# EXPERIMENT 1

**AIM:** Write a program for performing batch gradient descent and stochastic gradient descent.

**THEORY:**

## BATCH GRADIENT DESCENT:

**CODE:**

import numpy as np # type: ignore # Example data

# Features: x1, x2 and Target: y data = np.array([

[1, 2, 3], # x1, x2, y

[2, 4, 5],

[3, 6, 7],

[4, 8, 9]

])

# Split features (x) and target (y) X = data[:, :-1] # Features (x1, x2)

y = data[:, -1] # Target (y)

# Add a bias term (column of ones for Q0) to the features

X = np.c\_[np.ones(X.shape[0]), X] # Adding bias (or Q0) as the first column

# Hyperparameters learning\_rate = 0.02

epochs = 10000 # Number of iterations threshold = 0.000002

# Initialize weights (Q0, Q1, Q2) to zeros weights = np.zeros(X.shape[1])

# Batch Gradient Descent

m = len(y) # Number of data points for epoch in range(epochs):

# Compute predictions y\_hat = np.dot(X, weights)

# Compute error error = y\_hat - y

# Compute gradients

gradients = (1 / m) \* np.dot(X.T, error)

# Update weights

weights = weights - learning\_rate \* gradients cost = (1 / (2 \* m)) \* np.sum(error \*\* 2)

# (Optional) Print progress

if epoch % 100 == 0 or epoch == epochs - 1:

print(f"Epoch {epoch + 1}/{epochs}, Cost: {cost:.4f}, Weights: {weights}" if cost<=threshold:

print(f"Epoch {epoch + 1}/{epochs}, Cost: {cost:.4f}, Weights: {weights}") break

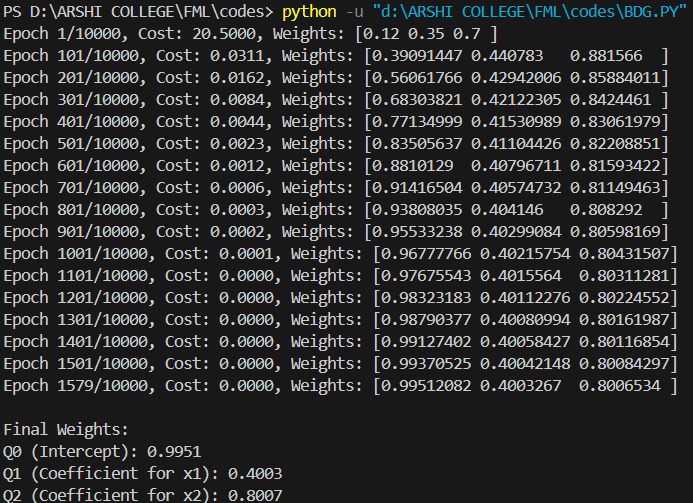
# Final weights

print("\nFinal Weights:")

print(f"Q0 (Intercept): {weights[0]:.4f}")

print(f"Q1 (Coefficient for x1): {weights[1]:.4f}") print(f"Q2 (Coefficient for x2): {weights[2]:.4f}")

**CODE OUTPUT:**

****

## STOCHASTIC GRADIENT DESCENT:

**CODE:**

import numpy as np

# Example data

# Features: x1, x2 and Target: y data = np.array([

[1, 2, 3], # x1, x2, y

[2, 4, 5],

[3, 6, 7],

[4, 8, 9]

])

# Split features (x) and target (y) X = data[:, :-1] # Features (x1, x2)

y = data[:, -1] # Target (y)

# Add a bias term (column of ones for b0) to the features

X = np.c\_[np.ones(X.shape[0]), X] # Adding bias (or Q0) as the first column

# Hyperparameters learning\_rate = 0.01

epochs = 100 # Number of passes through the dataset threshold = 0.02

# Initialize weights (Q0, Q1, Q2) to zeros weights = np.zeros(X.shape[1])

# Stochastic Gradient Descent for epoch in range(epochs):

for i in range(len(X)):

# Predict y\_hat

y\_hat = np.dot(X[i], weights) # print(X[i])

# Compute error error = y\_hat - y[i]

# Update weights

weights = weights - learning\_rate \* error \* X[i]

cost = (1 / (2 \* len(X))) \* np.sum((np.dot(X, weights) - y) \*\* 2) # (Optional) Print progress

if epoch % 10 == 0 or epoch == epochs - 1:

# cost = (1 / (2 \* len(X))) \* np.sum((np.dot(X, weights) - y) \*\* 2)

print(f"Epoch {epoch + 1}/{epochs}, Cost: {cost:.4f}, Weights: {weights}")

if cost<=threshold:

print(f"Epoch {epoch + 1}/{epochs}, Cost: {cost:.4f}, Weights: {weights}") break

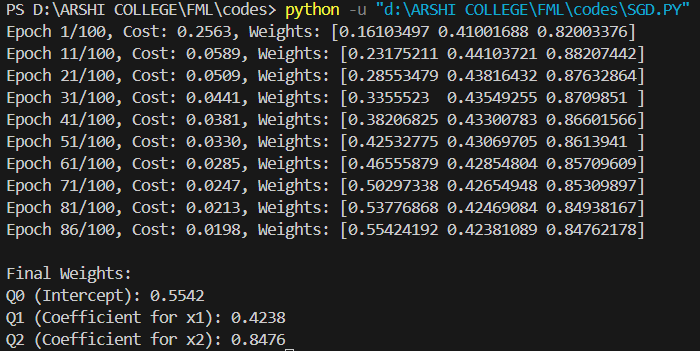
# Final weights

print("\nFinal Weights:")

print(f"Q0 (Intercept): {weights[0]:.4f}")

print(f"Q1 (Coefficient for x1): {weights[1]:.4f}") print(f"Q2 (Coefficient for x2): {weights[2]:.4f}")

**CODE OUTPUT:**

****

**LEARNING OUTCOME:**

# EXPERIMENT 2

**AIM:** Write a program for performing simple linear regression using and without using libraries.

**THEORY:**

## SIMPLE LINEAR REGRESSION (WITHOUT LIBRARIES):

**CODE:**

import numpy as np

import matplotlib.pyplot as plt import pandas as pd

def load\_dataset():

dataset = pd.read\_csv("Salary\_Data[1].csv") X = dataset.iloc[:, :-1].values

Y = dataset.iloc[:, -1].values.reshape(-1, 1)

# Handling missing data

mean\_X = np.nanmean(X) # Compute mean of non-null values X[np.isnan(X)] = mean\_X # Replace NaN values with mean

# Standardizing the data (optional but recommended) X = (X - np.mean(X)) / np.std(X)

Y = (Y - np.mean(Y)) / np.std(Y)

# Splitting data into training (67%) and test (33%) sets test\_size = int(len(X) \* 0.33)

indices = np.random.permutation(len(X))

X\_train, Y\_train = X[indices[:-test\_size]], Y[indices[:-test\_size]] X\_test, Y\_test = X[indices[-test\_size:]], Y[indices[-test\_size:]]

return X\_train, Y\_train, X\_test, Y\_test

def compute\_coefficients(x, y):

# Computing slope (m) and intercept (b) using Least Squares Method num = np.sum((x - np.mean(x)) \* (y - np.mean(y)))

den = np.sum((x - np.mean(x))\*\*2) m = num / den

b = np.mean(y) - m \* np.mean(x) return b, m

def plot\_regression\_line(x, y, coeff):

plt.scatter(x, y, color='red', marker='o', label="Data points") y\_pred = coeff[0] + coeff[1] \* x # Regression line

plt.plot(x, y\_pred, color='green', label="Regression line") plt.title("Simple Linear Regression")

plt.xlabel("Experience") plt.ylabel("Salary")

plt.legend() plt.show()

# Calculate Mean Squared Error mse = np.mean((y - y\_pred) \*\* 2)

print(f'Mean Squared Error = {mse:.4f}')

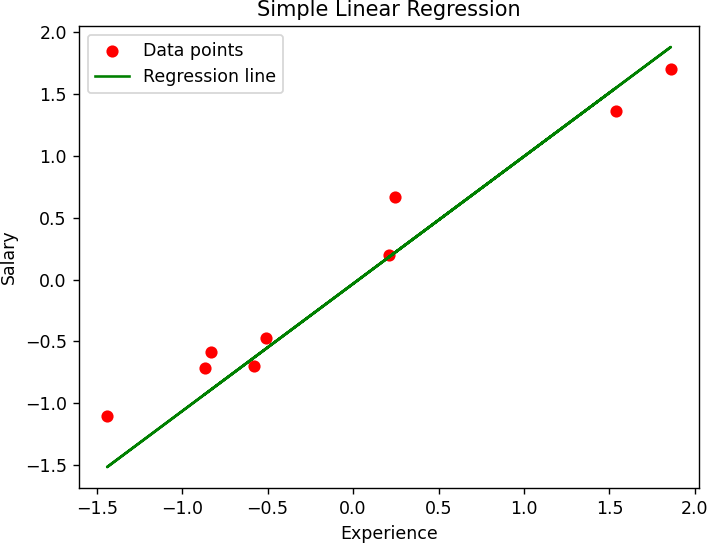
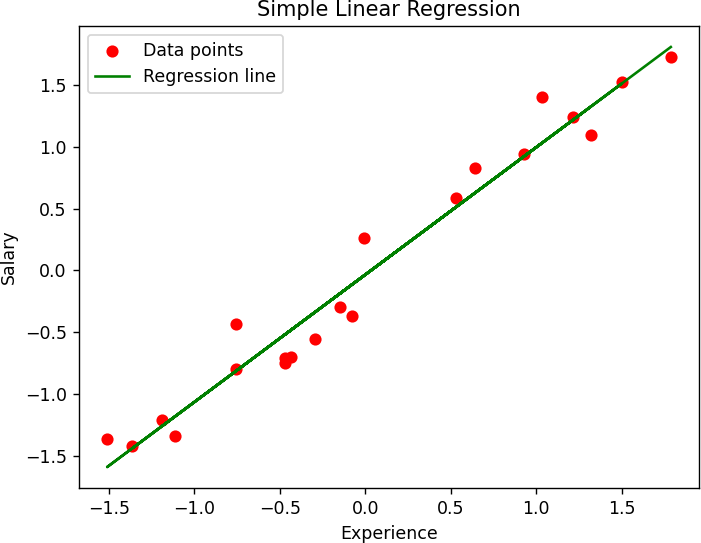
def main():

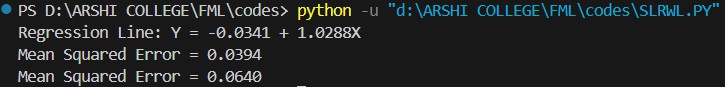
X\_train, Y\_train, X\_test, Y\_test = load\_dataset() coeff = compute\_coefficients(X\_train, Y\_train)

print(f"Regression Line: Y = {coeff[0]:.4f} + {coeff[1]:.4f}X") plot\_regression\_line(X\_train, Y\_train, coeff) # Train set plot\_regression\_line(X\_test, Y\_test, coeff) # Test set

if name == " main ": main()

**CODE OUTPUT:**

****



## SIMPLE LINEAR REGRESSION (WITH LIBRARIES):

**CODE:**

# Importing Libraries import numpy as np

import matplotlib.pyplot as plt

def mean\_squared\_error(y\_true, y\_predicted):

# Calculating the loss or cost

cost = np.sum((y\_true-y\_predicted)\*\*2) / len(y\_true) return cost

# Gradient Descent Function

# Here iterations, learning\_rate, stopping\_threshold # are hyperparameters that can be tuned

def gradient\_descent(x, y, iterations = 1000, learning\_rate = 0.0001, stopping\_threshold = 1e-6):

# Initializing weight, bias, learning rate and iterations current\_weight = 0.1

current\_bias = 0.01 iterations = iterations learning\_rate = learning\_rate n = float(len(x))

costs = [] weights = []

previous\_cost = None

# Estimation of optimal parameters

for i in range(iterations):

# Making predictions

y\_predicted = (current\_weight \* x) + current\_bias

# Calculating the current cost

current\_cost = mean\_squared\_error(y, y\_predicted)

# If the change in cost is less than or equal to

# stopping\_threshold we stop the gradient descent

if previous\_cost and abs(previous\_cost-current\_cost)<=stopping\_threshold: break

previous\_cost = current\_cost

costs.append(current\_cost) weights.append(current\_weight)

