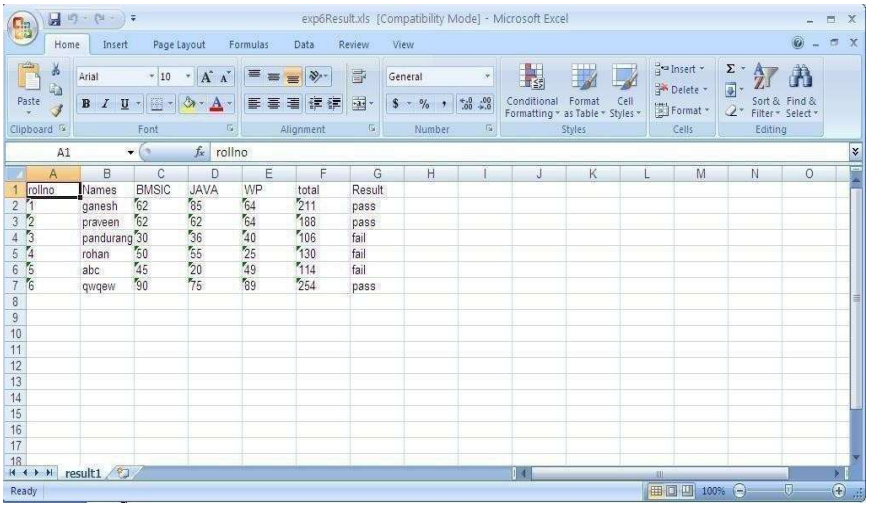
```
            ws.addCell(l1);

        }

    for(int i = 1; i < s.getRows(); i++) {

        for (int j = 2; j < s.getColumns(); j++) {
            a[i][j] = s.getCell(j, i).getContents();
            intx = Integer.parseInt(a[i][j]);
            if(x > 35) {

                Label l1 = new Label(6, i, "pass");
                ws.addCell(l1);
            }

            Else

            {

                Label l1 = new Label(6, i, "fail");
                ws.addCell(l1);
                break;

            }}}

    wwb.write();

    wwb.close();

}}
```

INPUT:



rollno	Names	BMSIC	JAVA	WP	total
1	ganesh	62	85	64	211
2	praveen	62	62	64	188
3	pandurang	30	36	40	106
4	rohan	50	55	25	130
5	abc	45	20	49	114
6	qwqew	90	75	89	254

OUTPUT:



rollno	Names	BMSIC	JAVA	WP	total	Result
1	ganesh	62	85	64	211	pass
2	praveen	62	62	64	188	pass
3	pandurang	30	36	40	106	fail
4	rohan	50	55	25	130	fail
5	abc	45	20	49	114	fail
6	qwqew	90	75	89	254	pass

Parul University

FACULTY OF ENGINEERING AND TECHNOLOGY
BACHELOR OF TECHNOLOGY

Pattern recognition

(303105482)

7TH SEMESTER

Computer Science & Engineering Department

Lab Manual

Parul® University
VADODARA, GUJARAT

NAAC A++

FACULTY OF ENGINEERING AND TECHNOLOGY
Subject Name: Pattern Recognition Laboratory
Subject Code: 30105482
B.Tech. 4rd year 7TH semester

CERTIFICATE

This is to certify that Student **MD SAMEER,** with Enrollment No. **2203031240809** has Successfully completed his Laboratory experiments in **Pattern Recognition (303105482)** from the department of **ARTIFICIAL INTELLINGENCE AND DATA SCIENCE** during the academic year **2024-2025**.

योगः कर्मसु कौशलम्

PARUL UNIVERSITY

Date of Submission: ..... Staff In Charge: ……..

Head of Department: .....

TABLE OF CONTENT

Practical No: 1

Aim:. Implementation of Gradient Descent.

What is Gradient Descent?

Gradient Descent stands as a cornerstone orchestrating the intricate dance of model optimization. At its core, it is a numerical optimization algorithm that aims to find the optimal parameters—weights and biases—of a neural network by minimizing a defined cost function.

Gradient Descent (GD) is a widely used optimization algorithm in machine learning and deep learning that minimises the cost function of a neural network model during training. It works by iteratively adjusting the weights or parameters of the model in the direction of the negative gradient of the cost function until the minimum of the cost function is reached.

The learning happens during the backpropagation while training the neural network-based model. There is a term known as Gradient Descent, which is used to optimize the weight and biases based on the cost function. The cost function evaluates the difference between the actual and predicted outputs.

Gradient Descent is a fundamental optimization algorithm in machine learning used to minimize the cost or loss function during model training.

- It iteratively adjusts model parameters by moving in the direction of the steepest decrease in the cost function.
- The algorithm calculates gradients, representing the partial derivatives of the cost function concerning each parameter.

These gradients guide the updates, ensuring convergence towards the optimal parameter values that yield the lowest possible cost.

Gradient Descent is versatile and applicable to various machine learning models, including linear regression and neural networks. Its efficiency lies in navigating the parameter space efficiently, enabling models to learn patterns and make accurate predictions. Adjusting the learning rate is crucial to balance convergence speed and avoiding overshooting the optimal solution.

Gradient descent is an iterative optimization algorithm for finding the local minimum of a function.

To find the local minimum of a function using gradient descent, we must take steps proportional to the negative of the gradient (move away from the gradient) of the function at the current point. If we take steps proportional to the positive of the gradient (moving towards the gradient), we will approach a local maximum of the function, and the procedure is called **Gradient Ascent.**

DATASET TAKEN:- Globalisation Index

The dataset contains 167 rows and 14 columns, with the following features:

1. Country: The name of the country (categorical feature).
2. Average Score: The average score of various factors for each country (numeric, float).
3. Safety Security: The score related to safety and security (numeric, float).
4. Personal Freedom: The score for personal freedom in each country (numeric, float).
5. Governance: The score for governance (numeric, float).
6. Social Capital: The score for social capital (numeric, float).
7. Investment Environment: The score related to the investment environment (numeric, float).
8. Enterprise Conditions: The score for enterprise conditions (numeric, float).
9. Market Access Infrastructure: The score for market access and infrastructure (numeric, float).
10. Economic Quality: The score for economic quality (numeric, float).
11. Living Conditions: The score for living conditions (numeric, float).
12. Health: The score for health conditions (numeric, float).
13. Education: The score for education (numeric, float).
14. Natural Environment: The score for natural environment quality (numeric, float).

All features except for "Country" are numeric and indicate scores related to various aspects of each country's conditions.

Program:

```
import numpy as np
import matplotlib.pyplot as plt

# Generatesomedata
np.random.seed(42)
X = 2 * np.random.rand(100, 1)
y = 4 + 3 * X + np.random.randn(100, 1)

# Function to compute the cost def
compute_cost(X, y, theta):
        m = len(y)
        predictions=X.dot(theta)
        cost = (1/2*m) * np.sum(np.square(predictions - y)) return cost

# Function to perform gradient descent
def gradient_descent(X, y, theta, learning_rate, iterations): m = len(y)
        cost_history=np.zeros(iterations)

        for i in range(iterations): predictions= X.dot(theta)
                theta = theta - (1/m) * learning_rate * (X.T.dot(predictions - cost_history[i] =

y))          compute_cost(X, y, theta)
```

retu rn theta, cost_history

Add x0 = 1 to each instance X_b =
np.c_[np.ones((100, 1)), X]

# Initialize theta
theta = np.random.randn(2, 1)

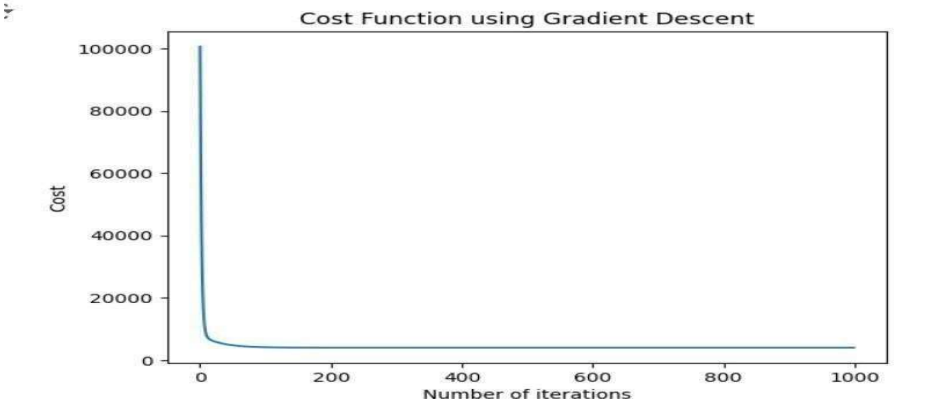
Sethyperparameters learning_rate =
0.1
iterations= 1000

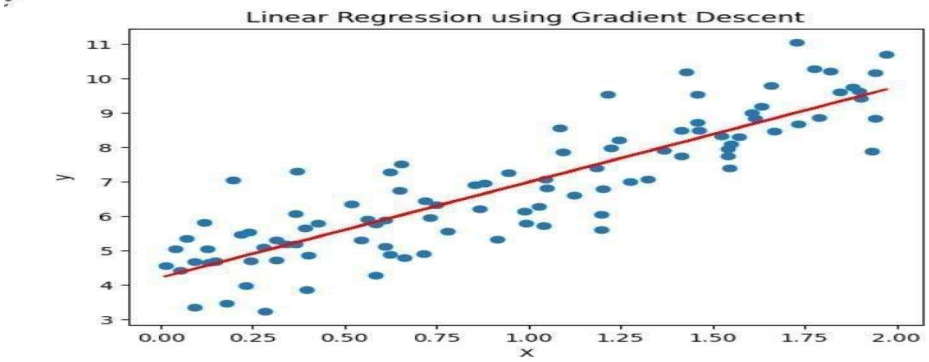
Perform gradient descent
theta, cost_history = gradient_descent(X_b, y, theta, learning_rate, iterations)

Plot the cost function plt.plot(range(iterations), cost_history)

```
plt.xlabel("Number of iterations")
plt.ylabel("Cost")
plt.title("Cost Function using Gradient Descent")
plt.show()

# Plot the data and the regression line
plt.scatter(X, y)
plt.plot(X, X_b.dot(theta), color='red')
plt.xlabel("X")
plt.ylabel("y")
plt.title("Linear Regression using Gradient Descent")
t.show()

print("Theta (parameters):", theta)
print("Final cost:", cost_history[-1])
```

Output:






```
Theta (parameters): [[4.21509609]
 [2.77011344]]
Final cost: 4032.922819835273
```

Practical No: 2

Aim: Implementation of Linear Regression using Gradient Descent

What is Linear Gradient Descent?

Linear Gradient Descent is an optimization algorithm used to minimize the cost function in linear regression. It's an iterative method that adjusts the model's parameters to reduce the difference between predicted and actual values. How
does it work?

1. Initialization: Initialize the model's parameters (weights and bias) randomly.
2. Prediction: Use the current parameters to make predictions on the training data.
3. Error calculation: Calculate the error between predicted and actual values using a loss function
(e.g., Mean Squared Error).
4. Gradient calculation: Calculate the gradient of the loss function with respect to each parameter.
5. Parameter update: Update each parameter by subtracting a fraction of the gradient (learning
rate) from the current value.
6. Repeat: Repeat steps 2-5 until convergence or a stopping criterion is reached.

Key concepts:

- Learning rate: A hyperparameter that controls the step size of each update.
- Gradient: The direction of the steepest ascent of the loss function.
- Convergence: The point at which the algorithm stops improving the model's performance.

Advantages:

- Simple to implement
- Fast convergence
- Can be used for large datasets

Disadvantages:

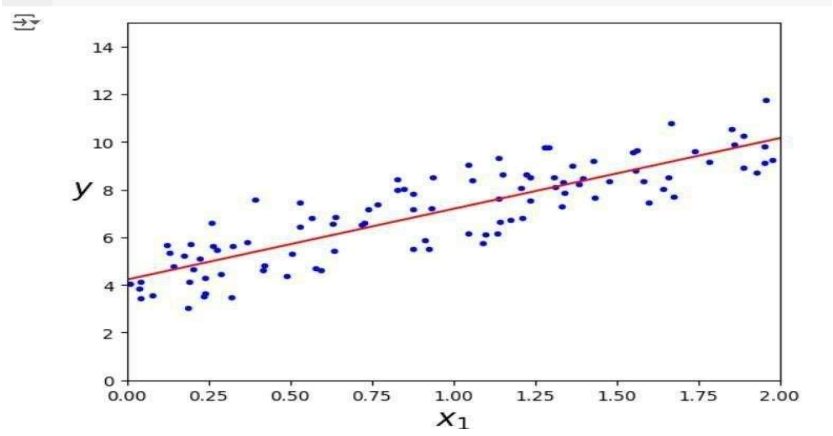
- May get stuck in local minima
- Requires careful tuning of the learning rate

Program:

Program:

```
import numpy as np
import matplotlib.pyplot as plt

# Step 2: Generate sample data
np.random.seed(0)
X = 2 * np.random.rand(100, 1)
y = 4 + 3 * X + np.random.randn(100, 1)

# Step 3: Add bias term
X_b = np.c_[np.ones((100, 1)), X]  # add x0 = 1 to each instance

# Step 4: Initialize parameters
theta = np.random.randn(2, 1)

# Step 5: Define hyperparameters
eta = 0.1 # learning rate
n_iterations = 1000

# Step 6: Gradient Descent
for iteration in range(n_iterations):
    gradients = 2/100 * X_b.T.dot(X_b.dot(theta) - y)
    theta = theta - eta * gradients

# Step 7: Prediction
X_new = np.array([[0], [2]])
X_new_b = np.c_[np.ones((2, 1)), X_new]
y_predict = X_new_b.dot(theta)

# Step 8: Plotting
plt.plot(X, y, "b.")
plt.plot(X_new, y_predict, "r-")
plt.xlabel("$x_1$", fontsize=18)
plt.ylabel("$y$", rotation=0, fontsize=18)
plt.axis([0, 2, 0, 15])
plt.show()
```

Output:



Practical

Aim: decision tree classificationalgorithm.

Theory:

Decision Tree is a Supervised learning technique that can be used for both classification and Regression problems, but mostly it is preferred for solving Classification problems. It is a tree- structured classifier, where internal nodes represent the features of a dataset, branches represent the decision rules and each leaf node represents the outcome. It is a graphical representation for getting all the possible solutions to a problem/decision based on given conditions.

Decision Tree Terminologies

- Root Node: Root node is from where the decision tree starts. It represents the entire dataset, which further gets divided into two or more homogeneous sets.
- Leaf Node: Leaf nodes are the final output node, and the tree cannot be segregated further after getting a leaf node.
- Splitting: Splitting is the process of dividing the decision node/root node into sub- nodes according to the given conditions.
- Branch/Sub Tree: A tree formed by splitting the tree.
- Pruning: Pruning is the process of removing the unwanted branches from the tree.
- Parent/Child node: The root node of the tree is called the parent node, and other nodes are called the child nodes.

Program:

```
#decision tree:
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score, classification_report
import matplotlib.pyplot as plt
from sklearn import tree
data = {
    'weather': ['Sunny', 'Sunny', 'cloudy', 'Rainy', 'Rainy', 'Rainy',
'cloudy', 'Sunny', 'Sunny', 'Rainy', 'Sunny', 'cloudy',
'cloudy','rain'],
    'Temperature': ['Hot', 'Hot', 'Hot', 'Mild', 'Cool', 'Cool',
'Cool', 'Mild', 'Cool', 'Mild', 'Mild', 'Mild', 'Hot','Mild'],
    'Humidity': ['High', 'High', 'High', 'High', 'Normal', 'Normal',
'Normal', 'High', 'Normal', 'Normal','Normal','High', 'Normal','High'],
```

```
    'Windy': ['weak','strong','weak','weak','weak','strong','strong','weak','weak','
weak','strong','strong','weak','strong'],
    'Playfootball':['No', 'No', 'Yes', 'Yes', 'Yes', 'No', 'Yes',
'No', 'Yes', 'Yes', 'Yes', 'Yes','yes', 'No']
}

df = pd.DataFrame(data)
df_encoded = pd.get_dummies(df, drop_first=True)
print(df_encoded)
X = df_encoded.drop('Playfootball_Yes', axis=1) y =
df_encoded['Playfootball_Yes']

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)
clf=DecisionTreeClassifier()
clf.fit(X_train, y_train) y_pred =
clf.predict(X_test)

print("Accuracy:", accuracy_score(y_test, y_pred)) print("Classification Report:\n",
classification_report(y_test, y_pred))
plt.figure(figsize=(12,8))
tree.plot_tree(clf, feature_names=X.columns, class_names=['No', 'Yes'], filled=True)
plt.show()
```

Output:

	weather_Sunny	weather_cloudy	weather_rain	Temperature_Hot	\
0	True	False	False	True	
1	True	False	False	True	
2	False	True	False	True	
3	False	False	False	False	
4	False	False	False	False	
5	False	False	False	False	
6	False	True	False	False	
7	True	False	False	False	
8	True	False	False	False	
9	False	False	False	False	
10	True	False	False	False	
11	False	True	False	False	
12	False	True	False	True	
13	False	False	True	False	

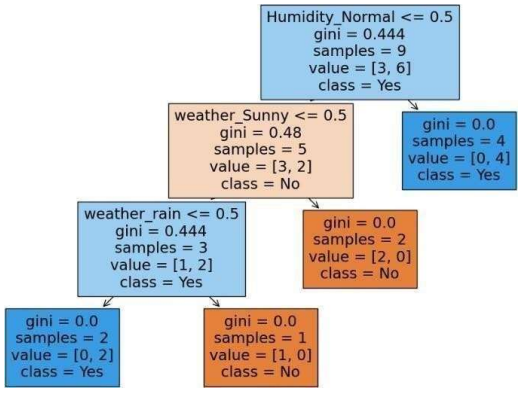
	Temperature_Mild	Humidity_Normal	Windy_weak	Playfootball_Yes	\
0	False	False	True	False	
1	False	False	False	False	
2	False	False	True	True	
3	True	False	True	True	
4	False	True	True	True	

5	False	True	False	False
6	False	True	False	True
7	True	False	True	False
8	False	True	True	True
9	True	True	True	True
10	True	True	False	True
11	True	False	False	True
12	False	True	True	False
13	True	False	False	False

	Playfootball_yes
0	False
1	False
2	False
3	False
4	False
5	False
6	False
7	False
8	False
9	False
10	False
11	False
12	True
13	False

Accuracy: 0.6 Classification Report:

	precision	recall	f1-score	support
False	1.00	0.33	0.50	3
True	0.50	1.00	0.67	2
accuracy			0.60	5
macro avg	0.75	0.67	0.58	5
weighted avg	0.80	0.60	0.57	5



Practical No: 3

Aim: Comparison of Classification Accuracy of SVM for given dataset

Theory:

Support Vector Machines (SVMs) are a set of supervised learning methods used for classification, regression, and outliers detection. The primary goal of the SVM algorithm is to find a hyperplane in an N-dimensional space (N is the number of features) that distinctly classifies the data points.

Key concepts:

- Hyperplane: In SVM, a hyperplane is a decision boundary that helps classify the data points. Data points falling on either side of the hyperplane can be attributed to different classes.
- Support Vectors: Data points that are closer to the hyperplane and influence the position and orientation of the hyperplane. These are the critical elements of the dataset.
- Margin: The distance between the hyperplane and the closest data points from either class. SVM aims to maximize this margin.

- Import Libraries: Import necessary libraries such as numpy, pandas, matplotlib, and sklearn.
- Load the Dataset: Load and explore the dataset to understand its structure and features.
- Preprocess the Data: Handle missing values, encode categorical variables, and split the data into training and testing sets.
- Train the SVM Model: Train the SVM model using different kernel functions (e.g., linear, polynomial, RBF).
- Evaluate the Model: Evaluate the model's performance using classification accuracy and other relevant metrics.
- Compare Results: Compare the classification accuracy of the SVM model with different kernel functions.

Program:

```
#Comparison of Classification Accuracy of SVM for given dataset using
iris dataset

import numpy as np
from sklearn import datasets
from sklearn.model_selection import train_test_split
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score
```

```
# Load Iris dataset
iris = datasets.load_iris()
X = iris.data
y = iris.target

# Split the data into training and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.3, random_state=42)

# Define SVM models with different kernels
kernels = ['linear', 'poly', 'rbf', 'sigmoid']
accuracies = {}

for kernel in kernels:
    # Initialize and train the SVM model
    model = SVC(kernel=kernel, random_state=42)
    model.fit(X_train, y_train)

    # Make predictions
    y_pred = model.predict(X_test)

    # Calculate accuracy
    accuracy = accuracy_score(y_test, y_pred)
    accuracies[kernel] = accuracy
    print(f'Accuracy with {kernel} kernel: {accuracy:.4f}')

# Display results
print("\nComparison of Classification Accuracy:")
for kernel, accuracy in accuracies.items():
    print(f"Kernel: {kernel} - Accuracy: {accuracy:.4f}")
```

Output:

Accuracy with linear kernel: 1.0000 Accuracy with poly kernel: 0.9778 Accuracy with rbf kernel: 1.0000 Accuracy with sigmoid kernel: 0.2222

Comparisonof Classification Accuracy: Kernel: linear - Accuracy: 1.0000 Kernel: poly - Accuracy: 0.9778 Kernel: rbf - Accuracy: 1.0000

Kernel: sigmoid - Accuracy: 0.2222

Practical

Aim: Principal compound analysis.

Theory:

What is Principal Component Analysis (PCA)?

The Principal Component Analysis is a popular unsupervised learning technique for reducing the dimensionality of large data sets. It increases interpretability yet, at the same time, it minimizes information loss. It helps to find the most significant features in a dataset and makes the data easy for plotting in 2D and 3D. PCA helps in finding a sequence of linear combinations of variables.

Steps for PCA Algorithm

1. Standardize the data: PCA requires standardized data, so the first step is to standardize the data to ensure that all variables have a mean of 0 and a standard deviation of 1.
2. Calculate the covariance matrix: The next step is to calculate the covariance matrix of the standardized data. This matrix shows how each variable is related to every other variable in the dataset.
3. Calculate the eigenvectors and eigenvalues: The eigenvectors and eigenvalues of the covariance matrix are then calculated. The eigenvectors represent the directions in which the data varies the most, while the eigenvalues represent the amount of variation along each eigenvector.
4. Choose the principal components: The principal components are the eigenvectors with the highest eigenvalues. These components represent the directions in which the data varies the most and are used to transform the original data into a lower-dimensional space.
5. Transform the data: The final step is to transform the original data into the lower- dimensional space defined by the principal components.

Program:

```
import numpy as np
import pandas as pd
import pprint
import matplotlib.pyplot as plt
from sklearn.datasets import load_iris
import pylab as pl
iris = load_iris()
iris_df = pd.DataFrame(iris.data,columns=[iris.feature_names])
```

```
iris_df.head()
X = iris.data
X.shape

from sklearn.preprocessing import StandardScaler
X_std = StandardScaler().fit_transform(X)
print (X std[0:5])
```

Output:

```
  [[-0.901      1.019   -1.34      -1.315]
   [-1.143     -0.132   -1.34      -1.315]
   [-1.385      0.328   -1.397     -1.315]
   [-1.507      0.098   -1.283     -1.315]
  [-1.022  1.249 -1.34  -1.315]]
The shape of Feature Matrix is - (150, 4)
```

Program:

```
print( "The shape of Feature Matrix is -",X_std.shape)
```

Output:

```
  [[-0.901      1.019 -1.34  -1.315]
   [-1.143    -0.132 -1.34  -1.315]
   [-1.385     0.328 -1.397 -1.315]
   [-1.507     0.098 -1.283 -1.315]
   [-1.022     1.249 -1.34  -1.315]]
 The shape    of Feature Matrix is - (150, 4)
```

Program:

```
eig_vals,      eig_vecs = np.linalg.eig(X_covariance_matrix) genvectors
print('Ei       n%s' %eig_vecs)
print('\n      Eigenvalues \n%s' %eig_vals)
```

Output:

```
Eigenvectors
[[ 0.521 -0.377 -0.72  0.261]
 [-0.269 -0.923  0.244 -0.124]
 [ 0.58  -0.024  0.142 -0.801]
 [ 0.565 -0.067  0.634  0.524]]

Eigenvalues
[2.938 0.92 0.148 0.021]
```

Program:

```
# Make a list of (eigenvalue, eigenvector) tuples
eig_pairs = [(np.abs(eig_vals[i]), eig_vecs[:,i]) for i in
range(len(eig_vals))]

# Sort the (eigenvalue, eigenvector) tuples from high to low


eig_pairs.sort(key=lambda x: x[0], reverse=True)

# Visually confirm that the list is correctly sorted by decreasing
eigenvalues
print('Eigenvalues in descending order:')
for i in eig_pairs:
    print(i[0])
```

Output:

Eigenvaluesindescending order: 2.938085050199995
0.9201649041624864
0.1477418210449475
0.020853862176462696

Program:

```
tot = sum(eig_vals)
var_exp = [(i / tot)*100 for i in sorted(eig_vals, reverse=True)]
cum_var_exp = np.cumsum(var_exp)
print("Variance captured by each component is \n",var_exp)
print(40 * '-')
print("Cumulative variance captured as we travel each component
\n",cum_var_exp)
```

Output:

Variance captured by each component is
[72.96244541329989, 22.850761786701753, 3.668921889282865,
0.5178709107154905]

Cumulative variance captured as we travel each component [72.962
95.813 99.482 100.]

Program:

```
print ("All Eigen Values along with Eigen Vectors")
pprint.pprint(eig_pairs)
print(40 * '-')
matrix_w = np.hstack((eig_pairs[0][1].reshape(4,1),
                      eig_pairs[1][1].reshape(4,1)))

print ('Matrix W:\n', matrix_w)
```

Output:

All Eigen Values along with Eigen Vectors [(2.938085050199995,
array([0.521, -0.269, 0.58 , 0.565])),
 (0.9201649041624864, array([-0.377, -0.923, -0.024, -0.067])),
 (0.1477418210449475, array([-0.72 , 0.244, 0.142, 0.634])),
 (0.020853862176462696, array([0.261, -0.124, -0.801, 0.524]))]

Matrix W:
 [[0.521 -0.377]
 [-0.269 -0.923]
 [0.58 -0.024]
 [0.565 -0.067]]

Program:

```
Y = X_std.dot(matrix_w)
print( Y[0:5])
```

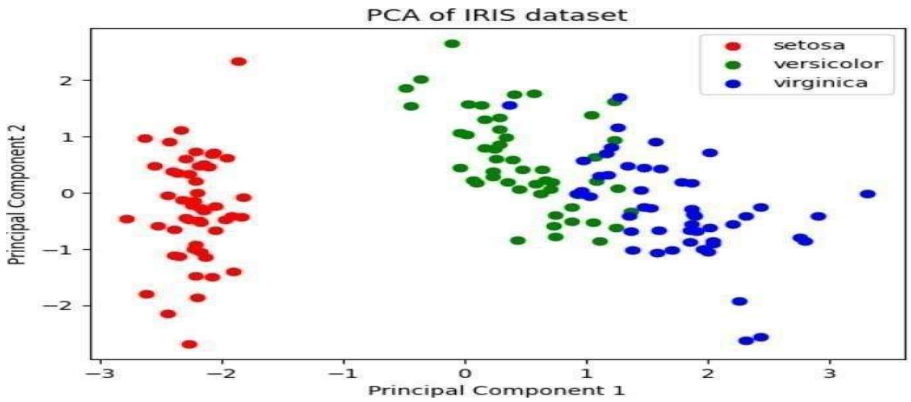
Output:

[[-2.265 -0.48]
 [-2.081 0.674]
 [-2.364 0.342]
 [-2.299 0.597]
 [-2.39 -0.647]]

Program:

```
pl.figure()
target_names = iris.target_names
y = iris.target
for c, i, target_name in zip("rgb", [0, 1, 2], target_names):
    pl.scatter(Y[y==i,0], Y[y==i,1], c=c, label=target_name)
pl.xlabel('Principal Component 1')
pl.ylabel('Principal Component 2')
pl.legend()
pl.title('PCA of IRIS dataset')
pl.show()
```

Output:


PCA of IRIS dataset
setosa
versicolor
virginica
Principal Component 2
Principal Component 1
2
1
0
−1
−2
−3
−2
−1
0
1
2
3

Practical No: 4

Aim: Generate your own feature set by combining existing set of features, or defining new ones. Feature Representation.

Theory:

Certainly! Let's use the famous Iris dataset to demonstrate feature engineering by combining existing features and creating new ones. The Iris dataset contains features about the length and width of the sepals and petals of iris flowers, and the goal is to classify them into three species.

Steps to Generate and Use New Features with the Iris Dataset

1. Load the Iris Dataset: We'll use scikit-learn's built-in Iris dataset.
2. Create New Features: Generate new features by combining or transforming the existing features.
3. Preprocess the Data: Prepare the data for model training.
4. Train and Evaluate Models: Train a model using the new feature set and evaluate its performance.

Program:

```
import pandas as pd
import numpy as np
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler, PolynomialFeatures
from sklearn.svm import SVC
from sklearn.metrics import classification_report, accuracy_score

# Load the Iris Dataset
iris = load_iris()
data = pd.DataFrame(data=iris.data, columns=iris.feature_names)
data['target'] = iris.target

# Display the first few rows of the dataset
print("Original Data:\n", data.head())

# Generate New Features
# 1. Mathematical Combinations
data['sepal_length_petal_length_sum'] = data['sepal length (cm)'] +
data['petal length (cm)']
data['sepal_width_petal_width_product'] = data['sepal width (cm)'] *
data['petal width (cm)']
```

```
# 2. Interaction Features
data['sepal_length_petal_width_ratio'] = np.where(data['petal width (cm)'] != 0, data['sepal length (cm)'] / data['petal width (cm)'], 0)

# 3. Polynomial Features (degree 2 for interaction terms) poly = PolynomialFeatures(degree=2, include_bias=False)
poly_features = poly.fit_transform(data[['sepal length (cm)', 'sepal width (cm)', 'petal length (cm)', 'petal width (cm)']]) poly_feature_names = poly.get_feature_names_out(['sepallength (cm)', 'sepal width (cm)', 'petal length (cm)', 'petal width (cm)']) poly_df = pd.DataFrame(poly_features, columns=poly_feature_names)

# Combine polynomial features with the original dataframe data = pd.concat([data, poly_df], axis=1)

# Drop the original features if necessary
data.drop(['sepal length (cm)', 'sepal width (cm)', 'petal length (cm)', 'petal width (cm)'], axis=1, inplace=True)

# Split Data into Features and Target
X = data.drop('target', axis=1) # Features y = data['target']                                    # Target

# Split Data into Training and Testing Sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

# Standardize the Data scaler= StandardScaler()
X_train = scaler.fit_transform(X_train) X_test = scaler.transform(X_test)

# Train the SVM Model with RBF Kernel (for example) model = SVC(kernel='rbf', random_state=42) model.fit(X_train, y_train)

# Evaluate the Model
y_pred= model.predict(X_test)
accuracy = accuracy_score(y_test, y_pred) report = classification_report(y_test, y_pred)

print(f"Accuracy: {accuracy:.4f}") print(f"Classification Report:\n{report}")
```

Output:

Original Data:

	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)
0	5.1	3.5	1.4	0.2
1	4.9	3.0	1.4	0.2
2	4.7	3.2	1.3	0.2
3	4.6	3.1	1.5	0.2
4	5.0	3.6	1.4	0.2

	target
0	0
1	0
2	0
3	0
4	0

Accuracy: 1.0000 Classification Report:

	precision	recall	f1-score	support
0	1.00	1.00	1.00	19
1	1.00	1.00	1.00	13
2	1.00	1.00	1.00	13
accuracy			1.00	45
macro avg	1.00	1.00	1.00	45
weighted avg	1.00	1.00	1.00	45

Practical No:5

Aim: Generate samples of a normal distribution with specific parameters with respect to Mean and Covariance

Theory:

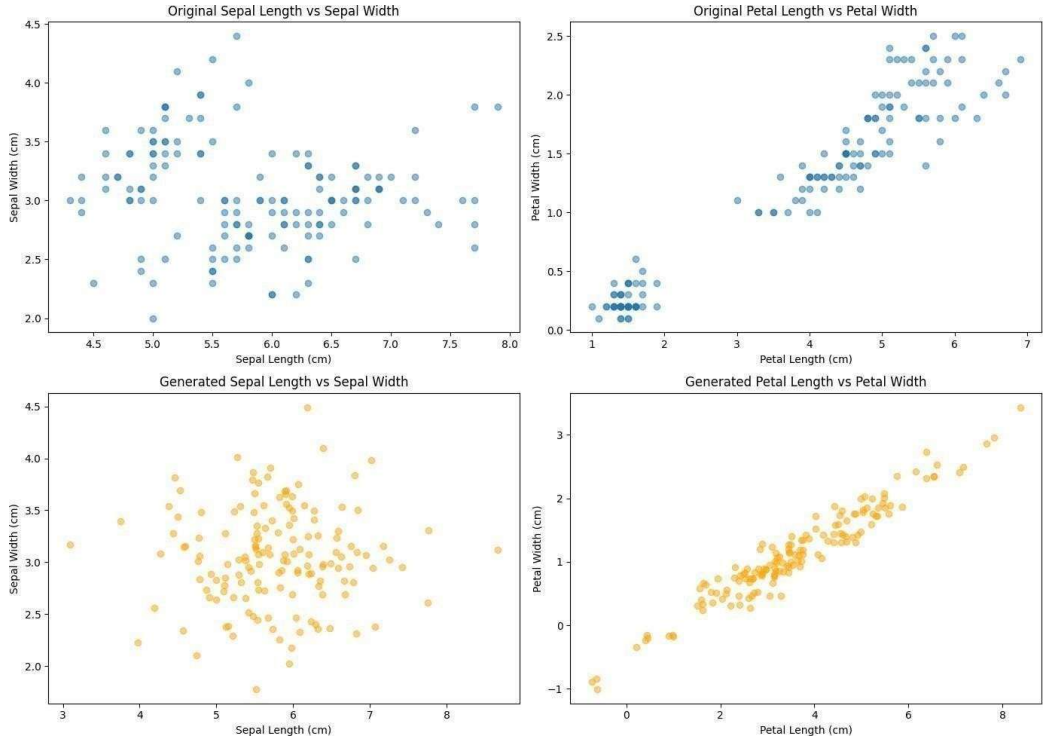
- **Load the Iris Dataset**:
  - The Iris dataset is loaded and converted into a pandas DataFrame for easy manipulation.
- **Calculate Mean and Covariance**:
  - Compute the mean and covariance matrix of the features in the Iris dataset using mean() and cov().
- **Generate Samples**:
  - Use np.random.multivariate_normalto generate samples from a multivariate normal distribution with the calculated mean and covariance matrix.
- **Visualization**:
  - The generated samples are plotted alongside the original data for visual comparison. This helps to understand how well the generated samples resemble the original distribution.