# Calculating the gradients

weight\_derivative = -(2/n) \* sum(x \* (y-y\_predicted)) bias\_derivative = -(2/n) \* sum(y-y\_predicted)

# Updating weights and bias

current\_weight = current\_weight - (learning\_rate \* weight\_derivative) current\_bias = current\_bias - (learning\_rate \* bias\_derivative)

# Printing the parameters for each 1000th iteration print(f"Iteration {i+1}: Cost {current\_cost}, Weight \

{current\_weight}, Bias {current\_bias}")

# Visualizing the weights and cost at for all iterations plt.figure(figsize = (8,6))

plt.plot(weights, costs)

plt.scatter(weights, costs, marker='o', color='red')

plt.title("Cost vs Weights") plt.ylabel("Cost") plt.xlabel("Weight") plt.show()

return current\_weight, current\_bias

def main():

# Data

X = np.array([32.50234527, 53.42680403, 61.53035803, 47.47563963, 59.81320787,

55.14218841, 52.21179669, 39.29956669, 48.10504169, 52.55001444,

45.41973014, 54.35163488, 44.1640495 , 58.16847072, 56.72720806,

48.95588857, 44.68719623, 60.29732685, 45.61864377, 38.81681754])

Y = np.array([31.70700585, 68.77759598, 62.5623823 , 71.54663223, 87.23092513,

78.21151827, 79.64197305, 59.17148932, 75.3312423 , 71.30087989,

55.16567715, 82.47884676, 62.00892325, 75.39287043, 81.43619216,

60.72360244, 82.89250373, 97.37989686, 48.84715332, 56.87721319])

# Estimating weight and bias using gradient descent

estimated\_weight, estimated\_bias = gradient\_descent(X, Y, iterations=2000)

print(f"Estimated Weight: {estimated\_weight}\nEstimated Bias: {estimated\_bias}")

# Making predictions using estimated parameters Y\_pred = estimated\_weight\*X + estimated\_bias

# Plotting the regression line plt.figure(figsize = (8,6))

plt.scatter(X, Y, marker='o', color='red')

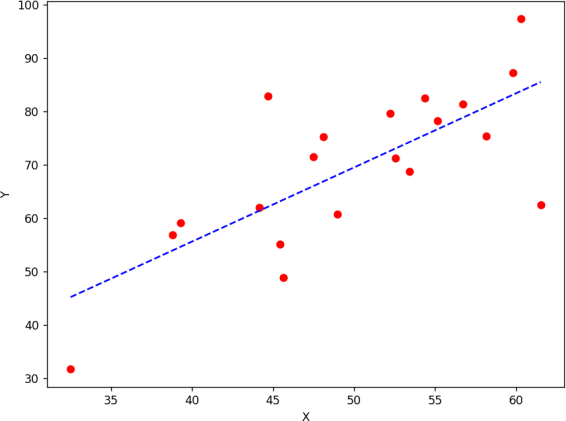
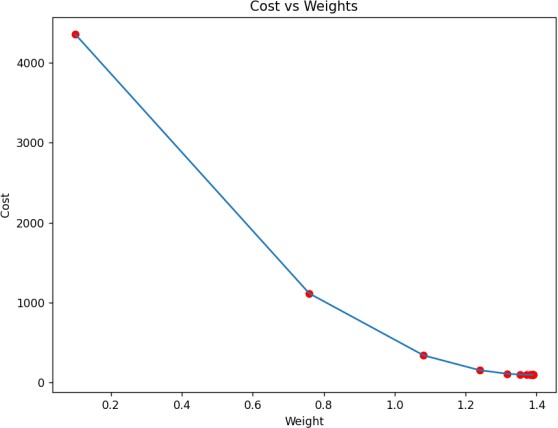
plt.plot([min(X), max(X)], [min(Y\_pred), max(Y\_pred)], color='blue',markerfacecolor='red', markersize=10,linestyle='dashed')

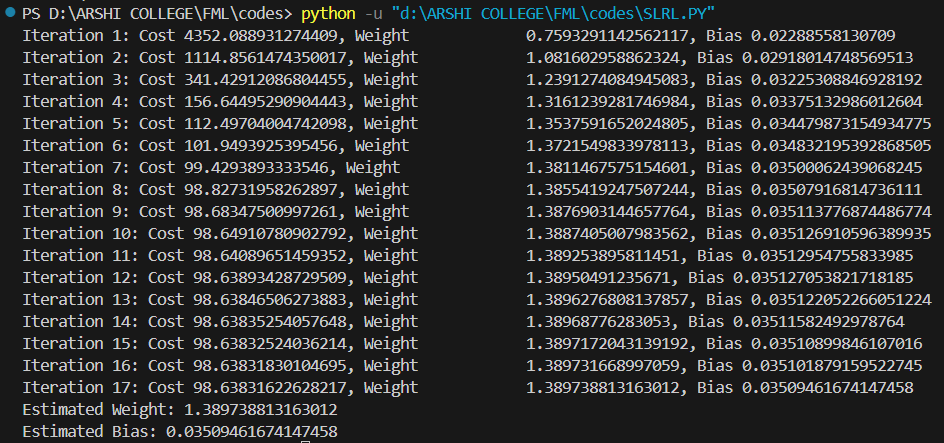
plt.xlabel("X")

plt.ylabel("Y") plt.show()

if name ==" main ": main()

**CODE OUTPUT:**

****



**LEARNING OUTCOME:**

# EXPERIMENT 3

**AIM:** Write a program for performing multiple linear regression using and without using libraries.

**THEORY:**

## MULTIPLE LINEAR REGRESSION (WITH LIBRARIES):

**CODE:**

import numpy as np

import matplotlib.pyplot as plt import pandas as pd

import seaborn as sns

df = pd.read\_csv('AMZN.csv') df.shape

df.head()

df.drop('Date',axis=1,inplace=True) df.drop('Company',axis=1,inplace=True) df.head()

df.corr()

sns.heatmap(df.corr(),annot=True,cmap='YlOrRd',linewidth=0.5,linecolor='yellow') from sklearn.model\_selection import train\_test\_split

X = df.drop('Adj Close',axis=1) Y = df['Adj Close']

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, Y, test\_size = 0.2, random\_state = 0) from sklearn import linear\_model, metrics

regressor = linear\_model.LinearRegression() regressor.fit(X\_train, y\_train)

y\_pred = regressor.predict(X\_test) print("Coefficients: ",regressor.coef\_)

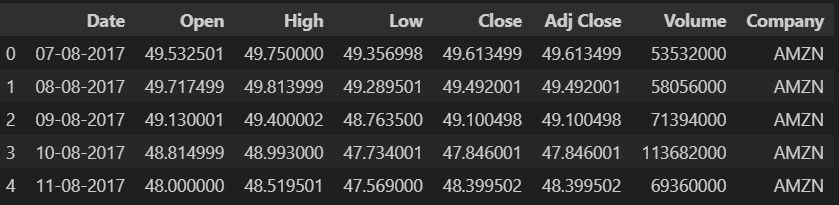
print("Variance score: {}".format(regressor.score(X\_test,y\_test))) np.set\_printoptions(precision=2)

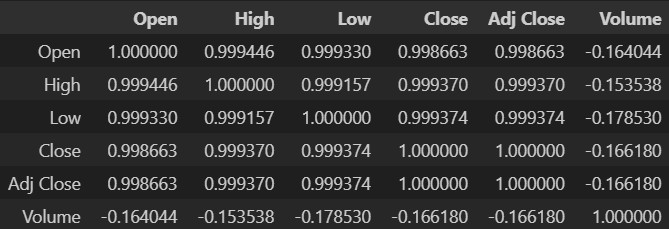
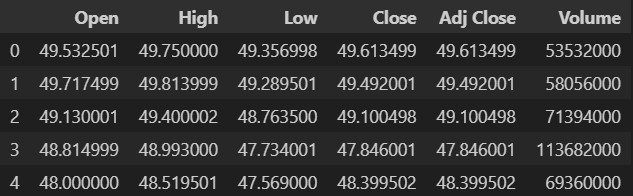
print(np.concatenate((y\_pred.reshape(len(y\_pred),1), y\_test.reshape(len(y\_test),1)),1)) plt.hist((y\_pred-y\_test)\*100/y\_test)

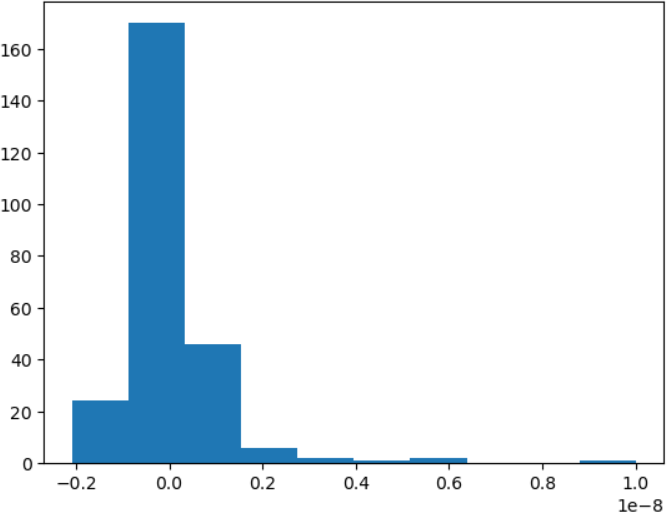
from sklearn import metrics

metrics.mean\_squared\_error(y\_test,y\_pred)

**CODE OUTPUT:**



****

****

## MULTIPLE LINEAR REGRESSION (WITHOUT LIBRARIES):

**CODE:**

import numpy as np

import matplotlib.pyplot as plt import seaborn as sns

import pandas as pd

def mean\_squared\_error(y\_true, y\_pred): return np.mean((y\_true - y\_pred) \*\* 2)

class CustomLinearRegression:

def init (self, learning\_rate=0.01, epochs=1000): self.learning\_rate = learning\_rate

self.epochs = epochs

def fit(self, X, y):

self.m, self.n = X.shape self.weights = np.zeros(self.n) self.bias = 0

# Gradient Descent

for \_ in range(self.epochs):

y\_pred = np.dot(X, self.weights) + self.bias error = y\_pred - y

# Compute Gradients

dw = (2 / self.m) \* np.dot(X.T, error) db = (2 / self.m) \* np.sum(error)

# Update Parameters

self.weights -= self.learning\_rate \* dw self.bias -= self.learning\_rate \* db

def predict(self, X):

return np.dot(X, self.weights) + self.bias

df = pd.read\_csv("https://raw.githubusercontent.com/mwaskom/seaborn-data/master/iris.csv")

# Select features & target variable

X = df[['sepal\_width', 'petal\_length', 'petal\_width']].values # Using 3 predictors y = df['sepal\_length'].values # Predicting 'sepal\_length'

# Normalize Features (Min-Max Scaling)

X = (X - X.min(axis=0)) / (X.max(axis=0) - X.min(axis=0))

# Split data into training and testing (80%-20%) split\_ratio = 0.8

split\_index = int(len(X) \* split\_ratio)

X\_train, X\_test = X[:split\_index], X[split\_index:] y\_train, y\_test = y[:split\_index], y[split\_index:]

mlr\_custom = CustomLinearRegression(learning\_rate=0.1, epochs=1000) mlr\_custom.fit(X\_train, y\_train)

y\_pred\_custom = mlr\_custom.predict(X\_test)

# Evaluate Model

mse = mean\_squared\_error(y\_test, y\_pred\_custom) print(f"Custom Multi Linear Regression MSE: {mse:.4f}")

plt.figure(figsize=(6, 5))

plt.scatter(y\_test, y\_pred\_custom, color='blue', edgecolors='k', alpha=0.7)

plt.plot([min(y\_test), max(y\_test)], [min(y\_test), max(y\_test)], color='red', linestyle='dashed') plt.xlabel("Actual Values")

plt.ylabel("Predicted Values")

plt.title("Actual vs Predicted Values (Custom Multi Linear Regression)") plt.show()

residuals = y\_test - y\_pred\_custom plt.figure(figsize=(6, 5))

plt.scatter(y\_pred\_custom, residuals, color='purple', edgecolors='k', alpha=0.7) plt.axhline(y=0, color='red', linestyle='dashed')

plt.xlabel("Predicted Values") plt.ylabel("Residuals")

plt.title("Residual Plot (Custom Multi Linear Regression)") plt.show()

feature\_names = ['sepal\_width', 'petal\_length', 'petal\_width'] coefficients = mlr\_custom.weights

plt.figure(figsize=(6, 5))