Program:

```
import numpy as np
import pandas as pd
from sklearn.datasets import load_iris
import matplotlib.pyplot as plt

# Load the Iris Dataset
iris = load_iris()
data = pd.DataFrame(data=iris.data, columns=iris.feature_names)

# Display basic statistics
print("Data Mean:\n", data.mean())
print("Data Covariance Matrix:\n", data.cov())

# Extract mean and covariance
mean = data.mean().values
covariance_matrix = data.cov().values
```

```
# Generate samples from a normal distribution with the given mean and covariance
num_samples = 150  # Number of samples to generate
samples = np.random.multivariate_normal(mean, covariance_matrix, num_samples)

# Create a DataFrame for the generated samples generated_data =
pd.DataFrame(samples, columns=data.columns)

# Display the first few rows of the generated data print("Generated
Samples:\n", generated_data.head())

# Plotting the original and generated data for comparison fig, axs =
plt.subplots(2, 2, figsize=(14, 10))

# Original Data Plots
axs[0, 0].scatter(data['sepal length (cm)'], data['sepal width (cm)'], alpha=0.5)
axs[0, 0].set_title('Original Sepal Length vs Sepal Width') axs[0,
0].set_xlabel('Sepal Length (cm)')
axs[0, 0].set_ylabel('Sepal Width (cm)')

axs[0, 1].scatter(data['petal length (cm)'], data['petal width (cm)'], alpha=0.5)
axs[0, 1].set_title('Original Petal Length vs Petal Width') axs[0,
1].set_xlabel('Petal Length (cm)')
axs[0, 1].set_ylabel('Petal Width (cm)')

# Generated Data Plots
axs[1, 0].scatter(generated_data['sepal length (cm)'], generated_data['sepalwidth
(cm)'], alpha=0.5, color='orange') axs[1, 0].set_title('Generated Sepal Length vs Sepal
Width') axs[1, 0].set_xlabel('Sepal Length (cm)')
axs[1, 0].set_ylabel('Sepal Width (cm)')

axs[1, 1].scatter(generated_data['petal length (cm)'], generated_data['petalwidth (cm)'],
alpha=0.5, color='orange') axs[1, 1].set_title('Generated Petal Length vs Petal Width')
axs[1, 1].set_xlabel('Petal Length (cm)')
axs[1, 1].set_ylabel('Petal Width (cm)')

plt.tight_layout()
plt.show()
```

Output:


Original Sepal Length vs Sepal Width
Sepal Width (cm)
Sepal Length (cm)
Original Petal Length vs Petal Width
Petal Width (cm)
Petal Length (cm)
Generated Sepal Length vs Sepal Width
Sepal Width (cm)
Sepal Length (cm)
Generated Petal Length vs Petal Width
Petal Width (cm)
Petal Length (cm)

Practical No: 6

Aim: Implement Linear Perceptron Learning algorithm

Theory:

1. **Load and Prepare the Dataset:**
   - Load the Iris dataset and filter it to include only two classes for binary classification.
   - Standardize the features for better convergence.
2. **Define the Perceptron Model:**
   - Perceptron class:
     - ___init_: Initializes the learning rate and number of iterations.
     - fit: Trains the Perceptron by iterating through epochs and updating weights based on misclassification.
     - predict: Makes predictions based on the learned weights.
     - _net_input: Calculates the net input to the activation function.
3. **Train and Evaluate the Model:**
   - Train the Perceptron using the training data.
   - Evaluate the model using accuracy and a classification report.
   - Plot training errors over epochs to visualize the convergence of the model.

This implementation provides a basic understanding of the Perceptron algorithm and how it can be applied to the Iris dataset. For more complex datasets or multi-class problems, consider using more sophisticated algorithms or extending the Perceptron model.

Program:

```
import numpy as np
import pandas as pd
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import accuracy_score, classification_report
import matplotlib.pyplot as plt

# Load the Iris Dataset
iris = load_iris()
data = pd.DataFrame(data=iris.data, columns=iris.feature_names)
data['target'] = iris.target

# For binary classification: Iris-setosa vs. Iris-versicolor
data = data[data['target'] != 2] # Exclude Iris-virginica
data['target'] = data['target'].map({0: 0, 1: 1}) # Map to binary
target
```

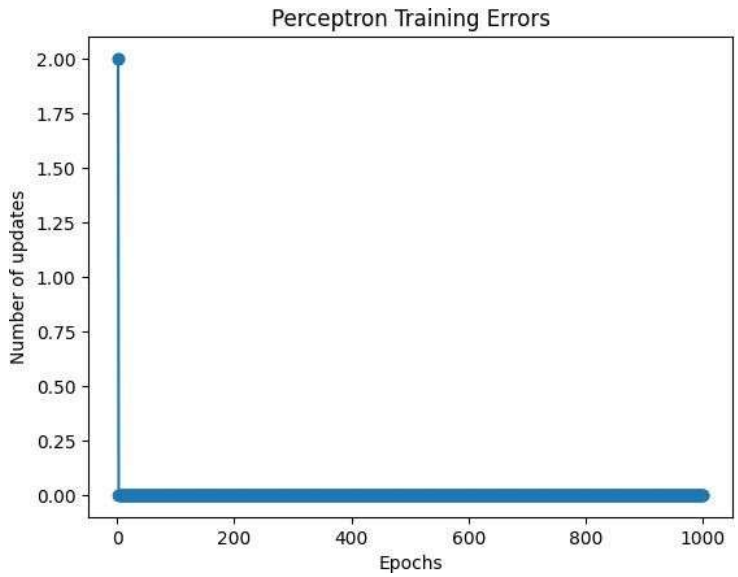
```
# Split Data into Features and Target X =
data.drop('target', axis=1).values y =
data['target'].values

# Split Data into Training and Testing Sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

# Standardize the Data scaler=
StandardScaler()
X_train = scaler.fit_transform(X_train) X_test =
scaler.transform(X_test)

# Define the Perceptron Algorithm class
Perceptro n:
    def init (self, learning_rate=0.01, n_iter=1000):
        self.learning_rate = learning_rate self.n_iter = n_iter

    def fit(self, X, y):
        self.weights = np.zeros(X.shape[1] + 1) # Including bias term self.errors = []

        for_inrange(self.n_iter): errors = 0
            for xi, target in zip(X, y):
                update = self.learning_rate * (target - self.predict(xi))
                self.weights[1:] += update * xi
                self.weights[0] += update errors +=
                int(update != 0.0)
            self.errors.append(errors)

    def predict(self, X):
        return np.where(self._net_input(X) >= 0.0, 1, 0)

    def _net_input(self, X):
        returnnp.dot(X, self.weights[1:]) + self.weights[0]

# Train the Perceptron Model
perceptron=Perceptron(learning_rate=0.01, n_iter=1000) perceptron.fit(X_train, y_train)

# Make Predictions
y_pred= perceptron.predict(X_test)
```

```
# Evaluate the Model
accuracy = accuracy_score(y_test, y_pred)
report = classification_report(y_test, y_pred)

print(f"Accuracy: {accuracy:.4f}")
print(f"Classification Report:\n{report}")

# Plot the Training Errors
plt.plot(range(1, len(perceptron.errors) + 1), perceptron.errors,
marker='o')
plt.xlabel('Epochs')
plt.ylabel('Number of updates')
plt.title('Perceptron Training Errors')
plt.show()
```

Output:

Accuracy: 1.0000 Classification Report:

	precision	recall	f1-score	support
0	1.00	1.00	1.00	17
1	1.00	1.00	1.00	13
accuracy			1.00	30
macro avg	1.00	1.00	1.00	30
weighted avg	1.00	1.00	1.00	30



Practical No: 7

Aim: Build IRIS flower classification in python using pattern recognition models

Theory:

We'll use the following pattern recognition models:

1. **Logistic Regression**
2. **k-Nearest Neighbors (k-NN)**
3. **Support Vector Machine (SVM)**
4. **Decision Tree**
5. **Random Forest**

Here's a step-by-step guide to implement and evaluate these models using the Iris dataset in Python.

Step-by-Step Implementation

1. **Load and Prepare the Dataset**:
   - Load the Iris dataset.
   - Preprocess the data (e.g., standardization).
   - Split the data into training and testing sets.
2. **Define and Train Models**:
   - Implement each model and fit it to the training data.
3. **Evaluate the Models**:
   - Assess the performance of each model using accuracy and other metrics.
4. **Compare Results**:
   - Compare the classification accuracy and other metrics of each model.

Program:

```
import numpy as np
import pandas as pd
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LogisticRegression
from sklearn.neighbors import KNeighborsClassifier
from sklearn.svm import SVC
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score, classification_report
import matplotlib.pyplot as plt

# Load the Iris Dataset
iris = load_iris()
```

```
X = iris.data y=
iris.target

# Split Data into Training and Testing Sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

# Standardize the Data scaler=
StandardScaler()
X_train = scaler.fit_transform(X_train) X_test =
scaler.transform(X_test)

# Definethe Models models = {
    "Logistic Regression": LogisticRegression(max_iter=200), "k-Nearest
    Neighbors": KNeighborsClassifier(),
    "Support Vector Machine": SVC(), "Decision Tree":
    DecisionTreeClassifier(), "Random Forest":
    RandomForestClassifier()
}

# Train and Evaluate Models results = {}
formodel_name, model inmodels.items(): model.fit(X_train, y_train)
    y_pred= model.predict(X_test)
    accuracy = accuracy_score(y_test, y_pred) report =
    classification_report(y_test, y_pred,
target_names=iris.target_names)
    results[model_name] = {
        "accuracy":accuracy,
        "report": report
    }

# Print Results
for model_name, metrics in results.items(): print(f"Model:
    {model_name}") print(f"Accuracy:
    {metrics['accuracy']:.4f}")
    print(f"Classification Report:\n{metrics['report']}") print("-" * 60)

# Plotting the results model_names =
list(results.keys())
accuracies = [results[model_name]['accuracy'] for model_name in model_names]
```

```
plt.figure(figsize=(10, 6))
plt.bar(model_names, accuracies, color='skyblue')
plt.xlabel('Model')
plt.ylabel('Accuracy')
plt.title('Model Comparison')
plt.xticks(rotation=45)
plt.ylim(0, 1)
plt.show()
```

Output:

Model: Logistic Regression Accuracy:
1.0000
Classification Report:

	precision	recall	f1-score	support
setosa	1.00	1.00	1.00	19
versicolor	1.00	1.00	1.00	13
virginica	1.00	1.00	1.00	13
accuracy			1.00	45
macro avg	1.00	1.00	1.00	45
weighted avg	1.00	1.00	1.00	45

Model: k-Nearest Neighbors Accuracy:
1.0000 Classification Report:

	precision	recall	f1-score	support
setosa	1.00	1.00	1.00	19
versicolor	1.00	1.00	1.00	13
virginica	1.00	1.00	1.00	13
accuracy			1.00	45
macroavg	1.00	1.00	1.00	45
weighted avg	1.00	1.00	1.00	45

Model: Support Vector Machine
Accuracy: 1.0000 Classification
Report:

	precision	recall	f1-score	support
setosa	1.00	1.00	1.00	19
versicolor	1.00	1.00	1.00	13
virginica	1.00	1.00	1.00	13
accuracy			1.00	45
macro avg	1.00	1.00	1.00	45
weighted avg	1.00	1.00	1.00	45

Model: Decision Tree Accuracy:
1.0000 Classification Report:

	precision	recall	f1-score	support
setosa	1.00	1.00	1.00	19
versicolor	1.00	1.00	1.00	13
virginica	1.00	1.00	1.00	13
accuracy			1.00	45
macro avg	1.00	1.00	1.00	45
weighted avg	1.00	1.00	1.00	45

Model: Random Forest

Parul University

FACULTY OF ENGINEERING AND TECHNOLOGY
BACHELOR OF TECHNOLOGY

Pattern recognition

(303105482)

7TH SEMESTER

Computer Science & Engineering Department

Lab Manual

Parul® University
VADODARA, GUJARAT

NAAC A++

FACULTY OF ENGINEERING AND TECHNOLOGY
Subject Name: Pattern Recognition Laboratory
Subject Code: 30105482
B.Tech. 4rd year 7TH semester

CERTIFICATE

This is to certify that Student **MERUGUMALI RAVITEJA,** with Enrollment No. **2203031240822** has Successfully completed his Laboratory experiments in **Pattern Recognition (303105482)** from the department of **ARTIFICIAL INTELLINGENCE AND DATA SCIENCE** during the academic year **2024-2025**.

योगः कर्मसु कौशलम्

PARUL UNIVERSITY

Date of Submission: ..... Staff In Charge: ……..

Head of Department: .....

TABLE OF CONTENT

Practical No: 1

Aim:. Implementation of Gradient Descent.

What is Gradient Descent?

Gradient Descent stands as a cornerstone orchestrating the intricate dance of model optimization. At its core, it is a numerical optimization algorithm that aims to find the optimal parameters—weights and biases—of a neural network by minimizing a defined cost function.

Gradient Descent (GD) is a widely used optimization algorithm in machine learning and deep learning that minimises the cost function of a neural network model during training. It works by iteratively adjusting the weights or parameters of the model in the direction of the negative gradient of the cost function until the minimum of the cost function is reached.

The learning happens during the backpropagation while training the neural network-based model. There is a term known as Gradient Descent, which is used to optimize the weight and biases based on the cost function. The cost function evaluates the difference between the actual and predicted outputs.

Gradient Descent is a fundamental optimization algorithm in machine learning used to minimize the cost or loss function during model training.

- It iteratively adjusts model parameters by moving in the direction of the steepest decrease in the cost function.
- The algorithm calculates gradients, representing the partial derivatives of the cost function concerning each parameter.

These gradients guide the updates, ensuring convergence towards the optimal parameter values that yield the lowest possible cost.

Gradient Descent is versatile and applicable to various machine learning models, including linear regression and neural networks. Its efficiency lies in navigating the parameter space efficiently, enabling models to learn patterns and make accurate predictions. Adjusting the learning rate is crucial to balance convergence speed and avoiding overshooting the optimal solution.

Gradient descent is an iterative optimization algorithm for finding the local minimum of a function.

To find the local minimum of a function using gradient descent, we must take steps proportional to the negative of the gradient (move away from the gradient) of the function at the current point. If we take steps proportional to the positive of the gradient (moving towards the gradient), we will approach a local maximum of the function, and the procedure is called **Gradient Ascent.**

DATASET TAKEN:- Globalisation Index

The dataset contains 167 rows and 14 columns, with the following features:

1. Country: The name of the country (categorical feature).
2. Average Score: The average score of various factors for each country (numeric, float).
3. Safety Security: The score related to safety and security (numeric, float).
4. Personal Freedom: The score for personal freedom in each country (numeric, float).
5. Governance: The score for governance (numeric, float).
6. Social Capital: The score for social capital (numeric, float).
7. Investment Environment: The score related to the investment environment (numeric, float).
8. Enterprise Conditions: The score for enterprise conditions (numeric, float).
9. Market Access Infrastructure: The score for market access and infrastructure (numeric, float).
10. Economic Quality: The score for economic quality (numeric, float).
11. Living Conditions: The score for living conditions (numeric, float).
12. Health: The score for health conditions (numeric, float).
13. Education: The score for education (numeric, float).
14. Natural Environment: The score for natural environment quality (numeric, float).

All features except for "Country" are numeric and indicate scores related to various aspects of each country's conditions.

Program:

```
import numpy as np
import matplotlib.pyplot as plt

#Generatesomedata
np.random.seed(42)
X = 2 * np.random.rand(100, 1)
y = 4 + 3 * X + np.random.randn(100, 1)

# Function to compute the cost def
compute_cost(X, y, theta):
        m = len(y)
        predictions=X.dot(theta)
        cost = (1/2*m) * np.sum(np.square(predictions - y)) return cost

# Function to perform gradient descent
def gradient_descent(X, y, theta, learning_rate, iterations): m = len(y)
        cost_history=np.zeros(iterations)

        for i in range(iterations): predictions= X.dot(theta)
                theta = theta - (1/m) * learning_rate * (X.T.dot(predictions - cost_history[i] =

y))          compute_cost(X, y, theta)
```

retu rn theta, cost_history

Add x0 = 1 to each instance X_b =
np.c_[np.ones((100, 1)), X]

# Initialize theta
theta = np.random.randn(2, 1)

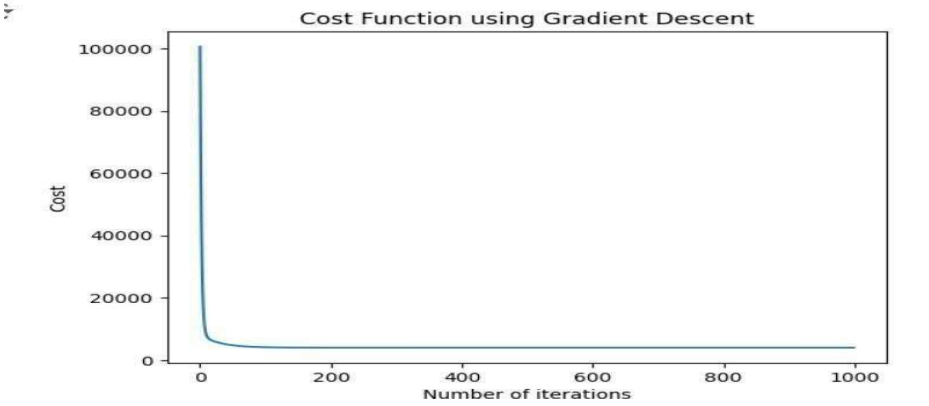
Sethyperparameters learning_rate =
0.1
iterations= 1000

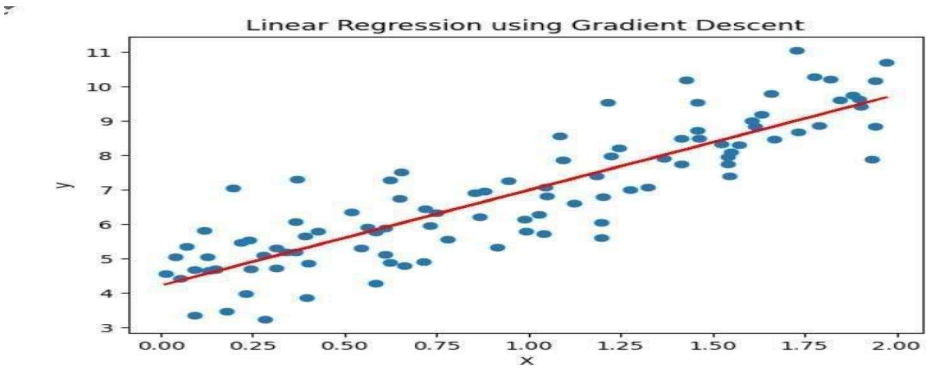
Perform gradient descent
theta, cost_history = gradient_descent(X_b, y, theta, learning_rate, iterations)

Plot the cost function plt.plot(range(iterations), cost_history)

```
plt.xlabel("Number of iterations")
plt.ylabel("Cost")
plt.title("Cost Function using Gradient Descent")
plt.show()

# Plot the data and the regression line
plt.scatter(X, y)
plt.plot(X, X_b.dot(theta), color='red')
plt.xlabel("X")
plt.ylabel("y")
plt.title("Linear Regression using Gradient Descent")
t.show()

print("Theta (parameters):", theta)
print("Final cost:", cost_history[-1])
```

Output:






```
Theta (parameters): [[4.21509609]
 [2.77011344]]
Final cost: 4032.922819835273
```

Practical No: 2

Aim: Implementation of Linear Regression using Gradient Descent

What is Linear Gradient Descent?

Linear Gradient Descent is an optimization algorithm used to minimize the cost function in linear regression. It's an iterative method that adjusts the model's parameters to reduce the difference between predicted and actual values. How
does it work?

1. Initialization: Initialize the model's parameters (weights and bias) randomly.
2. Prediction: Use the current parameters to make predictions on the training data.
3. Error calculation: Calculate the error between predicted and actual values using a loss function
(e.g., Mean Squared Error).
4. Gradient calculation: Calculate the gradient of the loss function with respect to each parameter.
5. Parameter update: Update each parameter by subtracting a fraction of the gradient (learning
rate) from the current value.
6. Repeat: Repeat steps 2-5 until convergence or a stopping criterion is reached.

Key concepts:

- Learning rate: A hyperparameter that controls the step size of each update.
- Gradient: The direction of the steepest ascent of the loss function.
- Convergence: The point at which the algorithm stops improving the model's performance.

Advantages:

- Simple to implement
- Fast convergence
- Can be used for large datasets

Disadvantages:

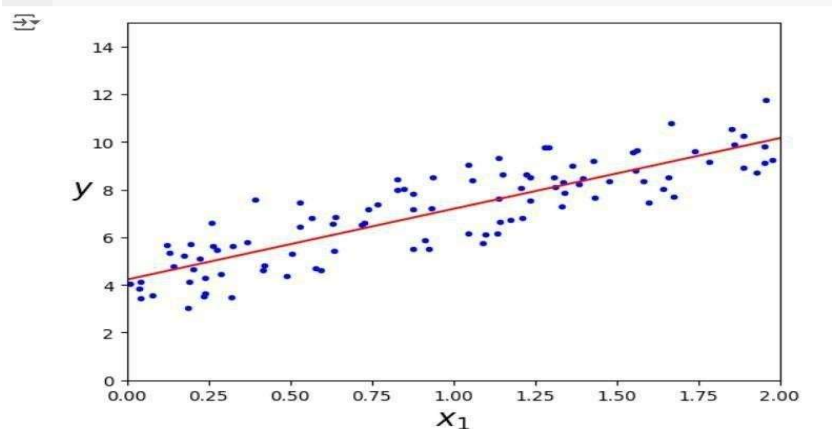
- May get stuck in local minima
- Requires careful tuning of the learning rate

Program:

Program:

```
import numpy as np
import matplotlib.pyplot as plt

# Step 2: Generate sample data
np.random.seed(0)
X = 2 * np.random.rand(100, 1)
y = 4 + 3 * X + np.random.randn(100, 1)

# Step 3: Add bias term
X_b = np.c_[np.ones((100, 1)), X]  # add x0 = 1 to each instance

# Step 4: Initialize parameters
theta = np.random.randn(2, 1)

# Step 5: Define hyperparameters
eta = 0.1 # learning rate
n_iterations = 1000

# Step 6: Gradient Descent
for iteration in range(n_iterations):
    gradients = 2/100 * X_b.T.dot(X_b.dot(theta) - y)
    theta = theta - eta * gradients

# Step 7: Prediction
X_new = np.array([[0], [2]])
X_new_b = np.c_[np.ones((2, 1)), X_new]
y_predict = X_new_b.dot(theta)

# Step 8: Plotting
plt.plot(X, y, "b.")
plt.plot(X_new, y_predict, "r-")
plt.xlabel("$x_1$", fontsize=18)
plt.ylabel("$y$", rotation=0, fontsize=18)
plt.axis([0, 2, 0, 15])
plt.show()
```

Output:



Practical

Aim: decision tree classificationalgorithm.

Theory:

Decision Tree is a Supervised learning technique that can be used for both classification and Regression problems, but mostly it is preferred for solving Classification problems. It is a tree- structured classifier, where internal nodes represent the features of a dataset, branches represent the decision rules and each leaf node represents the outcome. It is a graphical representation for getting all the possible solutions to a problem/decision based on given conditions.

Decision Tree Terminologies

- Root Node: Root node is from where the decision tree starts. It represents the entire dataset, which further gets divided into two or more homogeneous sets.
- Leaf Node: Leaf nodes are the final output node, and the tree cannot be segregated further after getting a leaf node.
- Splitting: Splitting is the process of dividing the decision node/root node into sub- nodes according to the given conditions.
- Branch/Sub Tree: A tree formed by splitting the tree.
- Pruning: Pruning is the process of removing the unwanted branches from the tree.
- Parent/Child node: The root node of the tree is called the parent node, and other nodes are called the child nodes.

Program:

```
#decision tree:
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score, classification_report
import matplotlib.pyplot as plt
from sklearn import tree
data = {
    'weather': ['Sunny', 'Sunny', 'cloudy', 'Rainy', 'Rainy', 'Rainy',
'cloudy', 'Sunny', 'Sunny', 'Rainy', 'Sunny', 'cloudy',
'cloudy','rain'],
    'Temperature': ['Hot', 'Hot', 'Hot', 'Mild', 'Cool', 'Cool',
'Cool', 'Mild', 'Cool', 'Mild', 'Mild', 'Mild', 'Hot','Mild'],
    'Humidity': ['High', 'High', 'High', 'High', 'Normal', 'Normal',
'Normal', 'High', 'Normal', 'Normal','Normal','High', 'Normal','High'],
```

```
    'Windy': ['weak','strong','weak','weak','weak','strong','strong','weak','weak','
weak','strong','strong','weak','strong'],
    'Playfootball':['No', 'No', 'Yes', 'Yes', 'Yes', 'No', 'Yes',
'No', 'Yes', 'Yes', 'Yes', 'Yes','yes', 'No']
}

df = pd.DataFrame(data)
df_encoded = pd.get_dummies(df, drop_first=True)
print(df_encoded)
X = df_encoded.drop('Playfootball_Yes', axis=1) y =
df_encoded['Playfootball_Yes']

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)
clf=DecisionTreeClassifier()
clf.fit(X_train, y_train) y_pred =
clf.predict(X_test)

print("Accuracy:", accuracy_score(y_test, y_pred)) print("Classification Report:\n",
classification_report(y_test, y_pred))
plt.figure(figsize=(12,8))
tree.plot_tree(clf, feature_names=X.columns, class_names=['No', 'Yes'], filled=True)
plt.show()
```

Output:

	weather_Sunny	weather_cloudy	weather_rain	Temperature_Hot	\
0	True	False	False	True	
1	True	False	False	True	
2	False	True	False	True	
3	False	False	False	False	
4	False	False	False	False	
5	False	False	False	False	
6	False	True	False	False	
7	True	False	False	False	
8	True	False	False	False	
9	False	False	False	False	
10	True	False	False	False	
11	False	True	False	False	
12	False	True	False	True	
13	False	False	True	False	

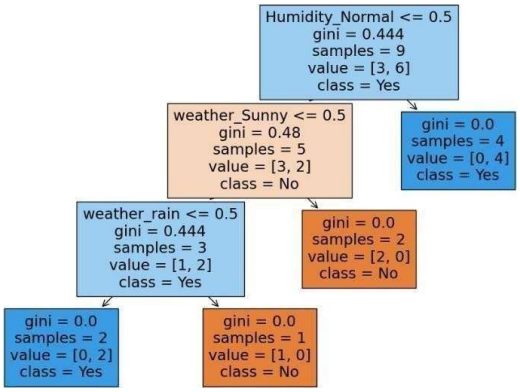
	Temperature_Mild	Humidity_Normal	Windy_weak	Playfootball_Yes	\
0	False	False	True	False	
1	False	False	False	False	
2	False	False	True	True	
3	True	False	True	True	
4	False	True	True	True	

5	False	True	False	False
6	False	True	False	True
7	True	False	True	False
8	False	True	True	True
9	True	True	True	True
10	True	True	False	True
11	True	False	False	True
12	False	True	True	False
13	True	False	False	False

	Playfootball_yes
0	False
1	False
2	False
3	False
4	False
5	False
6	False
7	False
8	False
9	False
10	False
11	False
12	True
13	False

Accuracy: 0.6 Classification Report:

	precision	recall	f1-score	support
False	1.00	0.33	0.50	3
True	0.50	1.00	0.67	2
accuracy			0.60	5
macro avg	0.75	0.67	0.58	5
weighted avg	0.80	0.60	0.57	5



Practical No: 3

Aim: Comparison of Classification Accuracy of SVM for given dataset

Theory:

Support Vector Machines (SVMs) are a set of supervised learning methods used for classification, regression, and outliers detection. The primary goal of the SVM algorithm is to find a hyperplane in an N-dimensional space (N is the number of features) that distinctly classifies the data points.

Key concepts:

- Hyperplane: In SVM, a hyperplane is a decision boundary that helps classify the data points. Data points falling on either side of the hyperplane can be attributed to different classes.
- Support Vectors: Data points that are closer to the hyperplane and influence the position and orientation of the hyperplane. These are the critical elements of the dataset.
- Margin: The distance between the hyperplane and the closest data points from either class. SVM aims to maximize this margin.

- Import Libraries: Import necessary libraries such as numpy, pandas, matplotlib, and sklearn.
- Load the Dataset: Load and explore the dataset to understand its structure and features.
- Preprocess the Data: Handle missing values, encode categorical variables, and split the data into training and testing sets.
- Train the SVM Model: Train the SVM model using different kernel functions (e.g., linear, polynomial, RBF).
- Evaluate the Model: Evaluate the model's performance using classification accuracy and other relevant metrics.
- Compare Results: Compare the classification accuracy of the SVM model with different kernel functions.

Program:

```
#Comparison of Classification Accuracy of SVM for given dataset using
iris dataset

import numpy as np
from sklearn import datasets
from sklearn.model_selection import train_test_split
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score
```

```
# Load Iris dataset
iris = datasets.load_iris()
X = iris.data
y = iris.target

# Split the data into training and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.3, random_state=42)

# Define SVM models with different kernels
kernels = ['linear', 'poly', 'rbf', 'sigmoid']
accuracies = {}

for kernel in kernels:
    # Initialize and train the SVM model
    model = SVC(kernel=kernel, random_state=42)
    model.fit(X_train, y_train)

    # Make predictions
    y_pred = model.predict(X_test)

    # Calculate accuracy
    accuracy = accuracy_score(y_test, y_pred)
    accuracies[kernel] = accuracy
    print(f'Accuracy with {kernel} kernel: {accuracy:.4f}')

# Display results
print("\nComparison of Classification Accuracy:")
for kernel, accuracy in accuracies.items():
    print(f"Kernel: {kernel} - Accuracy: {accuracy:.4f}")
```

Output:

Accuracy with linear kernel: 1.0000 Accuracy with poly kernel: 0.9778 Accuracy with rbf kernel: 1.0000 Accuracy with sigmoid kernel: 0.2222

Comparisonof Classification Accuracy: Kernel: linear - Accuracy: 1.0000 Kernel: poly - Accuracy: 0.9778 Kernel: rbf - Accuracy: 1.0000

Kernel: sigmoid - Accuracy: 0.2222

Practical

Aim: Principal compound analysis.

Theory:

What is Principal Component Analysis (PCA)?

The Principal Component Analysis is a popular unsupervised learning technique for reducing the dimensionality of large data sets. It increases interpretability yet, at the same time, it minimizes information loss. It helps to find the most significant features in a dataset and makes the data easy for plotting in 2D and 3D. PCA helps in finding a sequence of linear combinations of variables.

Steps for PCA Algorithm

1. Standardize the data: PCA requires standardized data, so the first step is to standardize the data to ensure that all variables have a mean of 0 and a standard deviation of 1.
2. Calculate the covariance matrix: The next step is to calculate the covariance matrix of the standardized data. This matrix shows how each variable is related to every other variable in the dataset.
3. Calculate the eigenvectors and eigenvalues: The eigenvectors and eigenvalues of the covariance matrix are then calculated. The eigenvectors represent the directions in which the data varies the most, while the eigenvalues represent the amount of variation along each eigenvector.
4. Choose the principal components: The principal components are the eigenvectors with the highest eigenvalues. These components represent the directions in which the data varies the most and are used to transform the original data into a lower-dimensional space.
5. Transform the data: The final step is to transform the original data into the lower- dimensional space defined by the principal components.