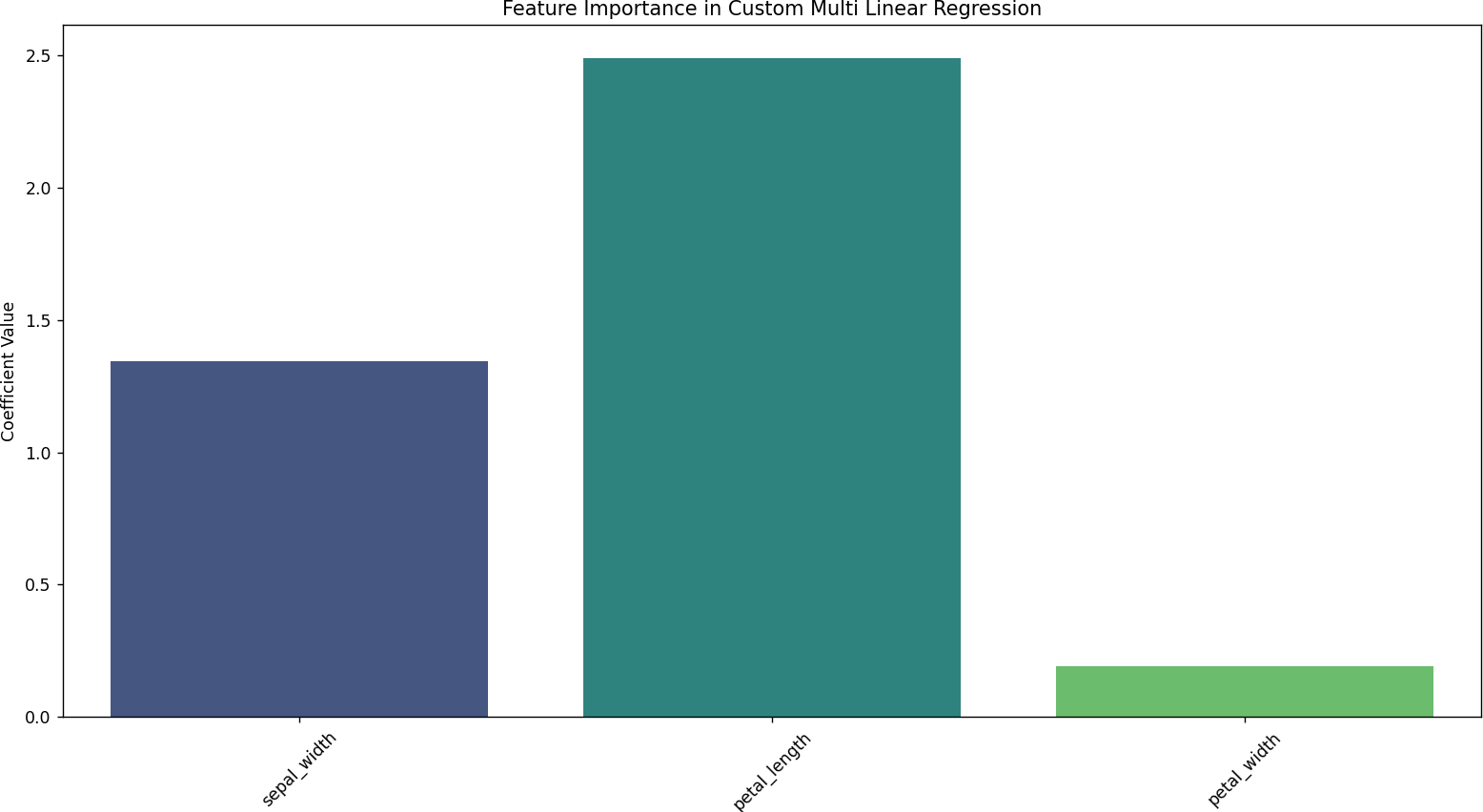
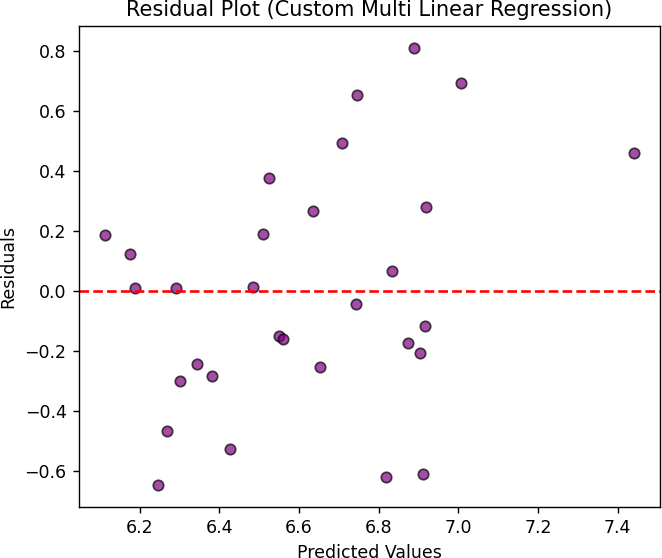
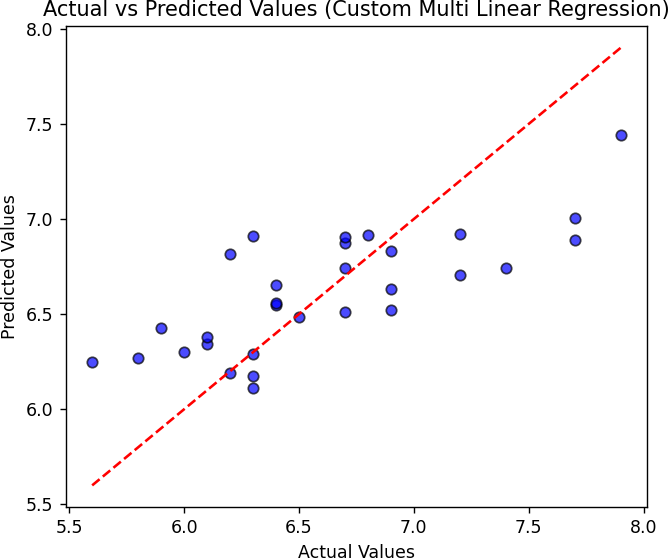
sns.barplot(x=feature\_names, y=coefficients, palette="viridis") plt.xlabel("Features")

plt.ylabel("Coefficient Value")

plt.title("Feature Importance in Custom Multi Linear Regression") plt.xticks(rotation=45)

plt.show()

**CODE OUTPUT:**

****

**LEARNING OUTCOME:**

# EXPERIMENT 4

**AIM:** Write a program for performing logistic regression using and without using libraries.

**THEORY:**

## LOGISTIC REGRESSION (WITH LIBRARIES):

**CODE:**

import pandas as pd

import matplotlib.pyplot as plt import numpy as np

import seaborn as sns

df = pd.read\_csv("apple\_quality.csv") df.head()

df.describe() df.info()

df.dropna(inplace=True) df.info()

type(df['Acidity'])

#‘coerce’ will set invalid parsing as NaN

df['Acidity']=pd.to\_numeric(df['Acidity'], errors='coerce') df.info()

#df.isna().any(axis=1) df.dropna(inplace=True)

df1 = df[df.isna().any(axis=1)] df2 = df.drop('A\_id',axis=1) df2.describe()

sns.heatmap(df2.drop('Quality',axis=1).corr(),cmap='YlOrRd',annot=True,linewidth=0.5,linecolor='Yel low')

sns.pairplot(df2)

sns.pairplot(df2,hue='Quality')

df.Quality = df2.Quality.replace({'good':1,'bad':0}) df2.head()

from sklearn.model\_selection import train\_test\_split from sklearn.linear\_model import LogisticRegression from sklearn import metrics

X = df2.drop('Quality', axis=1)

Y = df2['Quality']

X\_train, X\_test, Y\_train, Y\_test = train\_test\_split(X, Y, test\_size=0.25, random\_state=0) reg = LogisticRegression()

reg.fit(X\_train, Y\_train) Y\_pred = reg.predict(X\_test)

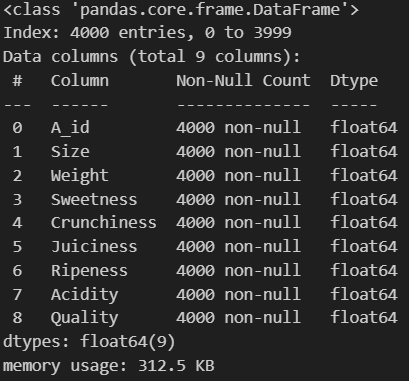
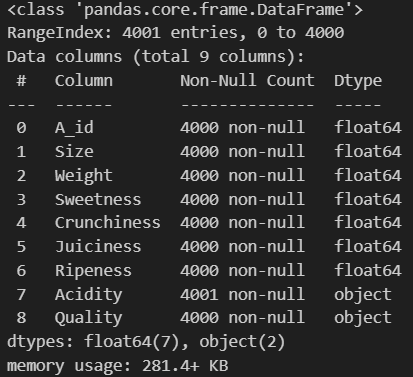
confusion\_matrix = metrics.confusion\_matrix(Y\_test, Y\_pred) print(confusion\_matrix)

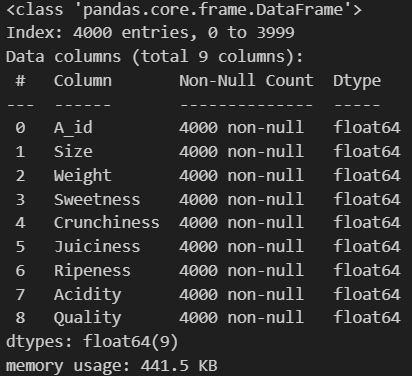
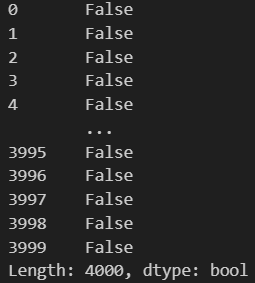
sns.heatmap(confusion\_matrix, cmap='YlGnBu', annot=True, fmt='g') metrics.accuracy\_score(Y\_pred, Y\_test)

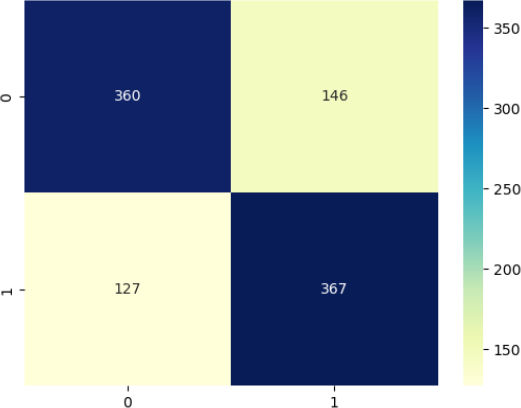
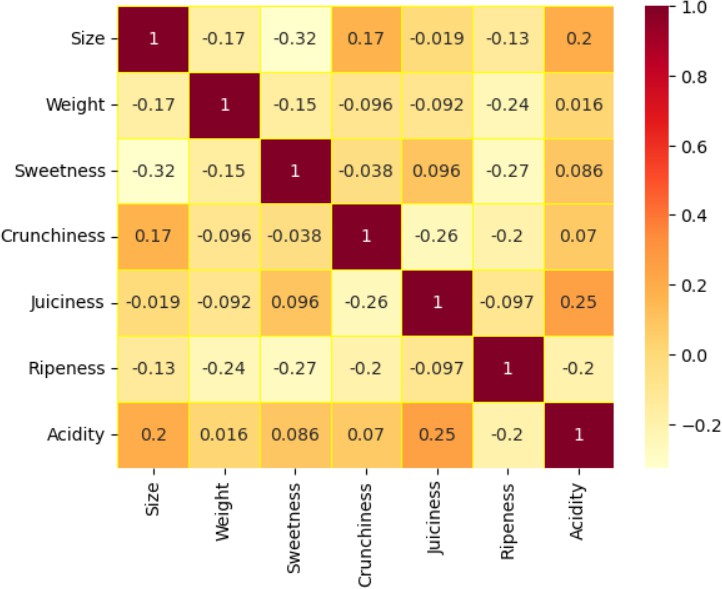
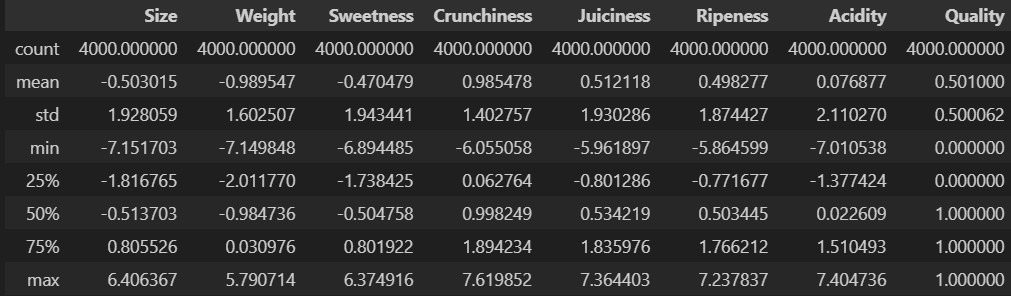
metrics.precision\_score(Y\_test, Y\_pred) metrics.f1\_score(Y\_test, Y\_pred)