Program:

```
import numpy as np
import pandas as pd
import pprint
import matplotlib.pyplot as plt
from sklearn.datasets import load_iris
import pylab as pl
iris = load_iris()
iris_df = pd.DataFrame(iris.data,columns=[iris.feature_names])
```

```
iris_df.head()
X = iris.data
X.shape

from sklearn.preprocessing import StandardScaler
X_std = StandardScaler().fit_transform(X)
print (X std[0:5])
```

Output:

```
  [[-0.901      1.019   -1.34      -1.315]
   [-1.143     -0.132   -1.34      -1.315]
   [-1.385      0.328   -1.397     -1.315]
   [-1.507      0.098   -1.283     -1.315]
  [-1.022  1.249 -1.34  -1.315]]
The shape of Feature Matrix is - (150, 4)
```

Program:

```
print( "The shape of Feature Matrix is -",X_std.shape)
```

Output:

```
  [[-0.901      1.019 -1.34  -1.315]
   [-1.143    -0.132 -1.34  -1.315]
   [-1.385     0.328 -1.397 -1.315]
   [-1.507     0.098 -1.283 -1.315]
   [-1.022     1.249 -1.34  -1.315]]
 The shape    of Feature Matrix is - (150, 4)
```

Program:

```
eig_vals,      eig_vecs = np.linalg.eig(X_covariance_matrix) genvectors
print('Ei       n%s' %eig_vecs)
print('\n      Eigenvalues \n%s' %eig_vals)
```

Output:

```
Eigenvectors
[[ 0.521 -0.377 -0.72  0.261]
 [-0.269 -0.923  0.244 -0.124]
 [ 0.58  -0.024  0.142 -0.801]
 [ 0.565 -0.067  0.634  0.524]]

Eigenvalues
[2.938 0.92 0.148 0.021]
```

Program:

```
# Make a list of (eigenvalue, eigenvector) tuples
eig_pairs = [(np.abs(eig_vals[i]), eig_vecs[:,i]) for i in
range(len(eig_vals))]

# Sort the (eigenvalue, eigenvector) tuples from high to low


eig_pairs.sort(key=lambda x: x[0], reverse=True)

# Visually confirm that the list is correctly sorted by decreasing
eigenvalues
print('Eigenvalues in descending order:')
for i in eig_pairs:
    print(i[0])
```

Output:

Eigenvaluesindescending order: 2.938085050199995
0.9201649041624864
0.1477418210449475
0.020853862176462696

Program:

```
tot = sum(eig_vals)
var_exp = [(i / tot)*100 for i in sorted(eig_vals, reverse=True)]
cum_var_exp = np.cumsum(var_exp)
print("Variance captured by each component is \n",var_exp)
print(40 * '-')
print("Cumulative variance captured as we travel each component
\n",cum_var_exp)
```

Output:

Variance captured by each component is
[72.96244541329989, 22.850761786701753, 3.668921889282865,
0.5178709107154905]

Cumulative variance captured as we travel each component [72.962
95.813 99.482 100.]

Program:

```
print ("All Eigen Values along with Eigen Vectors")
pprint.pprint(eig_pairs)
print(40 * '-')
matrix_w = np.hstack((eig_pairs[0][1].reshape(4,1),
                      eig_pairs[1][1].reshape(4,1)))

print ('Matrix W:\n', matrix_w)
```

Output:

All Eigen Values along with Eigen Vectors [(2.938085050199995,
array([0.521, -0.269, 0.58 , 0.565])),
 (0.9201649041624864, array([-0.377, -0.923, -0.024, -0.067])),
 (0.1477418210449475, array([-0.72 , 0.244, 0.142, 0.634])),
 (0.020853862176462696, array([0.261, -0.124, -0.801, 0.524]))]

Matrix W:
 [[0.521 -0.377]
 [-0.269 -0.923]
 [0.58 -0.024]
 [0.565 -0.067]]

Program:

```
Y = X_std.dot(matrix_w)
print( Y[0:5])
```

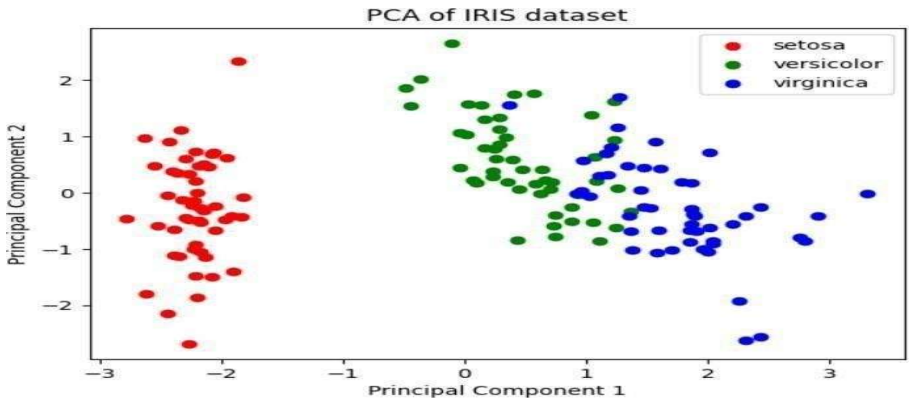
Output:

[[-2.265 -0.48]
 [-2.081 0.674]
 [-2.364 0.342]
 [-2.299 0.597]
 [-2.39 -0.647]]

Program:

```
pl.figure()
target_names = iris.target_names
y = iris.target
for c, i, target_name in zip("rgb", [0, 1, 2], target_names):
    pl.scatter(Y[y==i,0], Y[y==i,1], c=c, label=target_name)
pl.xlabel('Principal Component 1')
pl.ylabel('Principal Component 2')
pl.legend()
pl.title('PCA of IRIS dataset')
pl.show()
```

Output:


PCA of IRIS dataset
setosa
versicolor
virginica
Principal Component 2
Principal Component 1
2
1
0
−1
−2
−3
−2
−1
0
1
2
3

Practical No: 4

Aim: Generate your own feature set by combining existing set of features, or defining new ones. Feature Representation.

Theory:

Certainly! Let's use the famous Iris dataset to demonstrate feature engineering by combining existing features and creating new ones. The Iris dataset contains features about the length and width of the sepals and petals of iris flowers, and the goal is to classify them into three species.

Steps to Generate and Use New Features with the Iris Dataset

1. Load the Iris Dataset: We'll use scikit-learn's built-in Iris dataset.
2. Create New Features: Generate new features by combining or transforming the existing features.
3. Preprocess the Data: Prepare the data for model training.
4. Train and Evaluate Models: Train a model using the new feature set and evaluate its performance.

Program:

```
import pandas as pd
import numpy as np
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler, PolynomialFeatures
from sklearn.svm import SVC
from sklearn.metrics import classification_report, accuracy_score

# Load the Iris Dataset
iris = load_iris()
data = pd.DataFrame(data=iris.data, columns=iris.feature_names)
data['target'] = iris.target

# Display the first few rows of the dataset
print("Original Data:\n", data.head())

# Generate New Features
# 1. Mathematical Combinations
data['sepal_length_petal_length_sum'] = data['sepal length (cm)'] +
data['petal length (cm)']
data['sepal_width_petal_width_product'] = data['sepal width (cm)'] *
data['petal width (cm)']
```

```
# 2. Interaction Features
data['sepal_length_petal_width_ratio'] = np.where(data['petal width (cm)'] != 0, data['sepal length (cm)'] / data['petal width (cm)'], 0)

# 3. Polynomial Features (degree 2 for interaction terms) poly = PolynomialFeatures(degree=2, include_bias=False)
poly_features = poly.fit_transform(data[['sepal length (cm)', 'sepal width (cm)', 'petal length (cm)', 'petal width (cm)']]) poly_feature_names = poly.get_feature_names_out(['sepallength (cm)', 'sepal width (cm)', 'petal length (cm)', 'petal width (cm)']) poly_df = pd.DataFrame(poly_features, columns=poly_feature_names)

# Combine polynomial features with the original dataframe data = pd.concat([data, poly_df], axis=1)

# Drop the original features if necessary
data.drop(['sepal length (cm)', 'sepal width (cm)', 'petal length (cm)', 'petal width (cm)'], axis=1, inplace=True)

# Split Data into Features and Target
X = data.drop('target', axis=1) # Features y = data['target']                                    # Target

# Split Data into Training and Testing Sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

# Standardize the Data scaler= StandardScaler()
X_train = scaler.fit_transform(X_train) X_test = scaler.transform(X_test)

# Train the SVM Model with RBF Kernel (for example) model = SVC(kernel='rbf', random_state=42) model.fit(X_train, y_train)

# Evaluate the Model
y_pred= model.predict(X_test)
accuracy = accuracy_score(y_test, y_pred) report = classification_report(y_test, y_pred)

print(f"Accuracy: {accuracy:.4f}") print(f"Classification Report:\n{report}")
```

Output:

Original Data:

	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)
0	5.1	3.5	1.4	0.2
1	4.9	3.0	1.4	0.2
2	4.7	3.2	1.3	0.2
3	4.6	3.1	1.5	0.2
4	5.0	3.6	1.4	0.2

	target
0	0
1	0
2	0
3	0
4	0

Accuracy: 1.0000 Classification Report:

	precision	recall	f1-score	support
0	1.00	1.00	1.00	19
1	1.00	1.00	1.00	13
2	1.00	1.00	1.00	13
accuracy			1.00	45
macro avg	1.00	1.00	1.00	45
weighted avg	1.00	1.00	1.00	45

Practical No:5

Aim: Generate samples of a normal distribution with specific parameters with respect to Mean and Covariance

Theory:

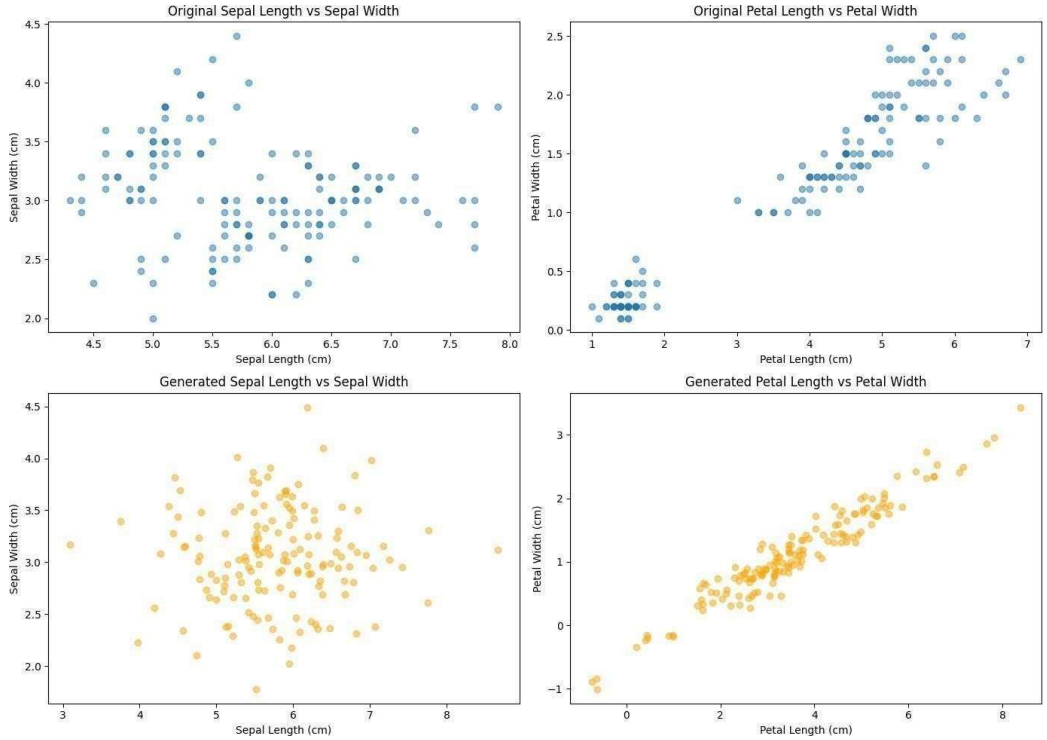
- **Load the Iris Dataset**:
  - The Iris dataset is loaded and converted into a pandas DataFrame for easy manipulation.
- **Calculate Mean and Covariance**:
  - Compute the mean and covariance matrix of the features in the Iris dataset using mean() and cov().
- **Generate Samples**:
  - Use np.random.multivariate_normalto generate samples from a multivariate normal distribution with the calculated mean and covariance matrix.
- **Visualization**:
  - The generated samples are plotted alongside the original data for visual comparison. This helps to understand how well the generated samples resemble the original distribution.

Program:

```
import numpy as np
import pandas as pd
from sklearn.datasets import load_iris
import matplotlib.pyplot as plt

# Load the Iris Dataset
iris = load_iris()
data = pd.DataFrame(data=iris.data, columns=iris.feature_names)

# Display basic statistics
print("Data Mean:\n", data.mean())
print("Data Covariance Matrix:\n", data.cov())

# Extract mean and covariance
mean = data.mean().values
covariance_matrix = data.cov().values
```

```
# Generate samples from a normal distribution with the given mean and covariance
num_samples = 150  # Number of samples to generate
samples = np.random.multivariate_normal(mean, covariance_matrix, num_samples)

# Create a DataFrame for the generated samples generated_data =
pd.DataFrame(samples, columns=data.columns)

# Display the first few rows of the generated data print("Generated
Samples:\n", generated_data.head())

# Plotting the original and generated data for comparison fig, axs =
plt.subplots(2, 2, figsize=(14, 10))

# Original Data Plots
axs[0, 0].scatter(data['sepal length (cm)'], data['sepal width (cm)'], alpha=0.5)
axs[0, 0].set_title('Original Sepal Length vs Sepal Width') axs[0,
0].set_xlabel('Sepal Length (cm)')
axs[0, 0].set_ylabel('Sepal Width (cm)')

axs[0, 1].scatter(data['petal length (cm)'], data['petal width (cm)'], alpha=0.5)
axs[0, 1].set_title('Original Petal Length vs Petal Width') axs[0,
1].set_xlabel('Petal Length (cm)')
axs[0, 1].set_ylabel('Petal Width (cm)')

# Generated Data Plots
axs[1, 0].scatter(generated_data['sepal length (cm)'], generated_data['sepalwidth
(cm)'], alpha=0.5, color='orange') axs[1, 0].set_title('Generated Sepal Length vs Sepal
Width') axs[1, 0].set_xlabel('Sepal Length (cm)')
axs[1, 0].set_ylabel('Sepal Width (cm)')

axs[1, 1].scatter(generated_data['petal length (cm)'], generated_data['petalwidth (cm)'],
alpha=0.5, color='orange') axs[1, 1].set_title('Generated Petal Length vs Petal Width')
axs[1, 1].set_xlabel('Petal Length (cm)')
axs[1, 1].set_ylabel('Petal Width (cm)')

plt.tight_layout()
plt.show()
```

Output:


Original Sepal Length vs Sepal Width
Sepal Width (cm)
Sepal Length (cm)
Original Petal Length vs Petal Width
Petal Width (cm)
Petal Length (cm)
Generated Sepal Length vs Sepal Width
Sepal Width (cm)
Sepal Length (cm)
Generated Petal Length vs Petal Width
Petal Width (cm)
Petal Length (cm)

Practical No: 6

Aim: Implement Linear Perceptron Learning algorithm

Theory:

1. **Load and Prepare the Dataset:**
   - Load the Iris dataset and filter it to include only two classes for binary classification.
   - Standardize the features for better convergence.
2. **Define the Perceptron Model:**
   - Perceptron class:
     - ___init_: Initializes the learning rate and number of iterations.
     - fit: Trains the Perceptron by iterating through epochs and updating weights based on misclassification.
     - predict: Makes predictions based on the learned weights.
     - _net_input: Calculates the net input to the activation function.
3. **Train and Evaluate the Model:**
   - Train the Perceptron using the training data.
   - Evaluate the model using accuracy and a classification report.
   - Plot training errors over epochs to visualize the convergence of the model.

This implementation provides a basic understanding of the Perceptron algorithm and how it can be applied to the Iris dataset. For more complex datasets or multi-class problems, consider using more sophisticated algorithms or extending the Perceptron model.

Program:

```
import numpy as np
import pandas as pd
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import accuracy_score, classification_report
import matplotlib.pyplot as plt

# Load the Iris Dataset
iris = load_iris()
data = pd.DataFrame(data=iris.data, columns=iris.feature_names)
data['target'] = iris.target

# For binary classification: Iris-setosa vs. Iris-versicolor
data = data[data['target'] != 2] # Exclude Iris-virginica
data['target'] = data['target'].map({0: 0, 1: 1}) # Map to binary
target
```

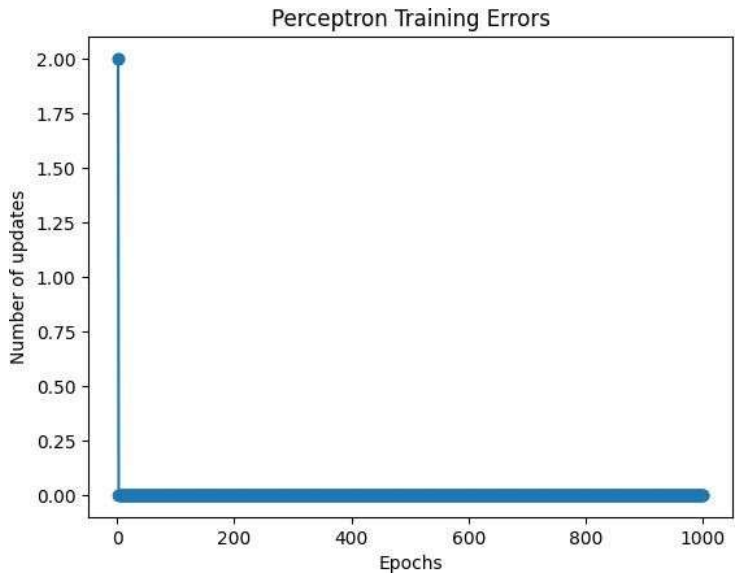
```
# Split Data into Features and Target X =
data.drop('target', axis=1).values y =
data['target'].values

# Split Data into Training and Testing Sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

# Standardize the Data scaler=
StandardScaler()
X_train = scaler.fit_transform(X_train) X_test =
scaler.transform(X_test)

# Define the Perceptron Algorithm class
Perceptron:
    def init (self, learning_rate=0.01, n_iter=1000):
        self.learning_rate = learning_rate self.n_iter = n_iter

    def fit(self, X, y):
        self.weights = np.zeros(X.shape[1] + 1)  # Including bias term self.errors = []

        for _inrange(self.n_iter): errors = 0
            for xi, target in zip(X, y):
                update = self.learning_rate * (target - self.predict(xi))
                self.weights[1:] += update * xi
                self.weights[0] += update errors +=
                int(update != 0.0)
            self.errors.append(errors)

    def predict(self, X):
        return np.where(self._net_input(X) >= 0.0, 1, 0)

    def _net_input(self, X):
        returnnp.dot(X, self.weights[1:]) + self.weights[0]

# Train the Perceptron Model
perceptron=Perceptron(learning_rate=0.01, n_iter=1000) perceptron.fit(X_train, y_train)

# Make Predictions
y_pred= perceptron.predict(X_test)
```

```
# Evaluate the Model
accuracy = accuracy_score(y_test, y_pred)
report = classification_report(y_test, y_pred)

print(f"Accuracy: {accuracy:.4f}")
print(f"Classification Report:\n{report}")

# Plot the Training Errors
plt.plot(range(1, len(perceptron.errors) + 1), perceptron.errors,
marker='o')
plt.xlabel('Epochs')
plt.ylabel('Number of updates')
plt.title('Perceptron Training Errors')
plt.show()
```

Output:

Accuracy: 1.0000 Classification Report:

	precision	recall	f1-score	support
0	1.00	1.00	1.00	17
1	1.00	1.00	1.00	13
accuracy			1.00	30
macro avg	1.00	1.00	1.00	30
weighted avg	1.00	1.00	1.00	30



Practical No: 7

Aim: Build IRIS flower classification in python using pattern recognition models

Theory:

We'll use the following pattern recognition models:

1. **Logistic Regression**
2. **k-Nearest Neighbors (k-NN)**
3. **Support Vector Machine (SVM)**
4. **Decision Tree**
5. **Random Forest**

Here's a step-by-step guide to implement and evaluate these models using the Iris dataset in Python.

Step-by-Step Implementation

1. **Load and Prepare the Dataset**:
   - Load the Iris dataset.
   - Preprocess the data (e.g., standardization).
   - Split the data into training and testing sets.
2. **Define and Train Models**:
   - Implement each model and fit it to the training data.
3. **Evaluate the Models**:
   - Assess the performance of each model using accuracy and other metrics.
4. **Compare Results**:
   - Compare the classification accuracy and other metrics of each model.

Program:

```
import numpy as np
import pandas as pd
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LogisticRegression
from sklearn.neighbors import KNeighborsClassifier
from sklearn.svm import SVC
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score, classification_report
import matplotlib.pyplot as plt

# Load the Iris Dataset
iris = load_iris()
```

```
X = iris.data y=
iris.target

# Split Data into Training and Testing Sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

# Standardize the Data scaler=
StandardScaler()
X_train = scaler.fit_transform(X_train) X_test =
scaler.transform(X_test)

# Definethe Models models = {
    "Logistic Regression": LogisticRegression(max_iter=200), "k-Nearest
    Neighbors": KNeighborsClassifier(),
    "Support Vector Machine": SVC(), "Decision Tree":
    DecisionTreeClassifier(), "Random Forest":
    RandomForestClassifier()
}

# Train and Evaluate Models results = {}
formodel_name, model inmodels.items(): model.fit(X_train, y_train)
    y_pred= model.predict(X_test)
    accuracy = accuracy_score(y_test, y_pred) report =
    classification_report(y_test, y_pred,
target_names=iris.target_names)
    results[model_name] = {
        "accuracy":accuracy,
        "report": report
    }

# Print Results
for model_name, metrics in results.items(): print(f"Model:
    {model_name}") print(f"Accuracy:
    {metrics['accuracy']:.4f}")
    print(f"Classification Report:\n{metrics['report']}") print("-" * 60)

# Plotting the results model_names =
list(results.keys())
accuracies = [results[model_name]['accuracy'] for model_name in model_names]
```

```
plt.figure(figsize=(10, 6))
plt.bar(model_names, accuracies, color='skyblue')
plt.xlabel('Model')
plt.ylabel('Accuracy')
plt.title('Model Comparison')
plt.xticks(rotation=45)
plt.ylim(0, 1)
plt.show()
```

Output:

Model: Logistic Regression Accuracy:
1.0000
Classification Report:

	precision	recall	f1-score	support
setosa	1.00	1.00	1.00	19
versicolor	1.00	1.00	1.00	13
virginica	1.00	1.00	1.00	13
accuracy			1.00	45
macro avg	1.00	1.00	1.00	45
weighted avg	1.00	1.00	1.00	45

Model: k-Nearest Neighbors Accuracy:
1.0000 Classification Report:

	precision	recall	f1-score	support
setosa	1.00	1.00	1.00	19
versicolor	1.00	1.00	1.00	13
virginica	1.00	1.00	1.00	13
accuracy			1.00	45
macroavg	1.00	1.00	1.00	45
weighted avg	1.00	1.00	1.00	45

Model: Support Vector Machine
Accuracy: 1.0000 Classification
Report:

	precision	recall	f1-score	support
setosa	1.00	1.00	1.00	19
versicolor	1.00	1.00	1.00	13
virginica	1.00	1.00	1.00	13
accuracy			1.00	45
macro avg	1.00	1.00	1.00	45
weighted avg	1.00	1.00	1.00	45

Model: Decision Tree Accuracy:
1.0000 Classification Report:

	precision	recall	f1-score	support
setosa	1.00	1.00	1.00	19
versicolor	1.00	1.00	1.00	13
virginica	1.00	1.00	1.00	13
accuracy			1.00	45
macro avg	1.00	1.00	1.00	45
weighted avg	1.00	1.00	1.00	45

Model: Random Forest

Parul University

FACULTY OF ENGINEERING AND TECHNOLOGY
BACHELOR OF TECHNOLOGY

Pattern recognition

(303105482)

7TH SEMESTER

Computer Science & Engineering Department

Lab Manual

Parul® University
VADODARA, GUJARAT

NAAC A++

FACULTY OF ENGINEERING AND TECHNOLOGY
Subject Name: Pattern Recognition Laboratory
Subject Code: 30105482
B.Tech. 4rd year 7TH semester

CERTIFICATE

This is to certify that Student **BOPPANA CHANDRASHEKAR,** with Enrollment No. **2203031240822** has Successfully completed his Laboratory experiments in **Pattern Recognition (303105482)** from the department of **ARTIFICIAL INTELLINGENCE AND DATA SCIENCE** during the academic year **2024-2025**.

योगः कर्मसु कौशलम्
PARUL UNIVERSITY

Date of Submission: ..... Staff In Charge: ……..

Head of Department: .....

TABLE OF CONTENT

Practical No: 1

Aim:. Implementation of Gradient Descent.

What is Gradient Descent?

Gradient Descent stands as a cornerstone orchestrating the intricate dance of model optimization. At its core, it is a numerical optimization algorithm that aims to find the optimal parameters—weights and biases—of a neural network by minimizing a defined cost function.

Gradient Descent (GD) is a widely used optimization algorithm in machine learning and deep learning that minimises the cost function of a neural network model during training. It works by iteratively adjusting the weights or parameters of the model in the direction of the negative gradient of the cost function until the minimum of the cost function is reached.

The learning happens during the backpropagation while training the neural network-based model. There is a term known as Gradient Descent, which is used to optimize the weight and biases based on the cost function. The cost function evaluates the difference between the actual and predicted outputs.

Gradient Descent is a fundamental optimization algorithm in machine learning used to minimize the cost or loss function during model training.

- It iteratively adjusts model parameters by moving in the direction of the steepest decrease in the cost function.
- The algorithm calculates gradients, representing the partial derivatives of the cost function concerning each parameter.

These gradients guide the updates, ensuring convergence towards the optimal parameter values that yield the lowest possible cost.

Gradient Descent is versatile and applicable to various machine learning models, including linear regression and neural networks. Its efficiency lies in navigating the parameter space efficiently, enabling models to learn patterns and make accurate predictions. Adjusting the learning rate is crucial to balance convergence speed and avoiding overshooting the optimal solution.

Gradient descent is an iterative optimization algorithm for finding the local minimum of a function.

To find the local minimum of a function using gradient descent, we must take steps proportional to the negative of the gradient (move away from the gradient) of the function at the current point. If we take steps proportional to the positive of the gradient (moving towards the gradient), we will approach a local maximum of the function, and the procedure is called **Gradient Ascent.**

DATASET TAKEN:- Globalisation Index

The dataset contains 167 rows and 14 columns, with the following features:

1. Country: The name of the country (categorical feature).
2. Average Score: The average score of various factors for each country (numeric, float).
3. Safety Security: The score related to safety and security (numeric, float).
4. Personal Freedom: The score for personal freedom in each country (numeric, float).
5. Governance: The score for governance (numeric, float).
6. Social Capital: The score for social capital (numeric, float).
7. Investment Environment: The score related to the investment environment (numeric, float).
8. Enterprise Conditions: The score for enterprise conditions (numeric, float).
9. Market Access Infrastructure: The score for market access and infrastructure (numeric, float).
10. Economic Quality: The score for economic quality (numeric, float).
11. Living Conditions: The score for living conditions (numeric, float).
12. Health: The score for health conditions (numeric, float).
13. Education: The score for education (numeric, float).
14. Natural Environment: The score for natural environment quality (numeric, float).

All features except for "Country" are numeric and indicate scores related to various aspects of each country's conditions.

Program:

```
import numpy as np
import matplotlib.pyplot as plt

# Generatesomedata
np.random.seed(42)
X = 2 * np.random.rand(100, 1)
y = 4 + 3 * X + np.random.randn(100, 1)

# Function to compute the cost def
compute_cost(X, y, theta):
        m = len(y)
        predictions=X.dot(theta)
        cost = (1/2*m) * np.sum(np.square(predictions - y)) return cost

# Function to perform gradient descent
def gradient_descent(X, y, theta, learning_rate, iterations): m = len(y)
        cost_history=np.zeros(iterations)

        for i in range(iterations): predictions= X.dot(theta)
                theta = theta - (1/m) * learning_rate * (X.T.dot(predictions - cost_history[i] =

y))          compute_cost(X, y, theta)
```

retu rn theta, cost_history

Add x0 = 1 to each instance X_b =
np.c_[np.ones((100, 1)), X]

# Initialize theta
theta = np.random.randn(2, 1)

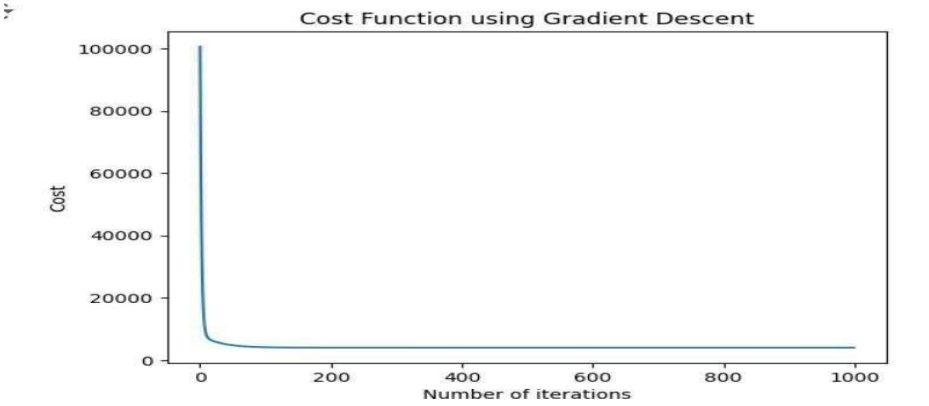
Sethyperparameters learning_rate =
0.1
iterations= 1000

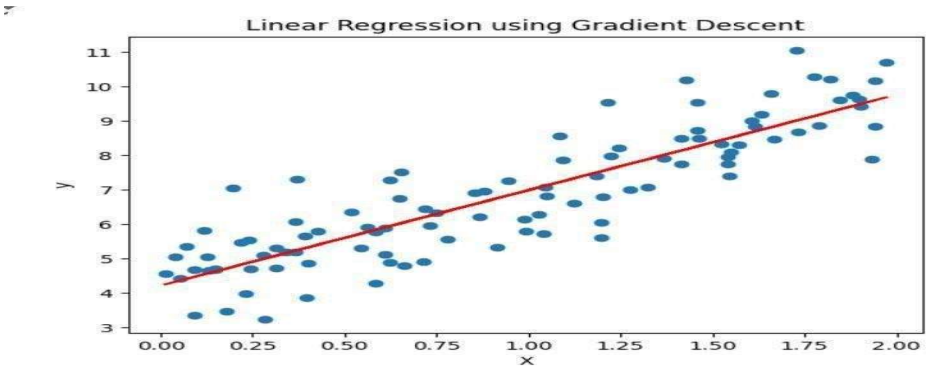
Perform gradient descent
theta, cost_history = gradient_descent(X_b, y, theta, learning_rate, iterations)

Plot the cost function plt.plot(range(iterations), cost_history)

```
plt.xlabel("Number of iterations")
plt.ylabel("Cost")
plt.title("Cost Function using Gradient Descent")
plt.show()

# Plot the data and the regression line
plt.scatter(X, y)
plt.plot(X, X_b.dot(theta), color='red')
plt.xlabel("X")
plt.ylabel("y")
plt.title("Linear Regression using Gradient Descent")
t.show()

print("Theta (parameters):", theta)
print("Final cost:", cost_history[-1])
```

Output:






```
Theta (parameters): [[4.21509609]
 [2.77011344]]
Final cost: 4032.922819835273
```

Practical No: 2

Aim: Implementation of Linear Regression using Gradient Descent

What is Linear Gradient Descent?

Linear Gradient Descent is an optimization algorithm used to minimize the cost function in linear regression. It's an iterative method that adjusts the model's parameters to reduce the difference between predicted and actual values. How
does it work?

1. Initialization: Initialize the model's parameters (weights and bias) randomly.
2. Prediction: Use the current parameters to make predictions on the training data.
3. Error calculation: Calculate the error between predicted and actual values using a loss function
(e.g., Mean Squared Error).
4. Gradient calculation: Calculate the gradient of the loss function with respect to each parameter.
5. Parameter update: Update each parameter by subtracting a fraction of the gradient (learning
rate) from the current value.
6. Repeat: Repeat steps 2-5 until convergence or a stopping criterion is reached.

Key concepts:

- Learning rate: A hyperparameter that controls the step size of each update.
- Gradient: The direction of the steepest ascent of the loss function.
- Convergence: The point at which the algorithm stops improving the model's performance.

Advantages:

- Simple to implement
- Fast convergence
- Can be used for large datasets

Disadvantages:

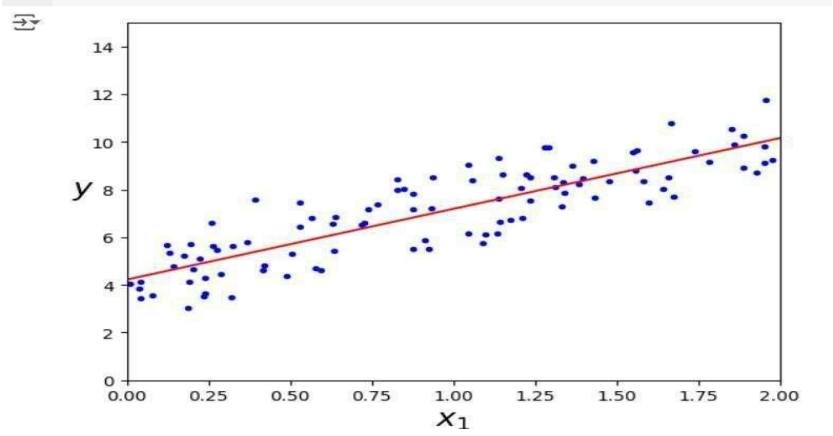
- May get stuck in local minima
- Requires careful tuning of the learning rate

Program:

Program:

```
import numpy as np
import matplotlib.pyplot as plt

# Step 2: Generate sample data
np.random.seed(0)
X = 2 * np.random.rand(100, 1)
y = 4 + 3 * X + np.random.randn(100, 1)

# Step 3: Add bias term
X_b = np.c_[np.ones((100, 1)), X]  # add x0 = 1 to each instance

# Step 4: Initialize parameters
theta = np.random.randn(2, 1)

# Step 5: Define hyperparameters
eta = 0.1 # learning rate
n_iterations = 1000

# Step 6: Gradient Descent
for iteration in range(n_iterations):
    gradients = 2/100 * X_b.T.dot(X_b.dot(theta) - y)
    theta = theta - eta * gradients

# Step 7: Prediction
X_new = np.array([[0], [2]])
X_new_b = np.c_[np.ones((2, 1)), X_new]
y_predict = X_new_b.dot(theta)

# Step 8: Plotting
plt.plot(X, y, "b.")
plt.plot(X_new, y_predict, "r-")
plt.xlabel("$x_1$", fontsize=18)
plt.ylabel("$y$", rotation=0, fontsize=18)
plt.axis([0, 2, 0, 15])
plt.show()
```

Output:



Practical

Aim: decision tree classificationalgorithm.