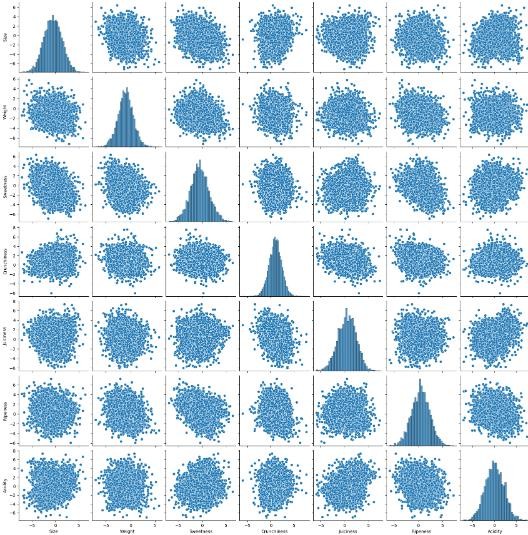
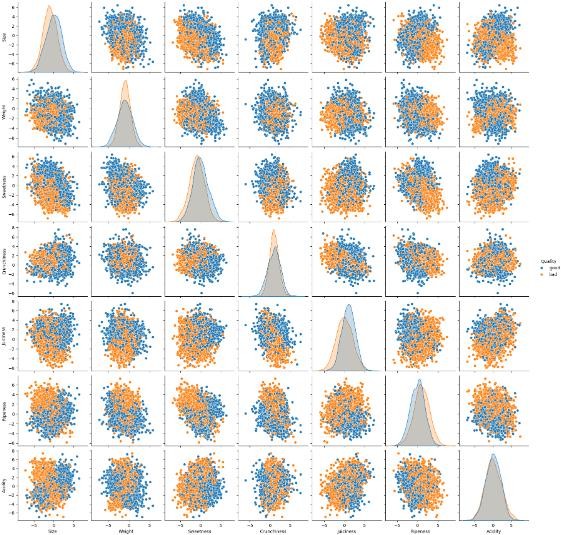
metrics.recall\_score(Y\_test, Y\_pred)

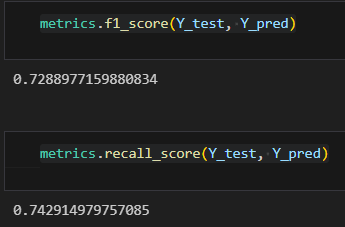
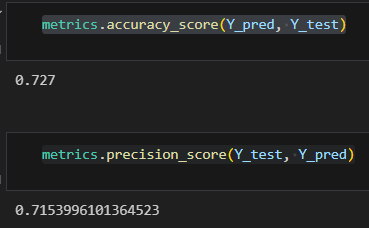
**CODE OUTPUT:**

****

****

****

## LOGISTIC REGRESSION (WITHOUT LIBRARIES):

**CODE:**

import numpy as np

import matplotlib.pyplot as plt import seaborn as sns

import pandas as pd

def sigmoid(z):

return 1 / (1 + np.exp(-z))

class CustomLogisticRegression:

def init (self, learning\_rate=0.01, epochs=1000): self.learning\_rate = learning\_rate

self.epochs = epochs

def fit(self, X, y):

self.m, self.n = X.shape self.weights = np.zeros(self.n) self.bias = 0

# Gradient Descent

for \_ in range(self.epochs):

model = np.dot(X, self.weights) + self.bias predictions = sigmoid(model)

# Compute Gradients

dw = (1 / self.m) \* np.dot(X.T, (predictions - y)) db = (1 / self.m) \* np.sum(predictions - y)

# Update Parameters

self.weights -= self.learning\_rate \* dw self.bias -= self.learning\_rate \* db

def predict(self, X):

model = np.dot(X, self.weights) + self.bias predictions = sigmoid(model)

return [1 if p > 0.5 else 0 for p in predictions]

df = pd.read\_csv("https://raw.githubusercontent.com/mwaskom/seaborn-data/master/iris.csv")

# Convert categorical target ('species') into numeric

df = df[df['species'] != 'virginica'] # Binary classification (Setosa vs. Versicolor)

df['species'] = df['species'].astype('category').cat.codes # 0: Setosa, 1: Versicolor

# Select features (using first 2 for visualization)

X = df[['sepal\_length', 'sepal\_width']].values y = df['species'].values

# Split data into training and testing (80%-20%)

split\_ratio = 0.8

split\_index = int(len(X) \* split\_ratio)

X\_train, X\_test = X[:split\_index], X[split\_index:] y\_train, y\_test = y[:split\_index], y[split\_index:]

log\_reg\_custom = CustomLogisticRegression(learning\_rate=0.1, epochs=1000) log\_reg\_custom.fit(X\_train, y\_train)

y\_pred\_custom = log\_reg\_custom.predict(X\_test)

# Calculate accuracy

accuracy = np.mean(y\_pred\_custom == y\_test)

print("Custom Logistic Regression Accuracy:", accuracy)

def plot\_decision\_boundary(model, X, y):

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

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

xx, yy = np.meshgrid(np.linspace(x\_min, x\_max, 100), np.linspace(y\_min, y\_max, 100))

# Predict on grid points

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

plt.contourf(xx, yy, Z, alpha=0.3, cmap='coolwarm')

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

plt.ylabel("Feature 2")

plt.title("Logistic Regression Decision Boundary (Custom Implementation)") plt.legend(\*scatter.legend\_elements(), title="Classes")

plt.show()

plot\_decision\_boundary(log\_reg\_custom, X\_train, y\_train)

def plot\_confusion\_matrix(y\_true, y\_pred): cm = np.zeros((2, 2), dtype=int)

for true, pred in zip(y\_true, y\_pred):

cm[true][pred] += 1

plt.figure(figsize=(6, 5))

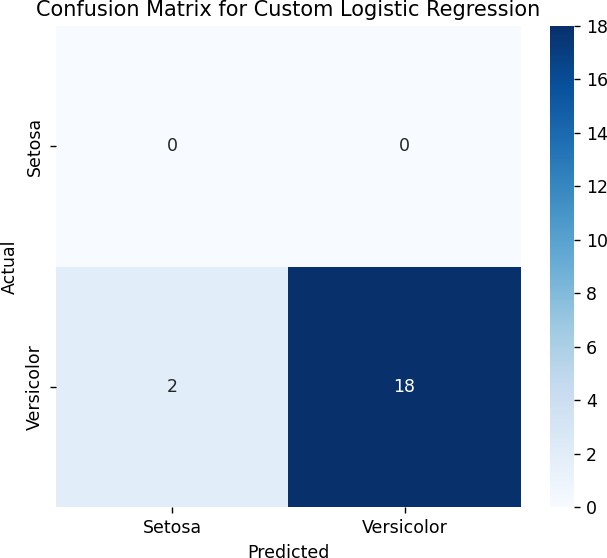
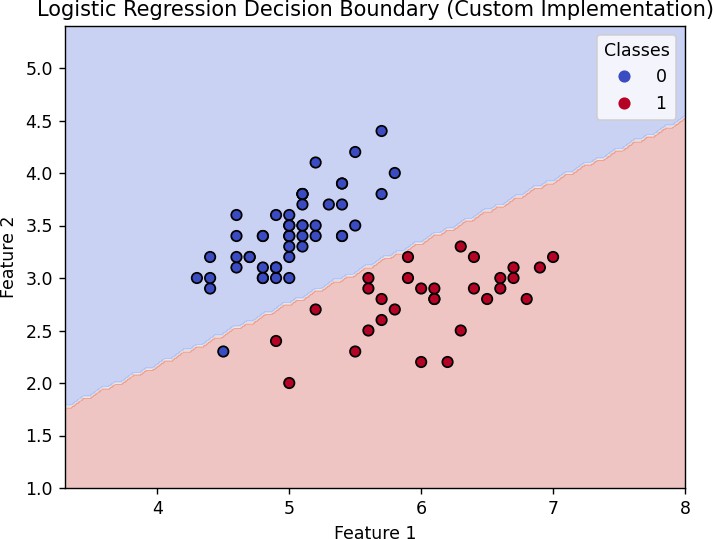
sns.heatmap(cm, annot=True, fmt="d", cmap="Blues", xticklabels=['Setosa', 'Versicolor'], yticklabels=['Setosa', 'Versicolor'])

plt.xlabel('Predicted') plt.ylabel('Actual')

plt.title('Confusion Matrix for Custom Logistic Regression') plt.show()

plot\_confusion\_matrix(y\_test, y\_pred\_custom)

**CODE OUTPUT:**

****

**LEARNING OUTCOME:**

# EXPERIMENT 5

**AIM:** Write a program for performing KNN using and without using libraries.

**THEORY:**

## KNN (WITH LIBRARIES):

**CODE:**

import numpy as np import pandas as pd

import matplotlib.pyplot as plt import seaborn as sns

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

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

df = pd.read\_csv("https://raw.githubusercontent.com/mwaskom/seaborn-data/master/iris.csv")

# Convert categorical target ('species') into numeric

df['species'] = df['species'].astype('category').cat.codes # 0: setosa, 1: versicolor, 2: virginica

# Select features (using first 2 for visualization)

X = df[['sepal\_length', 'sepal\_width']].values y = df['species'].values

# Split data into training and testing (80%-20%)

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

knn = KNeighborsClassifier(n\_neighbors=5) knn.fit(X\_train, y\_train)

y\_pred\_knn = knn.predict(X\_test)

# Evaluate

accuracy = accuracy\_score(y\_test, y\_pred\_knn) print("KNN (sklearn) Accuracy:", accuracy)

def plot\_decision\_boundary(model, X, y):

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

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

xx, yy = np.meshgrid(np.linspace(x\_min, x\_max, 100), np.linspace(y\_min, y\_max, 100))

# Predict on grid points

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

plt.contourf(xx, yy, Z, alpha=0.3)

scatter = plt.scatter(X[:, 0], X[:, 1], c=y, edgecolor='k') plt.xlabel("Feature 1")

plt.ylabel("Feature 2")

plt.title("KNN Decision Boundary (sklearn)")

plt.legend(\*scatter.legend\_elements(), title="Classes") plt.show()

plot\_decision\_boundary(knn, X\_train, y\_train)

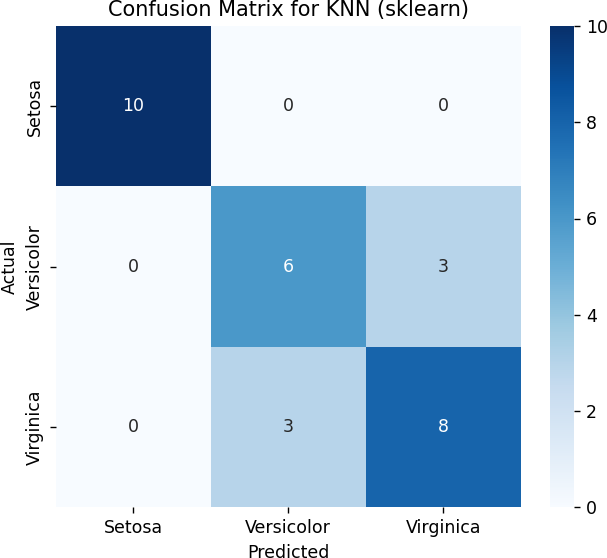
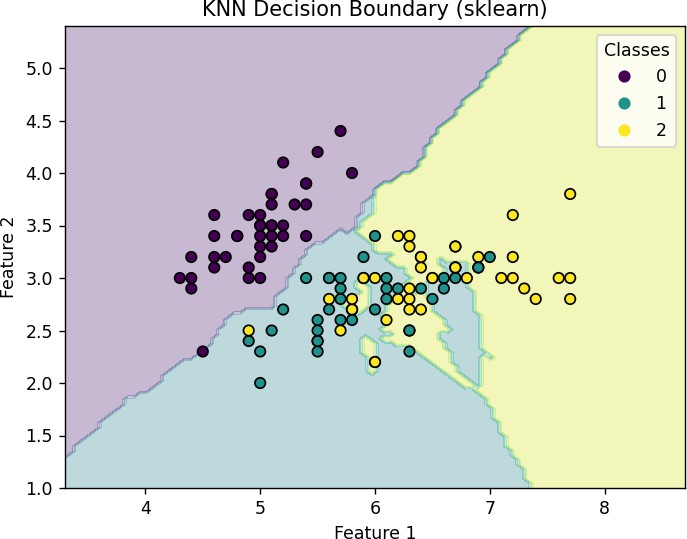
cm = confusion\_matrix(y\_test, y\_pred\_knn) plt.figure(figsize=(6, 5))

sns.heatmap(cm, annot=True, fmt="d", cmap="Blues", xticklabels=['Setosa', 'Versicolor', 'Virginica'], yticklabels=['Setosa', 'Versicolor', 'Virginica'])

plt.xlabel('Predicted') plt.ylabel('Actual')

plt.title('Confusion Matrix for KNN (sklearn)') plt.show()

**CODE OUTPUT:**

****

## KNN (WITHOUT LIBRARIES):

**CODE:**

import numpy as np

import matplotlib.pyplot as plt import seaborn as sns

from collections import Counter

def euclidean\_distance(x1, x2):

return np.sqrt(np.sum((x1 - x2) \*\* 2))

class CustomKNN:

def init (self, k=3):

self.k = k

def fit(self, X\_train, y\_train):

self.X\_train = X\_train self.y\_train = y\_train

def predict(self, X\_test):

predictions = [self.\_predict(x) for x in X\_test]

return np.array(predictions)

def \_predict(self, x):

# Compute distances

distances = [euclidean\_distance(x, x\_train) for x\_train in self.X\_train] # Get k nearest labels

k\_indices = np.argsort(distances)[:self.k]

k\_nearest\_labels = [self.y\_train[i] for i in k\_indices] # Majority vote

most\_common = Counter(k\_nearest\_labels).most\_common(1) return most\_common[0][0]

import pandas as pd

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

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

df = pd.read\_csv("https://raw.githubusercontent.com/mwaskom/seaborn-data/master/iris.csv")

# Convert categorical target ('species') into numeric

df['species'] = df['species'].astype('category').cat.codes # 0: setosa, 1: versicolor, 2: virginica

# Select features (using first 2 for visualization)

X = df[['sepal\_length', 'sepal\_width']].values y = df['species'].values

# Split data into training and testing (80%-20%)

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

knn\_custom = CustomKNN(k=5) knn\_custom.fit(X\_train, y\_train)

y\_pred\_custom = knn\_custom.predict(X\_test)

# Evaluate

accuracy\_custom = accuracy\_score(y\_test, y\_pred\_custom) print("Custom KNN Accuracy:", accuracy\_custom)

def plot\_decision\_boundary(model, X, y):

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

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

xx, yy = np.meshgrid(np.linspace(x\_min, x\_max, 100), np.linspace(y\_min, y\_max, 100))

# Predict on grid points

Z = np.array([model.\_predict(np.array([a, b])) for a, b in zip(xx.ravel(), yy.ravel())]) Z = Z.reshape(xx.shape)

plt.contourf(xx, yy, Z, alpha=0.3)

scatter = plt.scatter(X[:, 0], X[:, 1], c=y, edgecolor='k') plt.xlabel("Feature 1")

plt.ylabel("Feature 2")

plt.title("KNN Decision Boundary (Custom Implementation)") plt.legend(\*scatter.legend\_elements(), title="Classes") plt.show()

plot\_decision\_boundary(knn\_custom, X\_train, y\_train)

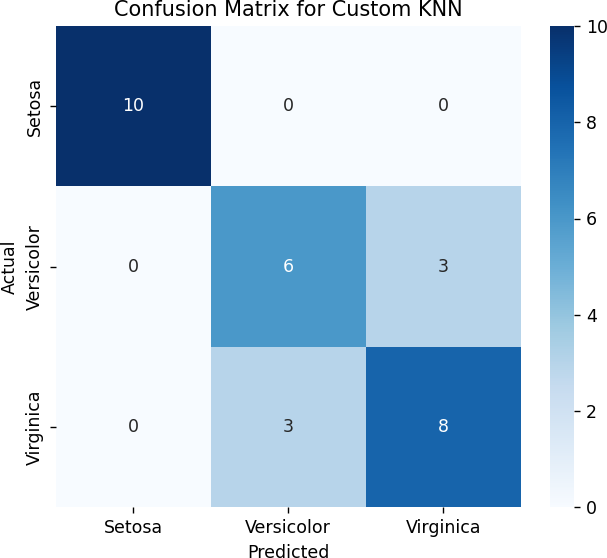
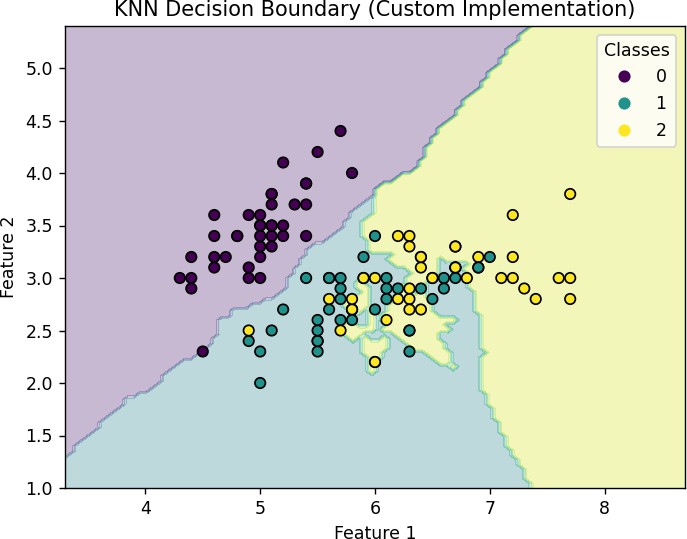
cm = confusion\_matrix(y\_test, y\_pred\_custom) plt.figure(figsize=(6, 5))

sns.heatmap(cm, annot=True, fmt="d", cmap="Blues", xticklabels=['Setosa', 'Versicolor', 'Virginica'], yticklabels=['Setosa', 'Versicolor', 'Virginica'])

plt.xlabel('Predicted') plt.ylabel('Actual')

plt.title('Confusion Matrix for Custom KNN') plt.show()

**CODE OUTPUT:**

****

**LEARNING OUTCOME:**

# EXPERIMENT 6

**AIM:** Write a program to Implement classification using SVM using library and without using library

**THEORY:**

## SVM (WITH LIBRARIES):

**CODE:**

# Import required libraries import numpy as np

import pandas as pd

import matplotlib.pyplot as plt import seaborn as sns

from sklearn import datasets

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

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

# Load the dataset

iris = datasets.load\_iris()

df = pd.DataFrame(iris.data, columns=iris.feature\_names) df['target'] = iris.target # Add the target column

df['species'] = df['target'].map({0: 'Setosa', 1: 'Versicolor', 2: 'Virginica'}) # Map to class names

# Display first 5 rows

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

# Data Visualization - Pairplot

sns.pairplot(df, hue="species", diag\_kind="kde", palette="husl") plt.show()

# Correlation Heatmap plt.figure(figsize=(8,6))

sns.heatmap(df.corr(), annot=True, cmap='coolwarm', fmt=".2f") plt.title("Feature Correlation Heatmap")

plt.show()

# Split data into train & test sets

X = df.iloc[:, :-2] # Features (excluding target & species columns) y = df['target'] # Target labels

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

# Train SVM model

svm\_model = SVC(kernel='linear', C=1.0) # Linear kernel svm\_model.fit(X\_train, y\_train)

# Predictions

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

# Model Evaluation

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

print("\nModel Accuracy: {:.2f}%".format(accuracy \* 100))

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

# Confusion Matrix Visualization

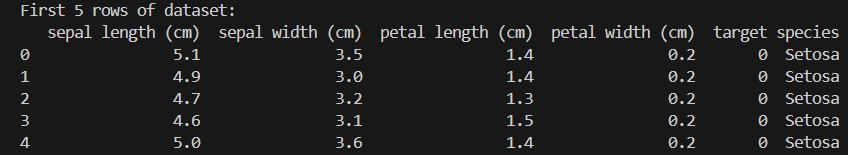
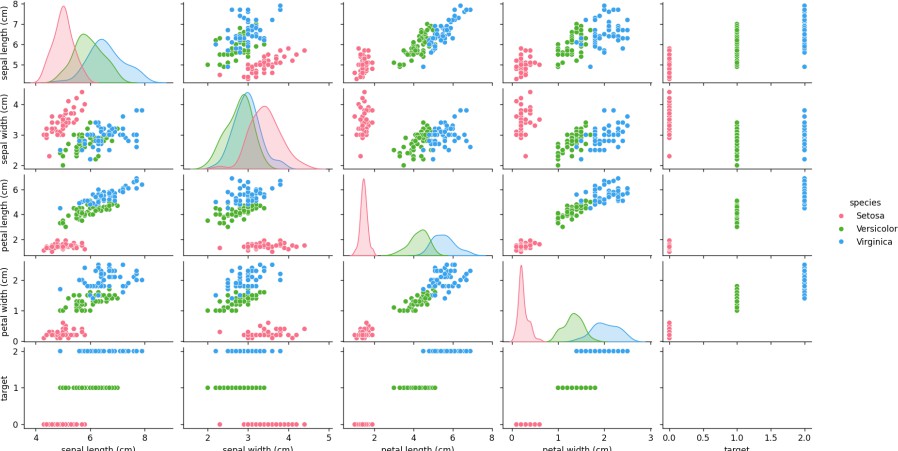
conf\_matrix = confusion\_matrix(y\_test, y\_pred) plt.figure(figsize=(6,4))

sns.heatmap(conf\_matrix, annot=True, cmap="Blues", fmt='d',

xticklabels=iris.target\_names, yticklabels=iris.target\_names) plt.xlabel("Predicted Label")

plt.ylabel("True Label") plt.title("Confusion Matrix") plt.show()

**CODE OUTPUT:**

****

## SVM (WITHOUT LIBRARIES):

**CODE:**

import numpy as np

import matplotlib.pyplot as plt import seaborn as sns

from sklearn.datasets import make\_blobs

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

# Generate synthetic dataset

X, y = make\_blobs(n\_samples=100, centers=2, random\_state=42, cluster\_std=1.5) y = np.where(y == 0, -1, 1) # Convert labels from {0,1} to {-1,1}

# Split 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)

# Visualizing the data plt.figure(figsize=(8,6))

sns.scatterplot(x=X\_train[:, 0], y=X\_train[:, 1], hue=y\_train, palette={-1: 'red', 1: 'blue'}) plt.title("Training Data Distribution")

plt.show()

# SVM Model from Scratch class SVM:

def init (self, learning\_rate=0.001, lambda\_param=0.01, epochs=1000): self.lr = learning\_rate

self.lambda\_param = lambda\_param self.epochs = epochs

self.w = None self.b = None

def fit(self, X, y):

n\_samples, n\_features = X.shape

self.w = np.zeros(n\_features) # Initialize weights self.b = 0 # Initialize bias

for \_ in range(self.epochs):

for i, x\_i in enumerate(X):

condition = y[i] \* (np.dot(x\_i, self.w) + self.b) >= 1 if condition:

self.w -= self.lr \* (2 \* self.lambda\_param \* self.w) else:

self.w -= self.lr \* (2 \* self.lambda\_param \* self.w - np.dot(x\_i, y[i])) self.b -= self.lr \* y[i]

def predict(self, X):

return np.sign(np.dot(X, self.w) + self.b)