Theory:

Decision Tree is a Supervised learning technique that can be used for both classification and Regression problems, but mostly it is preferred for solving Classification problems. It is a tree- structured classifier, where internal nodes represent the features of a dataset, branches represent the decision rules and each leaf node represents the outcome. It is a graphical representation for getting all the possible solutions to a problem/decision based on given conditions.

Decision Tree Terminologies

- Root Node: Root node is from where the decision tree starts. It represents the entire dataset, which further gets divided into two or more homogeneous sets.
- Leaf Node: Leaf nodes are the final output node, and the tree cannot be segregated further after getting a leaf node.
- Splitting: Splitting is the process of dividing the decision node/root node into sub- nodes according to the given conditions.
- Branch/Sub Tree: A tree formed by splitting the tree.
- Pruning: Pruning is the process of removing the unwanted branches from the tree.
- Parent/Child node: The root node of the tree is called the parent node, and other nodes are called the child nodes.

Program:

```
#decision tree:
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score, classification_report
import matplotlib.pyplot as plt
from sklearn import tree
data = {
    'weather': ['Sunny', 'Sunny', 'cloudy', 'Rainy', 'Rainy', 'Rainy',
'cloudy', 'Sunny', 'Sunny', 'Rainy', 'Sunny', 'cloudy',
'cloudy','rain'],
    'Temperature': ['Hot', 'Hot', 'Hot', 'Mild', 'Cool', 'Cool',
'Cool', 'Mild', 'Cool', 'Mild', 'Mild', 'Mild', 'Hot','Mild'],
    'Humidity': ['High', 'High', 'High', 'High', 'Normal', 'Normal',
'Normal', 'High', 'Normal', 'Normal','Normal','High', 'Normal','High'],
```

```
    'Windy': ['weak','strong','weak','weak','weak','strong','strong','weak','weak','
weak','strong','strong','weak','strong'],
    'Playfootball':['No', 'No', 'Yes', 'Yes', 'Yes', 'No', 'Yes',
'No', 'Yes', 'Yes', 'Yes', 'Yes','yes', 'No']
}

df = pd.DataFrame(data)
df_encoded = pd.get_dummies(df, drop_first=True)
print(df_encoded)
X = df_encoded.drop('Playfootball_Yes', axis=1) y =
df_encoded['Playfootball_Yes']

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)
clf=DecisionTreeClassifier()
clf.fit(X_train, y_train) y_pred =
clf.predict(X_test)

print("Accuracy:", accuracy_score(y_test, y_pred)) print("Classification Report:\n",
classification_report(y_test, y_pred))
plt.figure(figsize=(12,8))
tree.plot_tree(clf, feature_names=X.columns, class_names=['No', 'Yes'], filled=True)
plt.show()
```

Output:

	weather_Sunny	weather_cloudy	weather_rain	Temperature_Hot	\
0	True	False	False	True	
1	True	False	False	True	
2	False	True	False	True	
3	False	False	False	False	
4	False	False	False	False	
5	False	False	False	False	
6	False	True	False	False	
7	True	False	False	False	
8	True	False	False	False	
9	False	False	False	False	
10	True	False	False	False	
11	False	True	False	False	
12	False	True	False	True	
13	False	False	True	False	

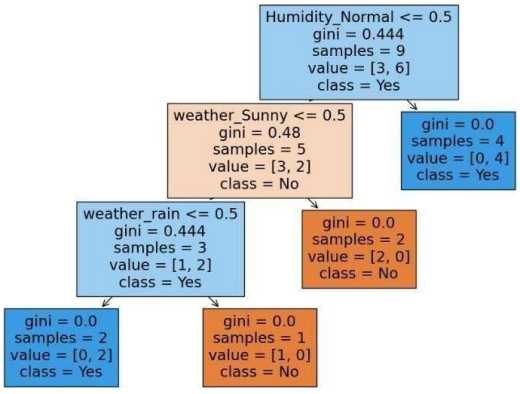
	Temperature_Mild	Humidity_Normal	Windy_weak	Playfootball_Yes	\
0	False	False	True	False	
1	False	False	False	False	
2	False	False	True	True	
3	True	False	True	True	
4	False	True	True	True	

5	False	True	False	False
6	False	True	False	True
7	True	False	True	False
8	False	True	True	True
9	True	True	True	True
10	True	True	False	True
11	True	False	False	True
12	False	True	True	False
13	True	False	False	False

	Playfootball_yes
0	False
1	False
2	False
3	False
4	False
5	False
6	False
7	False
8	False
9	False
10	False
11	False
12	True
13	False

Accuracy: 0.6 Classification Report:

	precision	recall	f1-score	support
False	1.00	0.33	0.50	3
True	0.50	1.00	0.67	2
accuracy			0.60	5
macro avg	0.75	0.67	0.58	5
weighted avg	0.80	0.60	0.57	5



Practical No: 3

Aim: Comparison of Classification Accuracy of SVM for given dataset

Theory:

Support Vector Machines (SVMs) are a set of supervised learning methods used for classification, regression, and outliers detection. The primary goal of the SVM algorithm is to find a hyperplane in an N-dimensional space (N is the number of features) that distinctly classifies the data points.

Key concepts:

- Hyperplane: In SVM, a hyperplane is a decision boundary that helps classify the data points. Data points falling on either side of the hyperplane can be attributed to different classes.
- Support Vectors: Data points that are closer to the hyperplane and influence the position and orientation of the hyperplane. These are the critical elements of the dataset.
- Margin: The distance between the hyperplane and the closest data points from either class. SVM aims to maximize this margin.

- Import Libraries: Import necessary libraries such as numpy, pandas, matplotlib, and sklearn.
- Load the Dataset: Load and explore the dataset to understand its structure and features.
- Preprocess the Data: Handle missing values, encode categorical variables, and split the data into training and testing sets.
- Train the SVM Model: Train the SVM model using different kernel functions (e.g., linear, polynomial, RBF).
- Evaluate the Model: Evaluate the model's performance using classification accuracy and other relevant metrics.
- Compare Results: Compare the classification accuracy of the SVM model with different kernel functions.

Program:

```
#Comparison of Classification Accuracy of SVM for given dataset using
iris dataset

import numpy as np
from sklearn import datasets
from sklearn.model_selection import train_test_split
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score
```

```
# Load Iris dataset
iris = datasets.load_iris()
X = iris.data
y = iris.target

# Split the data into training and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.3, random_state=42)

# Define SVM models with different kernels
kernels = ['linear', 'poly', 'rbf', 'sigmoid']
accuracies = {}

for kernel in kernels:
    # Initialize and train the SVM model
    model = SVC(kernel=kernel, random_state=42)
    model.fit(X_train, y_train)

    # Make predictions
    y_pred = model.predict(X_test)

    # Calculate accuracy
    accuracy = accuracy_score(y_test, y_pred)
    accuracies[kernel] = accuracy
    print(f'Accuracy with {kernel} kernel: {accuracy:.4f}')

# Display results
print("\nComparison of Classification Accuracy:")
for kernel, accuracy in accuracies.items():
    print(f"Kernel: {kernel} - Accuracy: {accuracy:.4f}")
```

Output:

Accuracy with linear kernel: 1.0000 Accuracy with poly kernel: 0.9778 Accuracy with rbf kernel: 1.0000 Accuracy with sigmoid kernel: 0.2222

Comparisonof Classification Accuracy: Kernel: linear - Accuracy: 1.0000 Kernel: poly - Accuracy: 0.9778 Kernel: rbf - Accuracy: 1.0000

Kernel: sigmoid - Accuracy: 0.2222

Practical

Aim: Principal compound analysis.

Theory:

What is Principal Component Analysis (PCA)?

The Principal Component Analysis is a popular unsupervised learning technique for reducing the dimensionality of large data sets. It increases interpretability yet, at the same time, it minimizes information loss. It helps to find the most significant features in a dataset and makes the data easy for plotting in 2D and 3D. PCA helps in finding a sequence of linear combinations of variables.

Steps for PCA Algorithm

1. Standardize the data: PCA requires standardized data, so the first step is to standardize the data to ensure that all variables have a mean of 0 and a standard deviation of 1.
2. Calculate the covariance matrix: The next step is to calculate the covariance matrix of the standardized data. This matrix shows how each variable is related to every other variable in the dataset.
3. Calculate the eigenvectors and eigenvalues: The eigenvectors and eigenvalues of the covariance matrix are then calculated. The eigenvectors represent the directions in which the data varies the most, while the eigenvalues represent the amount of variation along each eigenvector.
4. Choose the principal components: The principal components are the eigenvectors with the highest eigenvalues. These components represent the directions in which the data varies the most and are used to transform the original data into a lower-dimensional space.
5. Transform the data: The final step is to transform the original data into the lower- dimensional space defined by the principal components.

Program:

```
import numpy as np
import pandas as pd
import pprint
import matplotlib.pyplot as plt
from sklearn.datasets import load_iris
import pylab as pl
iris = load_iris()
iris_df = pd.DataFrame(iris.data,columns=[iris.feature_names])
```

```
iris_df.head()
X = iris.data
X.shape

from sklearn.preprocessing import StandardScaler
X_std = StandardScaler().fit_transform(X)
print (X std[0:5])
```

Output:

```
  [[-0.901      1.019   -1.34      -1.315]
   [-1.143     -0.132   -1.34      -1.315]
   [-1.385      0.328   -1.397     -1.315]
   [-1.507      0.098   -1.283     -1.315]
  [-1.022  1.249 -1.34  -1.315]]
The shape of Feature Matrix is - (150, 4)
```

Program:

```
print( "The shape of Feature Matrix is -",X_std.shape)
```

Output:

```
  [[-0.901      1.019 -1.34  -1.315]
   [-1.143    -0.132 -1.34  -1.315]
   [-1.385     0.328 -1.397 -1.315]
   [-1.507     0.098 -1.283 -1.315]
   [-1.022     1.249 -1.34  -1.315]]
 The shape    of Feature Matrix is - (150, 4)
```

Program:

```
eig_vals,      eig_vecs = np.linalg.eig(X_covariance_matrix) genvectors
print('Ei       n%s' %eig_vecs)
print('\n      Eigenvalues \n%s' %eig_vals)
```

Output:

```
Eigenvectors
[[ 0.521 -0.377 -0.72  0.261]
 [-0.269 -0.923  0.244 -0.124]
 [ 0.58  -0.024  0.142 -0.801]
 [ 0.565 -0.067  0.634  0.524]]

Eigenvalues
[2.938 0.92 0.148 0.021]
```

Program:

```
# Make a list of (eigenvalue, eigenvector) tuples
eig_pairs = [(np.abs(eig_vals[i]), eig_vecs[:,i]) for i in
range(len(eig_vals))]

# Sort the (eigenvalue, eigenvector) tuples from high to low


eig_pairs.sort(key=lambda x: x[0], reverse=True)

# Visually confirm that the list is correctly sorted by decreasing
eigenvalues
print('Eigenvalues in descending order:')
for i in eig_pairs:
    print(i[0])
```

Output:

Eigenvaluesindescending order: 2.938085050199995
0.9201649041624864
0.1477418210449475
0.020853862176462696

Program:

```
tot = sum(eig_vals)
var_exp = [(i / tot)*100 for i in sorted(eig_vals, reverse=True)]
cum_var_exp = np.cumsum(var_exp)
print("Variance captured by each component is \n",var_exp)
print(40 * '-')
print("Cumulative variance captured as we travel each component
\n",cum_var_exp)
```

Output:

Variance captured by each component is
[72.96244541329989, 22.850761786701753, 3.668921889282865,
0.5178709107154905]

Cumulative variance captured as we travel each component [72.962
95.813 99.482 100.]

Program:

```
print ("All Eigen Values along with Eigen Vectors")
pprint.pprint(eig_pairs)
print(40 * '-')
matrix_w = np.hstack((eig_pairs[0][1].reshape(4,1),
                      eig_pairs[1][1].reshape(4,1)))

print ('Matrix W:\n', matrix_w)
```

Output:

All Eigen Values along with Eigen Vectors [(2.938085050199995,
array([0.521, -0.269, 0.58 , 0.565])),
 (0.9201649041624864, array([-0.377, -0.923, -0.024, -0.067])),
 (0.1477418210449475, array([-0.72 , 0.244, 0.142, 0.634])),
 (0.020853862176462696, array([0.261, -0.124, -0.801, 0.524]))]

Matrix W:
 [[0.521 -0.377]
 [-0.269 -0.923]
 [0.58 -0.024]
 [0.565 -0.067]]

Program:

```
Y = X_std.dot(matrix_w)
print( Y[0:5])
```

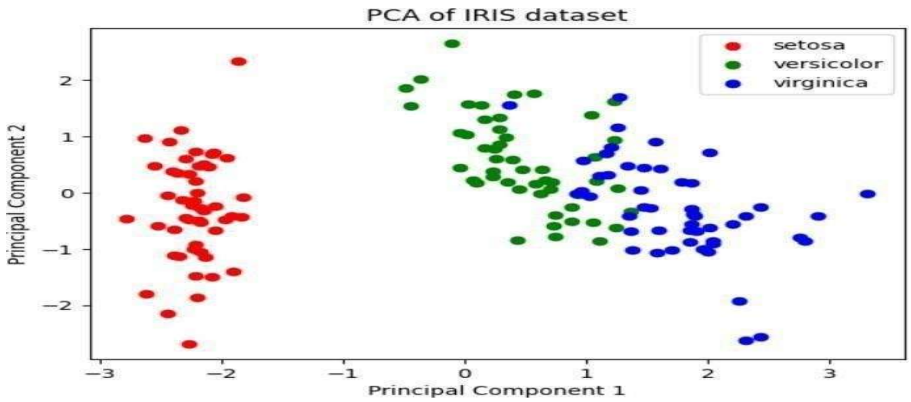
Output:

[[-2.265 -0.48]
 [-2.081 0.674]
 [-2.364 0.342]
 [-2.299 0.597]
 [-2.39 -0.647]]

Program:

```
pl.figure()
target_names = iris.target_names
y = iris.target
for c, i, target_name in zip("rgb", [0, 1, 2], target_names):
    pl.scatter(Y[y==i,0], Y[y==i,1], c=c, label=target_name)
pl.xlabel('Principal Component 1')
pl.ylabel('Principal Component 2')
pl.legend()
pl.title('PCA of IRIS dataset')
pl.show()
```

Output:


PCA of IRIS dataset
setosa
versicolor
virginica
Principal Component 2
Principal Component 1
2
1
0
−1
−2
−3
−2
−1
0
1
2
3

Practical No: 4

Aim: Generate your own feature set by combining existing set of features, or defining new ones. Feature Representation.

Theory:

Certainly! Let's use the famous Iris dataset to demonstrate feature engineering by combining existing features and creating new ones. The Iris dataset contains features about the length and width of the sepals and petals of iris flowers, and the goal is to classify them into three species.

Steps to Generate and Use New Features with the Iris Dataset

1. Load the Iris Dataset: We'll use scikit-learn's built-in Iris dataset.
2. Create New Features: Generate new features by combining or transforming the existing features.
3. Preprocess the Data: Prepare the data for model training.
4. Train and Evaluate Models: Train a model using the new feature set and evaluate its performance.