# Train SVM Model

svm = SVM() svm.fit(X\_train, y\_train)

# Predictions

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

# Accuracy Calculation

accuracy = np.mean(y\_pred == y\_test)

print(f"Model Accuracy: {accuracy \* 100:.2f}%")

# Visualizing Decision Boundary

def plot\_decision\_boundary(X, y, model):

fig, ax = plt.subplots(figsize=(8,6))

sns.scatterplot(x=X[:, 0], y=X[:, 1], hue=y, palette={-1: 'red', 1: 'blue'}, edgecolor='k')

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

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

xx, yy = np.meshgrid(np.linspace(x\_min, x\_max, 50), np.linspace(y\_min, y\_max, 50)) Z = model.predict(np.c\_[xx.ravel(), yy.ravel()])

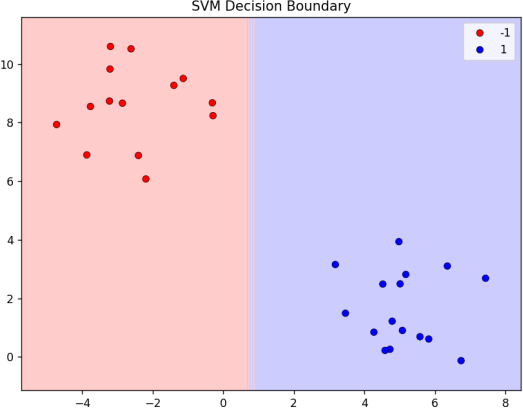
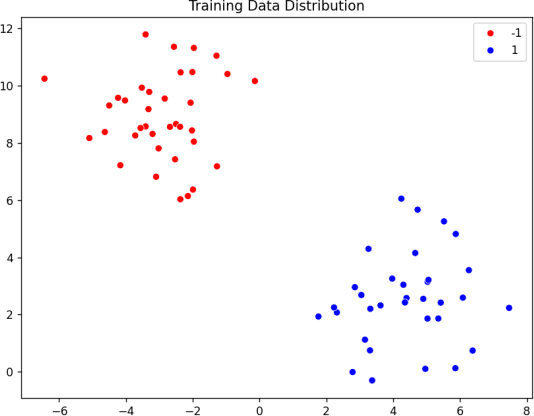
Z = Z.reshape(xx.shape)

ax.contourf(xx, yy, Z, alpha=0.2, colors=['red', 'blue']) ax.set\_title("SVM Decision Boundary")

plt.show()

# Plot the decision boundary plot\_decision\_boundary(X\_test, y\_test, svm)

**CODE OUTPUT:**

****

**LEARNING OUTCOME:**

# EXPERIMENT 7

**AIM:** Write a program for performing bagging using random forest with and without using libraries.

**THEORY:**

## RANDOM FOREST (WITH LIBRARIES):

**CODE:**

import numpy as np

import matplotlib.pyplot as plt

from sklearn.datasets import load\_iris

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

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

from matplotlib.colors import ListedColormap from sklearn.datasets import load\_iris

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

X = iris.data[:, :2] # Use only 2 features for visualization y = iris.target # Extract target labels

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

# Train Random Forest model

rf\_model = RandomForestClassifier(n\_estimators=10, max\_depth=5, bootstrap=True, random\_state=42)

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

# Make predictions

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

# Evaluate model performance

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

print(f"Random Forest Accuracy (with library): {accuracy:.4f}")

# Function to plot decision boundaries

def plot\_decision\_boundary(model, X, y, title):

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

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

xx, yy = np.meshgrid(np.linspace(x\_min, x\_max, 100), np.linspace(y\_min, y\_max, 100))

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

cmap\_light = ListedColormap(["#FFAAAA", "#AAFFAA", "#AAAAFF"]) cmap\_bold = ListedColormap(["#FF0000", "#00FF00", "#0000FF"]) plt.figure(figsize=(8, 6))

plt.contourf(xx, yy, Z, alpha=0.3, cmap=cmap\_light)

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

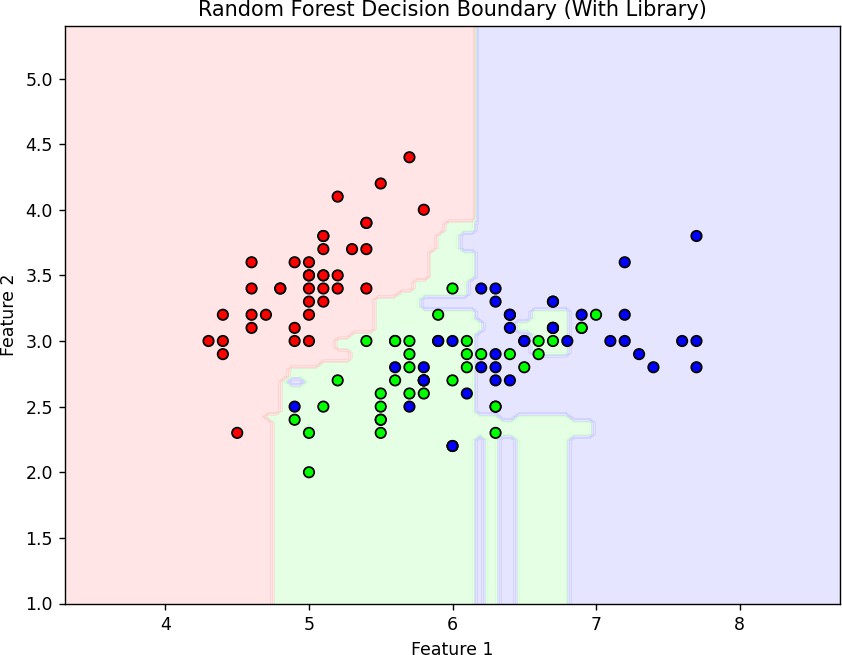
plt.xlabel("Feature 1")

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

# Plot the decision boundary of the Random Forest

plot\_decision\_boundary(rf\_model, X\_train, y\_train, "Random Forest Decision Boundary (With Library)")

**CODE OUTPUT:**

****

## RANDOM FOREST (WITHOUT LIBRARIES):

**CODE:**

import numpy as np

import matplotlib.pyplot as plt

from sklearn.tree import DecisionTreeClassifier from sklearn.datasets import load\_iris

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

from matplotlib.colors import ListedColormap from sklearn.datasets import load\_iris

iris = load\_iris()

X = iris.data[:, :2] # Use only 2 features for visualization y = iris.target # Extract target labels

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

# Bagging Parameters

n\_estimators = 10 # Number of decision trees

bootstrap\_size = int(0.8 \* len(X\_train)) # 80% samples in each bootstrap

# Initialize list to store models trees = []

# Train multiple decision trees using bootstrap samples for \_ in range(n\_estimators):

indices = np.random.choice(len(X\_train), bootstrap\_size, replace=True) X\_bootstrap, y\_bootstrap = X\_train[indices], y\_train[indices]

tree = DecisionTreeClassifier(max\_depth=5) tree.fit(X\_bootstrap, y\_bootstrap)

trees.append(tree)

# Make predictions using majority voting

predictions = np.array([tree.predict(X\_test) for tree in trees])

y\_pred = np.apply\_along\_axis(lambda x: np.bincount(x).argmax(), axis=0, arr=predictions)

# Evaluate model performance

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

print(f"Bagging with Decision Trees Accuracy (without library): {accuracy:.4f}")

# Function to plot decision boundaries for an ensemble of trees def plot\_decision\_boundary\_ensemble(models, X, y, title):

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

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

xx, yy = np.meshgrid(np.linspace(x\_min, x\_max, 100), np.linspace(y\_min, y\_max, 100))

# Aggregate predictions from all trees

pred\_votes = np.zeros((xx.shape[0] \* xx.shape[1], len(models))) for i, tree in enumerate(models):

pred\_votes[:, i] = tree.predict(np.c\_[xx.ravel(), yy.ravel()])

# Perform majority voting

final\_preds = np.apply\_along\_axis(lambda x: np.bincount(x.astype(int)).argmax(), axis=1, arr=pred\_votes) Z = final\_preds.reshape(xx.shape)

cmap\_light = ListedColormap(["#FFAAAA", "#AAFFAA", "#AAAAFF"]) cmap\_bold = ListedColormap(["#FF0000", "#00FF00", "#0000FF"])

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

plt.contourf(xx, yy, Z, alpha=0.3, cmap=cmap\_light)

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

plt.xlabel("Feature 1")

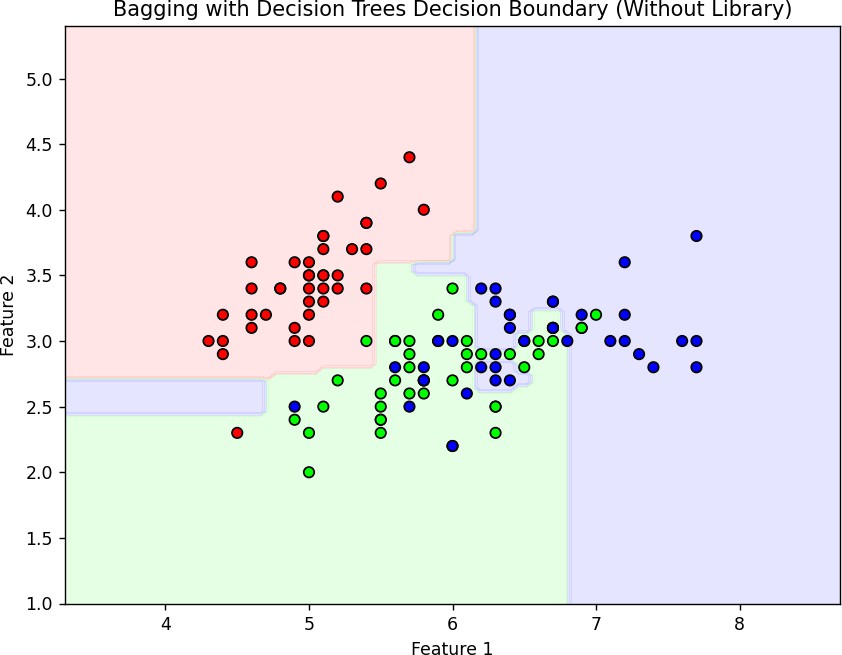
plt.ylabel("Feature 2") plt.show()

# Plot the decision boundary of the Bagging Classifier

plot\_decision\_boundary\_ensemble(trees, X\_train, y\_train, "Bagging with Decision Trees Decision Boundary (Without Library)")

**CODE OUTPUT:**

****

****

**LEARNING OUTCOME:**

# EXPERIMENT 8

**AIM:** Write a program to Implement Naïve Bayes using library and without using library

**THEORY:**

## NAÏVE BAYES (WITH LIBRARIES):

**CODE:**

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

from sklearn.preprocessing import StandardScaler from sklearn.metrics import accuracy\_score

from sklearn.datasets import make\_classification from collections import Counter

# Generate synthetic dataset with 1000 samples and 2 features

X, y = make\_classification(n\_samples=1000, n\_features=2, n\_informative=2, n\_redundant=0, n\_clusters\_per\_class=1, random\_state=42)

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

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

# Print class distributions in the training and testing set print(f"Class distribution in training set: {Counter(y\_train)}") print(f"Class distribution in testing set: {Counter(y\_test)}")

# Feature scaling using StandardScaler scaler = StandardScaler()

# Scale the training and testing data

X\_train\_scaled = scaler.fit\_transform(X\_train) X\_test\_scaled = scaler.transform(X\_test)

# Initialize the Gaussian Naive Bayes classifier nb = GaussianNB()

# Train the model on the scaled data

nb.fit(X\_train\_scaled, y\_train)

# Predict on the test data

y\_pred\_scaled = nb.predict(X\_test\_scaled)

# Calculate accuracy

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

print(f'Accuracy of Naive Bayes classifier (scaled data): {accuracy\_scaled \* 100:.2f}%')

**CODE OUTPUT:**

****

## NAÏVE BAYES (WITHOUT LIBRARIES):

**CODE:**

import numpy as np

# Sample data (expanded dataset from earlier, you can replace this with your data) X\_train = np.array([[1.1, 2.0], [1.9, 3.2], [2.3, 1.5], [3.1, 4.4], [1.5, 2.3], [3.0, 3.5],

[2.5, 3.5], [1.8, 2.5], [2.0, 2.8], [3.0, 3.0], [2.1, 2.9], [1.7, 2.4],

[2.9, 3.3], [3.5, 4.1], [1.3, 2.2], [3.2, 3.6], [2.8, 2.6], [1.6, 2.1],

[2.2, 2.3], [3.3, 3.8], [1.4, 2.7], [3.4, 4.0], [2.6, 3.4], [2.7, 2.5],

[2.4, 3.2], [1.2, 2.5], [3.0, 3.2], [1.0, 2.1], [2.8, 3.0], [1.9, 2.7],

[2.3, 3.1], [3.1, 3.7], [1.8, 2.3], [2.7, 3.6], [1.5, 2.4], [3.0, 3.5],

[2.2, 3.0], [2.9, 3.8], [3.2, 4.4], [1.4, 2.2], [1.6, 2.8], [2.6, 2.7],

[3.3, 4.0], [2.1, 2.5], [3.4, 3.9], [2.0, 2.2], [1.7, 2.6], [2.5, 3.3]])

y\_train = np.array([0, 0, 1, 1, 0, 1, 1, 0, 1, 1, 0, 0, 1, 1, 0, 1, 1, 0, 0, 1, 0, 1, 0, 1,

0, 1, 0, 1, 1, 0, 1, 0, 1, 1, 0, 1, 0, 1, 0, 1, 1, 0, 0, 1, 1, 0, 0, 1])

# Test data (small test set for demonstration)

X\_test = np.array([[2.1, 2.8], [1.7, 2.6], [2.8, 3.5], [3.0, 3.3], [2.2, 2.4], [1.8, 2.1]])

y\_test = np.array([1, 0, 1, 1, 0, 1]) # Actual labels

# Naive Bayes function (for simplicity, assume Gaussian Naive Bayes) def naive\_bayes(X\_train, y\_train, X\_test):

# Calculating prior probabilities classes = np.unique(y\_train)

prior\_probs = {class\_val: np.mean(y\_train == class\_val) for class\_val in classes}

# Calculating likelihoods (Gaussian distributions) likelihoods = {}

for class\_val in classes:

X\_class = X\_train[y\_train == class\_val] mean = X\_class.mean(axis=0)

std = X\_class.std(axis=0)

likelihoods[class\_val] = (mean, std)

# Making predictions for test data predictions = []

for x in X\_test:

# Calculate the posterior probability for each class posteriors = {}

for class\_val in classes:

mean, std = likelihoods[class\_val]

likelihood = np.prod((1 / (std \* np.sqrt(2 \* np.pi))) \* np.exp(-0.5 \* ((x - mean) / std) \*\* 2)) posterior = prior\_probs[class\_val] \* likelihood

posteriors[class\_val] = posterior

# Predict the class with the highest posterior probability predicted\_class = max(posteriors, key=posteriors.get)

predictions.append(predicted\_class)

return np.array(predictions)

# Get predictions

predictions = naive\_bayes(X\_train, y\_train, X\_test)

# Calculate accuracy

accuracy = np.mean(predictions == y\_test) \* 100

print(f"Predictions from Naive Bayes (from scratch): {predictions}") print(f"Accuracy: {accuracy:.2f}%")

**CODE OUTPUT:**

****

**LEARNING OUTCOME:**

# EXPERIMENT 9

**AIM:** Write a program for performing decision trees using and without using libraries.

**THEORY:**

## DECISION TREES (WITH LIBRARIES):

**CODE:**

import numpy as np import pandas as pd

import matplotlib.pyplot as plt import seaborn as sns

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

from sklearn.tree import DecisionTreeClassifier, DecisionTreeRegressor, plot\_tree from sklearn.metrics import accuracy\_score, mean\_squared\_error

from matplotlib.colors import ListedColormap

df = pd.read\_csv("https://raw.githubusercontent.com/mwaskom/seaborn-data/master/iris.csv")

# Convert categorical target ('species') into numeric

df['species'] = df['species'].astype('category').cat.codes # 0: Setosa, 1: Versicolor, 2: Virginica

X\_cls = df[['sepal\_length', 'sepal\_width']].values # Using only two features for visualization y\_cls = df['species'].values

# Split data

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

# Train Decision Tree Classifier

clf = DecisionTreeClassifier(max\_depth=4, random\_state=42) clf.fit(X\_train\_cls, y\_train\_cls)

y\_pred\_cls = clf.predict(X\_test\_cls)

# Evaluate

accuracy = accuracy\_score(y\_test\_cls, y\_pred\_cls) print(f"Decision Tree Classifier Accuracy: {accuracy:.4f}")

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

plot\_tree(clf, feature\_names=['sepal\_length', 'sepal\_width'], class\_names=['Setosa', 'Versicolor', 'Virginica'], filled=True)

plt.title("Decision Tree Visualization (Classification)") plt.show()

plt.figure(figsize=(6, 4))

sns.barplot(x=['sepal\_length', 'sepal\_width'], y=clf.feature\_importances\_, palette="coolwarm") plt.xlabel("Features")

plt.ylabel("Importance")

plt.title("Feature Importance in Decision Tree Classifier") plt.xticks(rotation=45)

plt.show()

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

y\_min, y\_max = X\_cls[:, 1].min() - 1, X\_cls[:, 1].max() + 1 xx, yy = np.meshgrid(np.arange(x\_min, x\_max, 0.02),

np.arange(y\_min, y\_max, 0.02))

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

plt.figure(figsize=(7, 5))

cmap = ListedColormap(["#FF9999", "#99FF99", "#9999FF"]) plt.contourf(xx, yy, Z, alpha=0.3, cmap=cmap)

scatter = plt.scatter(X\_cls[:, 0], X\_cls[:, 1], c=y\_cls, cmap=cmap, edgecolor='k') plt.xlabel("Sepal Length")

plt.ylabel("Sepal Width")

plt.title("Decision Boundary (Classification)")

plt.legend(handles=scatter.legend\_elements()[0], labels=['Setosa', 'Versicolor', 'Virginica'])

plt.show()

X\_reg = df[['sepal\_width', 'petal\_length', 'petal\_width']].values y\_reg = df['sepal\_length'].values

# Split data

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

# Train Decision Tree Regressor

reg = DecisionTreeRegressor(max\_depth=4, random\_state=42) reg.fit(X\_train\_reg, y\_train\_reg)

y\_pred\_reg = reg.predict(X\_test\_reg)

# Evaluate

mse = mean\_squared\_error(y\_test\_reg, y\_pred\_reg) print(f"Decision Tree Regressor MSE: {mse:.4f}")

plt.figure(figsize=(6, 4))

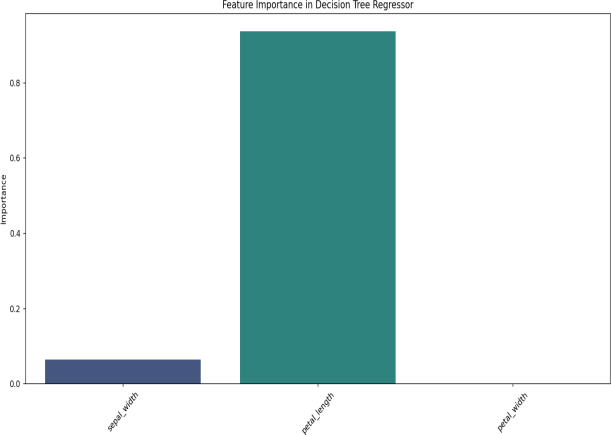
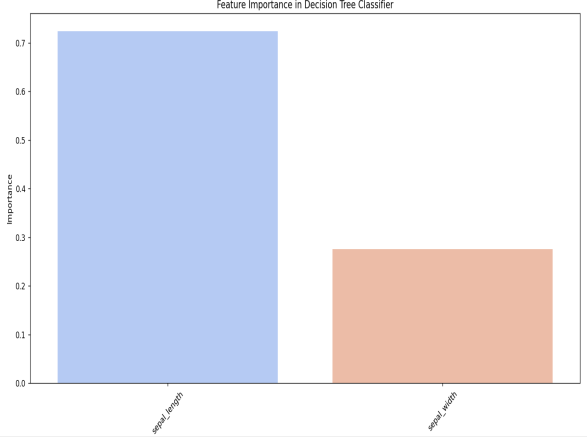
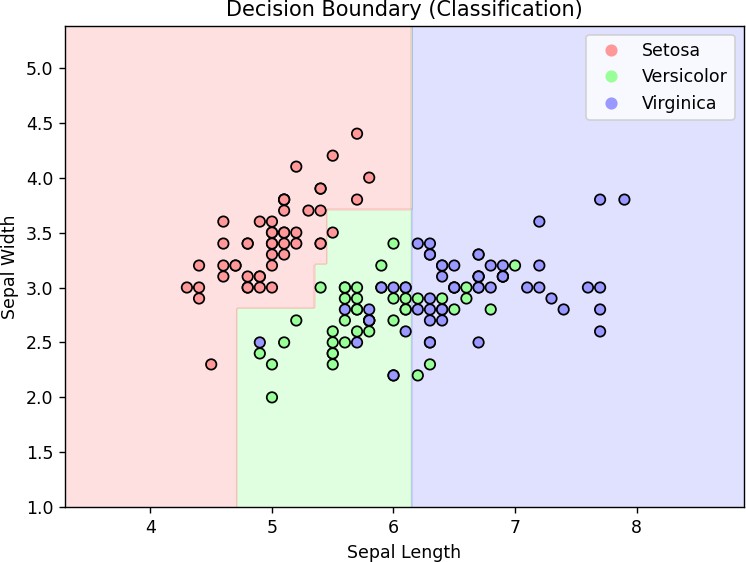
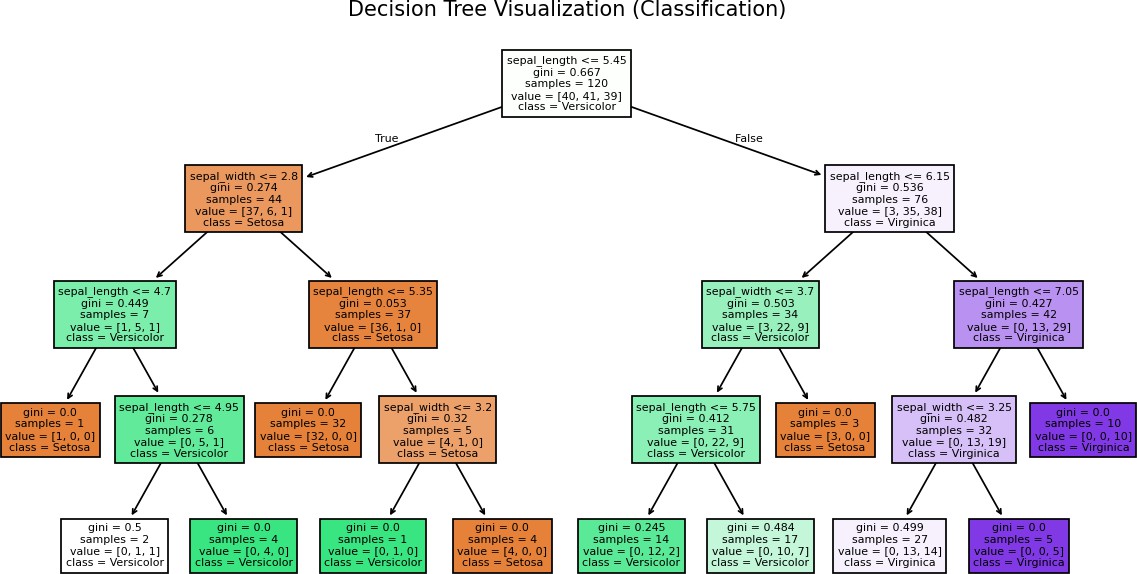
sns.barplot(x=['sepal\_width', 'petal\_length', 'petal\_width'], y=reg.feature\_importances\_, palette="viridis")

plt.xlabel("Features") plt.ylabel("Importance")

plt.title("Feature Importance in Decision Tree Regressor") plt.xticks(rotation=45)

plt.show()

**CODE OUTPUT:**

****

## DECISION TREES (WITHOUT LIBRARIES):

**CODE:**

import numpy as np import pandas as pd

import matplotlib.pyplot as plt import seaborn as sns

from matplotlib.colors import ListedColormap

#

# ‘’z Helper Functions for Decision Tree #

def gini\_impurity(y):