Program:

```
import pandas as pd
import numpy as np
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler, PolynomialFeatures
from sklearn.svm import SVC
from sklearn.metrics import classification_report, accuracy_score

# Load the Iris Dataset
iris = load_iris()
data = pd.DataFrame(data=iris.data, columns=iris.feature_names)
data['target'] = iris.target

# Display the first few rows of the dataset
print("Original Data:\n", data.head())

# Generate New Features
# 1. Mathematical Combinations
data['sepal_length_petal_length_sum'] = data['sepal length (cm)'] +
data['petal length (cm)']
data['sepal_width_petal_width_product'] = data['sepal width (cm)'] *
data['petal width (cm)']
```

```
# 2. Interaction Features
data['sepal_length_petal_width_ratio'] = np.where(data['petal width (cm)'] != 0, data['sepal length (cm)'] / data['petal width (cm)'], 0)

# 3. Polynomial Features (degree 2 for interaction terms) poly = PolynomialFeatures(degree=2, include_bias=False)
poly_features = poly.fit_transform(data[['sepal length (cm)', 'sepal width (cm)', 'petal length (cm)', 'petal width (cm)']]) poly_feature_names = poly.get_feature_names_out(['sepallength (cm)', 'sepal width (cm)', 'petal length (cm)', 'petal width (cm)']) poly_df = pd.DataFrame(poly_features, columns=poly_feature_names)

# Combine polynomial features with the original dataframe data = pd.concat([data, poly_df], axis=1)

# Drop the original features if necessary
data.drop(['sepal length (cm)', 'sepal width (cm)', 'petal length (cm)', 'petal width (cm)'], axis=1, inplace=True)

# Split Data into Features and Target
X = data.drop('target', axis=1) # Features y = data['target']                                    # Target

# Split Data into Training and Testing Sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

# Standardize the Data scaler= StandardScaler()
X_train = scaler.fit_transform(X_train) X_test = scaler.transform(X_test)

# Train the SVM Model with RBF Kernel (for example) model = SVC(kernel='rbf', random_state=42) model.fit(X_train, y_train)

# Evaluate the Model
y_pred= model.predict(X_test)
accuracy = accuracy_score(y_test, y_pred) report = classification_report(y_test, y_pred)

print(f"Accuracy: {accuracy:.4f}") print(f"Classification Report:\n{report}")
```

Output:

Original Data:

	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)
0	5.1	3.5	1.4	0.2
1	4.9	3.0	1.4	0.2
2	4.7	3.2	1.3	0.2
3	4.6	3.1	1.5	0.2
4	5.0	3.6	1.4	0.2

	target
0	0
1	0
2	0
3	0
4	0

Accuracy: 1.0000 Classification Report:

	precision	recall	f1-score	support
0	1.00	1.00	1.00	19
1	1.00	1.00	1.00	13
2	1.00	1.00	1.00	13
accuracy			1.00	45
macro avg	1.00	1.00	1.00	45
weighted avg	1.00	1.00	1.00	45

Practical No:5

Aim: Generate samples of a normal distribution with specific parameters with respect to Mean and Covariance

Theory:

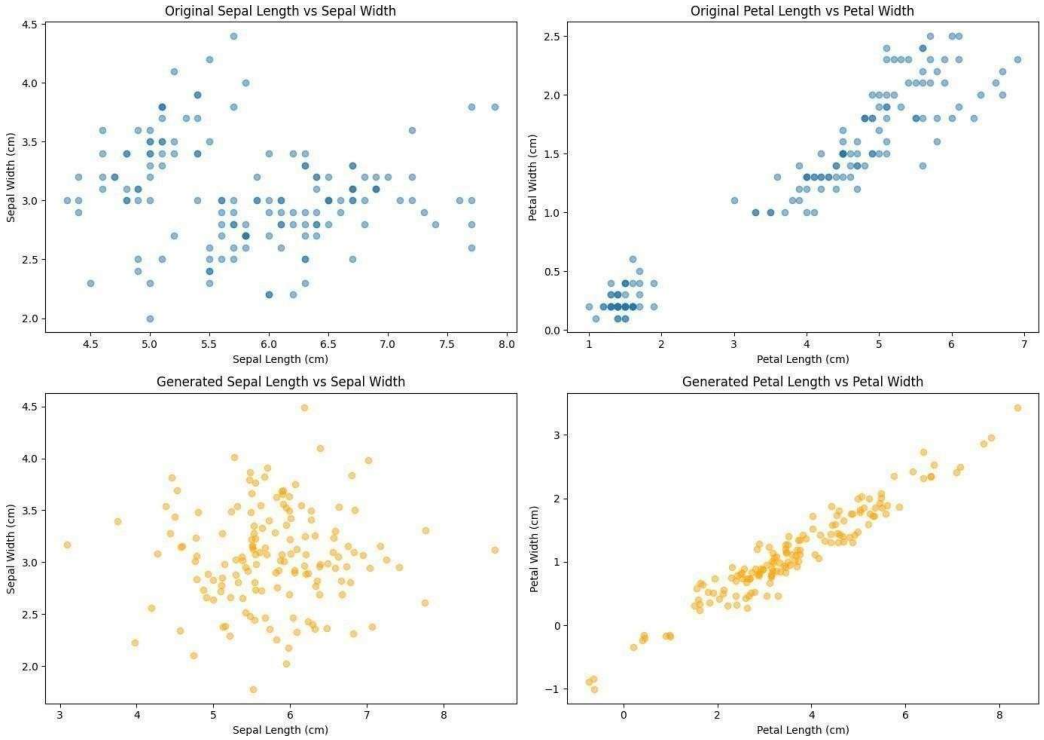
- **Load the Iris Dataset**:
  - The Iris dataset is loaded and converted into a pandas DataFrame for easy manipulation.
- **Calculate Mean and Covariance**:
  - Compute the mean and covariance matrix of the features in the Iris dataset using mean() and cov().
- **Generate Samples**:
  - Use np.random.multivariate_normalto generate samples from a multivariate normal distribution with the calculated mean and covariance matrix.
- **Visualization**:
  - The generated samples are plotted alongside the original data for visual comparison. This helps to understand how well the generated samples resemble the original distribution.

Program:

```
import numpy as np
import pandas as pd
from sklearn.datasets import load_iris
import matplotlib.pyplot as plt

# Load the Iris Dataset
iris = load_iris()
data = pd.DataFrame(data=iris.data, columns=iris.feature_names)

# Display basic statistics
print("Data Mean:\n", data.mean())
print("Data Covariance Matrix:\n", data.cov())

# Extract mean and covariance
mean = data.mean().values
covariance_matrix = data.cov().values
```

```
# Generate samples from a normal distribution with the given mean and covariance
num_samples = 150  # Number of samples to generate
samples = np.random.multivariate_normal(mean, covariance_matrix, num_samples)

# Create a DataFrame for the generated samples generated_data =
pd.DataFrame(samples, columns=data.columns)

# Display the first few rows of the generated data print("Generated
Samples:\n", generated_data.head())

# Plotting the original and generated data for comparison fig, axs =
plt.subplots(2, 2, figsize=(14, 10))

# Original Data Plots
axs[0, 0].scatter(data['sepal length (cm)'], data['sepal width (cm)'], alpha=0.5)
axs[0, 0].set_title('Original Sepal Length vs Sepal Width') axs[0,
0].set_xlabel('Sepal Length (cm)')
axs[0, 0].set_ylabel('Sepal Width (cm)')

axs[0, 1].scatter(data['petal length (cm)'], data['petal width (cm)'], alpha=0.5)
axs[0, 1].set_title('Original Petal Length vs Petal Width') axs[0,
1].set_xlabel('Petal Length (cm)')
axs[0, 1].set_ylabel('Petal Width (cm)')

# Generated Data Plots
axs[1, 0].scatter(generated_data['sepal length (cm)'], generated_data['sepalwidth
(cm)'], alpha=0.5, color='orange') axs[1, 0].set_title('Generated Sepal Length vs Sepal
Width') axs[1, 0].set_xlabel('Sepal Length (cm)')
axs[1, 0].set_ylabel('Sepal Width (cm)')

axs[1, 1].scatter(generated_data['petal length (cm)'], generated_data['petalwidth (cm)'],
alpha=0.5, color='orange') axs[1, 1].set_title('Generated Petal Length vs Petal Width')
axs[1, 1].set_xlabel('Petal Length (cm)')
axs[1, 1].set_ylabel('Petal Width (cm)')

plt.tight_layout()
plt.show()
```

Output:


Original Sepal Length vs Sepal Width
Sepal Width (cm)
Sepal Length (cm)
Original Petal Length vs Petal Width
Petal Width (cm)
Petal Length (cm)
Generated Sepal Length vs Sepal Width
Sepal Width (cm)
Sepal Length (cm)
Generated Petal Length vs Petal Width
Petal Width (cm)
Petal Length (cm)

Practical No: 6

Aim: Implement Linear Perceptron Learning algorithm

Theory:

1. **Load and Prepare the Dataset:**
   - Load the Iris dataset and filter it to include only two classes for binary classification.
   - Standardize the features for better convergence.
2. **Define the Perceptron Model:**
   - Perceptron class:
     - ___init_: Initializes the learning rate and number of iterations.
     - fit: Trains the Perceptron by iterating through epochs and updating weights based on misclassification.
     - predict: Makes predictions based on the learned weights.
     - _net_input: Calculates the net input to the activation function.
3. **Train and Evaluate the Model:**
   - Train the Perceptron using the training data.
   - Evaluate the model using accuracy and a classification report.
   - Plot training errors over epochs to visualize the convergence of the model.

This implementation provides a basic understanding of the Perceptron algorithm and how it can be applied to the Iris dataset. For more complex datasets or multi-class problems, consider using more sophisticated algorithms or extending the Perceptron model.

Program:

```
import numpy as np
import pandas as pd
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import accuracy_score, classification_report
import matplotlib.pyplot as plt

# Load the Iris Dataset
iris = load_iris()
data = pd.DataFrame(data=iris.data, columns=iris.feature_names)
data['target'] = iris.target

# For binary classification: Iris-setosa vs. Iris-versicolor
data = data[data['target'] != 2] # Exclude Iris-virginica
data['target'] = data['target'].map({0: 0, 1: 1}) # Map to binary
target
```

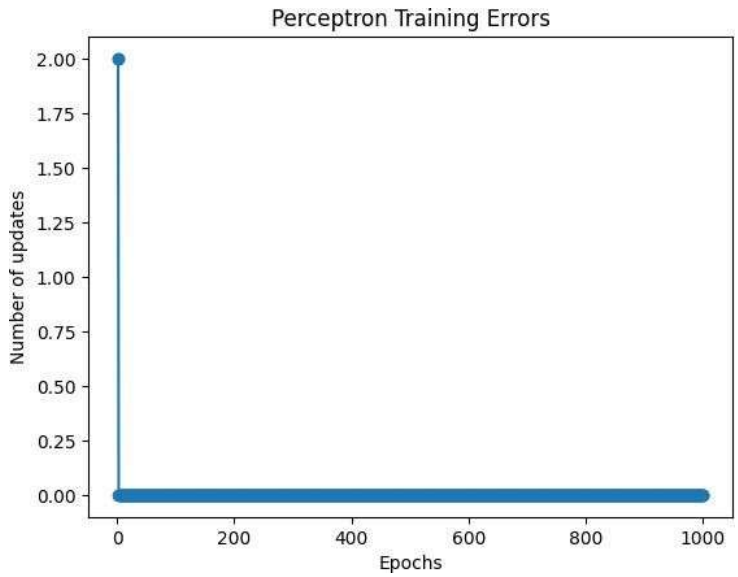
```
# Split Data into Features and Target X =
data.drop('target', axis=1).values y =
data['target'].values

# Split Data into Training and Testing Sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

# Standardize the Data scaler=
StandardScaler()
X_train = scaler.fit_transform(X_train) X_test =
scaler.transform(X_test)

# Define the Perceptron Algorithm class
Perceptron:
    def init (self, learning_rate=0.01, n_iter=1000):
        self.learning_rate = learning_rate self.n_iter = n_iter

    def fit(self, X, y):
        self.weights = np.zeros(X.shape[1] + 1)  # Including bias term self.errors = []

        for _inrange(self.n_iter): errors = 0
            for xi, target in zip(X, y):
                update = self.learning_rate * (target - self.predict(xi))
                self.weights[1:] += update * xi
                self.weights[0] += update errors +=
                int(update != 0.0)
            self.errors.append(errors)

    def predict(self, X):
        return np.where(self._net_input(X) >= 0.0, 1, 0)

    def _net_input(self, X):
        returnnp.dot(X, self.weights[1:]) + self.weights[0]

# Train the Perceptron Model
perceptron=Perceptron(learning_rate=0.01, n_iter=1000) perceptron.fit(X_train, y_train)

# Make Predictions
y_pred= perceptron.predict(X_test)
```

```
# Evaluate the Model
accuracy = accuracy_score(y_test, y_pred)
report = classification_report(y_test, y_pred)

print(f"Accuracy: {accuracy:.4f}")
print(f"Classification Report:\n{report}")

# Plot the Training Errors
plt.plot(range(1, len(perceptron.errors) + 1), perceptron.errors,
marker='o')
plt.xlabel('Epochs')
plt.ylabel('Number of updates')
plt.title('Perceptron Training Errors')
plt.show()
```

Output:

Accuracy: 1.0000 Classification Report:

	precision	recall	f1-score	support
0	1.00	1.00	1.00	17
1	1.00	1.00	1.00	13
accuracy			1.00	30
macro avg	1.00	1.00	1.00	30
weighted avg	1.00	1.00	1.00	30



Practical No: 7

Aim: Build IRIS flower classification in python using pattern recognition models

Theory:

We'll use the following pattern recognition models:

1. **Logistic Regression**
2. **k-Nearest Neighbors (k-NN)**
3. **Support Vector Machine (SVM)**
4. **Decision Tree**
5. **Random Forest**

Here's a step-by-step guide to implement and evaluate these models using the Iris dataset in Python.

Step-by-Step Implementation

1. **Load and Prepare the Dataset**:
   - Load the Iris dataset.
   - Preprocess the data (e.g., standardization).
   - Split the data into training and testing sets.
2. **Define and Train Models**:
   - Implement each model and fit it to the training data.
3. **Evaluate the Models**:
   - Assess the performance of each model using accuracy and other metrics.
4. **Compare Results**:
   - Compare the classification accuracy and other metrics of each model.

Program:

```
import numpy as np
import pandas as pd
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LogisticRegression
from sklearn.neighbors import KNeighborsClassifier
from sklearn.svm import SVC
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score, classification_report
import matplotlib.pyplot as plt

# Load the Iris Dataset
iris = load_iris()
```

```
X = iris.data y=
iris.target

# Split Data into Training and Testing Sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

# Standardize the Data scaler=
StandardScaler()
X_train = scaler.fit_transform(X_train) X_test =
scaler.transform(X_test)

# Definethe Models models = {
    "Logistic Regression": LogisticRegression(max_iter=200), "k-Nearest
    Neighbors": KNeighborsClassifier(),
    "Support Vector Machine": SVC(), "Decision Tree":
    DecisionTreeClassifier(), "Random Forest":
    RandomForestClassifier()
}

# Train and Evaluate Models results = {}
formodel_name, model inmodels.items(): model.fit(X_train, y_train)
    y_pred= model.predict(X_test)
    accuracy = accuracy_score(y_test, y_pred) report =
    classification_report(y_test, y_pred,
target_names=iris.target_names)
    results[model_name] = {
        "accuracy":accuracy,
        "report": report
    }

# Print Results
for model_name, metrics in results.items(): print(f"Model:
    {model_name}") print(f"Accuracy:
    {metrics['accuracy']:.4f}")
    print(f"Classification Report:\n{metrics['report']}") print("-" * 60)

# Plotting the results model_names =
list(results.keys())
accuracies = [results[model_name]['accuracy'] for model_name in model_names]
```

```
plt.figure(figsize=(10, 6))
plt.bar(model_names, accuracies, color='skyblue')
plt.xlabel('Model')
plt.ylabel('Accuracy')
plt.title('Model Comparison')
plt.xticks(rotation=45)
plt.ylim(0, 1)
plt.show()
```

Output:

Model: Logistic Regression Accuracy:
1.0000
Classification Report:

	precision	recall	f1-score	support
setosa	1.00	1.00	1.00	19
versicolor	1.00	1.00	1.00	13
virginica	1.00	1.00	1.00	13
accuracy			1.00	45
macro avg	1.00	1.00	1.00	45
weighted avg	1.00	1.00	1.00	45

Model: k-Nearest Neighbors Accuracy:
1.0000 Classification Report:

	precision	recall	f1-score	support
setosa	1.00	1.00	1.00	19
versicolor	1.00	1.00	1.00	13
virginica	1.00	1.00	1.00	13
accuracy			1.00	45
macroavg	1.00	1.00	1.00	45
weighted avg	1.00	1.00	1.00	45

Model: Support Vector Machine
Accuracy: 1.0000 Classification
Report:

	precision	recall	f1-score	support
setosa	1.00	1.00	1.00	19
versicolor	1.00	1.00	1.00	13
virginica	1.00	1.00	1.00	13
accuracy			1.00	45
macro avg	1.00	1.00	1.00	45
weighted avg	1.00	1.00	1.00	45

Model: Decision Tree Accuracy:
1.0000 Classification Report:

	precision	recall	f1-score	support
setosa	1.00	1.00	1.00	19
versicolor	1.00	1.00	1.00	13
virginica	1.00	1.00	1.00	13
accuracy			1.00	45
macro avg	1.00	1.00	1.00	45
weighted avg	1.00	1.00	1.00	45

Model: Random Forest