"""Calculate Gini impurity for a set of labels.""" unique\_classes, counts = np.unique(y, return\_counts=True) probabilities = counts / len(y)

return 1 - np.sum(probabilities \*\* 2)

def best\_split(X, y):

"""Find the best split by checking all possible thresholds for all features.""" best\_feature, best\_threshold, best\_gain = None, None, 0

current\_impurity = gini\_impurity(y)

for feature\_index in range(X.shape[1]): thresholds = np.unique(X[:, feature\_index])

for threshold in thresholds:

left\_mask = X[:, feature\_index] <= threshold right\_mask = ~left\_mask

if np.sum(left\_mask) == 0 or np.sum(right\_mask) == 0:

continue

left\_impurity = gini\_impurity(y[left\_mask])

right\_impurity = gini\_impurity(y[right\_mask])

weighted\_impurity = (np.sum(left\_mask) \* left\_impurity + np.sum(right\_mask) \* right\_impurity) / len(y)

gain = current\_impurity - weighted\_impurity

if gain > best\_gain:

best\_feature, best\_threshold, best\_gain = feature\_index, threshold, gain

return best\_feature, best\_threshold

class DecisionTreeNode:

"""A single node in the Decision Tree."""

def init (self, feature=None, threshold=None, left=None, right=None, value=None): self.feature = feature

self.threshold = threshold self.left = left

self.right = right self.value = value

def is\_leaf(self):

return self.value is not None

class DecisionTreeClassifierCustom: """Custom Decision Tree Classifier.""" def init (self, max\_depth=5):

self.max\_depth = max\_depth self.tree = None

def fit(self, X, y, depth=0):

"""Recursively builds the decision tree."""

if len(set(y)) == 1: # If all labels are the same return DecisionTreeNode(value=y[0])

if depth >= self.max\_depth or len(y) < 2: # Stopping criteria (FIXED ERROR) return DecisionTreeNode(value=np.bincount(y).argmax())

feature, threshold = best\_split(X, y)

if feature is None:

return DecisionTreeNode(value=np.bincount(y).argmax())

left\_mask = X[:, feature] <= threshold right\_mask = ~left\_mask

left\_subtree = self.fit(X[left\_mask], y[left\_mask], depth + 1)

right\_subtree = self.fit(X[right\_mask], y[right\_mask], depth + 1)

return DecisionTreeNode(feature, threshold, left\_subtree, right\_subtree)

def predict\_one(self, x, node): """Predict a single sample.""" if node.is\_leaf():

return node.value

if x[node.feature] <= node.threshold: return self.predict\_one(x, node.left)

else:

return self.predict\_one(x, node.right)

def predict(self, X):

"""Predict multiple samples."""

return np.array([self.predict\_one(x, self.tree) for x in X])

def train(self, X, y):

"""Train the Decision Tree Classifier.""" self.tree = self.fit(X, y)

#

# ‘’z Load Dataset (Iris) #

df = pd.read\_csv("https://raw.githubusercontent.com/mwaskom/seaborn-data/master/iris.csv") df['species'] = df['species'].astype('category').cat.codes # 0: Setosa, 1: Versicolor, 2: Virginica

X = df[['sepal\_length', 'sepal\_width']].values y = df['species'].values

# Train-Test Split

np.random.seed(42)

indices = np.random.permutation(len(X)) train\_size = int(0.8 \* len(X))

train\_idx, test\_idx = indices[:train\_size], indices[train\_size:]

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

#

# ‘’z Train Custom Decision Tree #

tree = DecisionTreeClassifierCustom(max\_depth=4) tree.train(X\_train, y\_train)

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

# Accuracy

accuracy = np.mean(y\_pred == y\_test)

print(f"Custom Decision Tree Accuracy: {accuracy:.4f}")

#

# ‘’z Decision Boundary Visualization #

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

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

xx, yy = np.meshgrid(np.arange(x\_min, x\_max, 0.02), np.arange(y\_min, y\_max, 0.02))

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

plt.figure(figsize=(7, 5))

cmap = ListedColormap(["#FF9999", "#99FF99", "#9999FF"]) plt.contourf(xx, yy, Z, alpha=0.3, cmap=cmap)

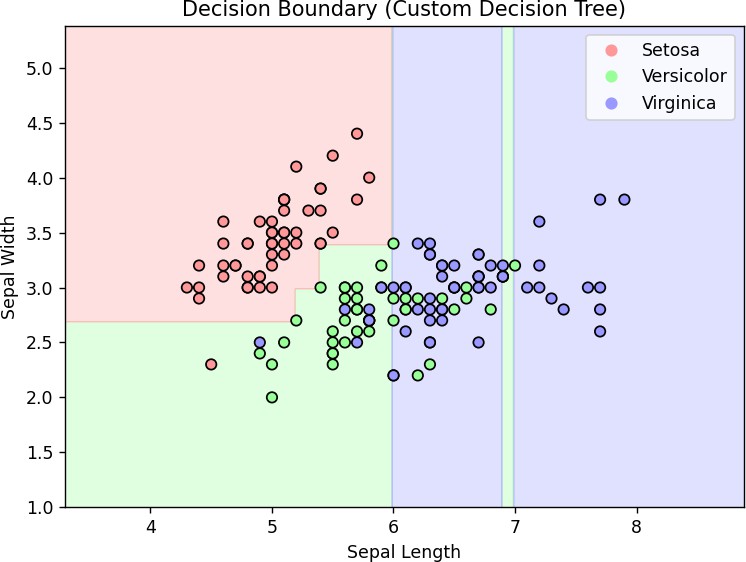
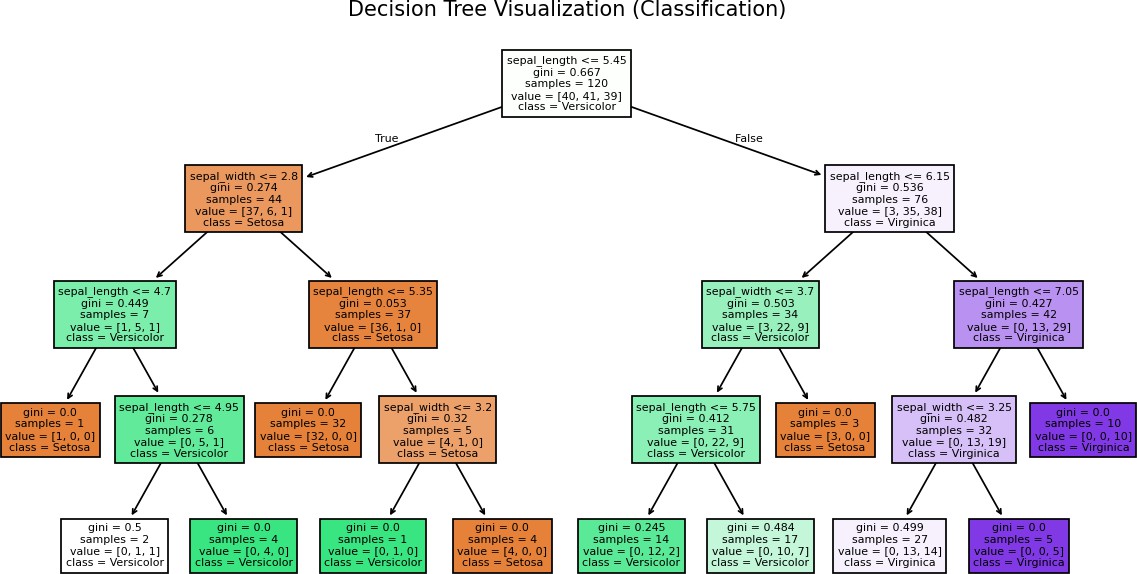
scatter = plt.scatter(X[:, 0], X[:, 1], c=y, cmap=cmap, edgecolor='k') plt.xlabel("Sepal Length")

plt.ylabel("Sepal Width")

plt.title("Decision Boundary (Custom Decision Tree)")

plt.legend(handles=scatter.legend\_elements()[0], labels=['Setosa', 'Versicolor', 'Virginica']) plt.show()

**CODE OUTPUT:**

****

**LEARNING OUTCOME:**

# EXPERIMENT 10

**AIM:** Write a program for performing K-means clustering with and without using libraries.

**THEORY:**

## K-MEANS CLUSTERING (WITH LIBRARIES):

**CODE:**

import numpy as np

import matplotlib.pyplot as plt import pandas as pd

from sklearn.cluster import KMeans from sklearn import datasets

# Load Iris dataset

iris = datasets.load\_iris()

X = iris.data[:, [2, 3]] # Using petal length and petal width for clustering

# Using the Elbow Method to find the optimal number of clusters wcss = []

for i in range(1, 11):

kmeans = KMeans(n\_clusters=i, init='k-means++', n\_init=10, random\_state=42) kmeans.fit(X)

wcss.append(kmeans.inertia\_)

# Plot the Elbow Method

plt.plot(range(1, 11), wcss, marker='o') plt.title('The Elbow Curve')

plt.xlabel('Number of clusters') plt.ylabel('WCSS')

plt.show()

# Applying K-Means with the chosen number of clusters (3 for Iris dataset)

kmeans = KMeans(n\_clusters=4, init='k-means++', n\_init=10, random\_state=42) y\_kmeans = kmeans.fit\_predict(X)

# Visualizing the clusters colors = ['red', 'blue', 'green'] for i in range(3):

plt.scatter(X[y\_kmeans == i, 0], X[y\_kmeans == i, 1], s=11, c=colors[i], label=f'Cluster {i+1}')

# Plot centroids

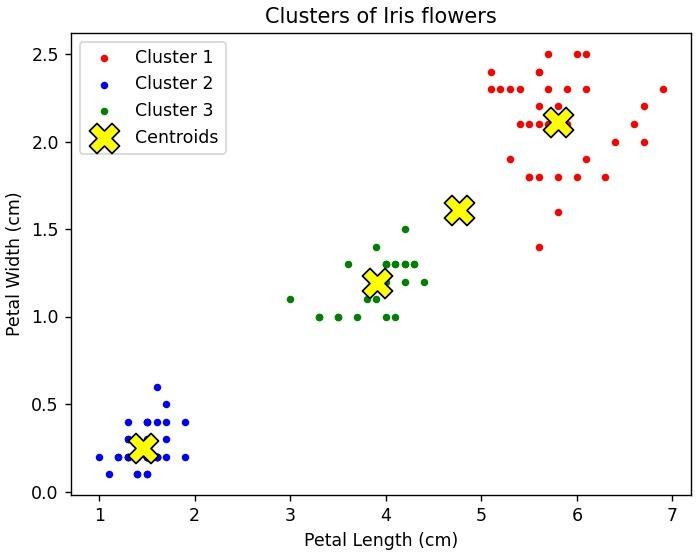
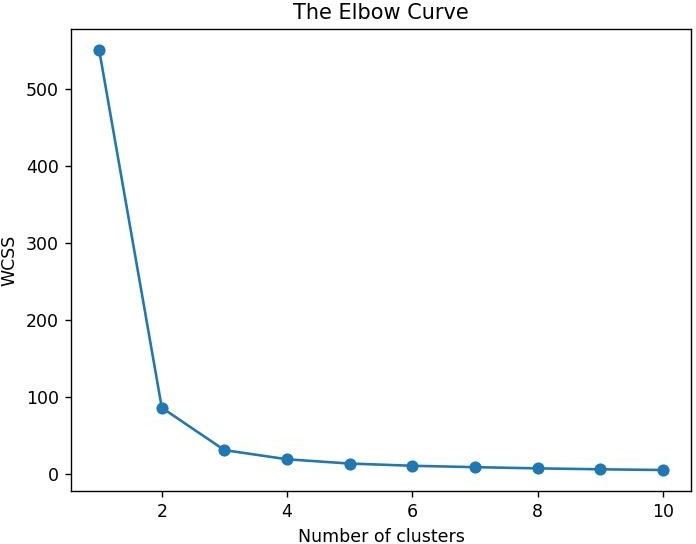
plt.scatter(kmeans.cluster\_centers\_[:, 0], kmeans.cluster\_centers\_[:, 1], s=300, c='yellow', edgecolors='black', marker='X', label='Centroids')

plt.title('Clusters of Iris flowers') plt.xlabel('Petal Length (cm)')

plt.ylabel('Petal Width (cm)') plt.legend()

plt.show()

**CODE OUTPUT:**

****

## K-MEANS CLUSTERING (WITHOUT LIBRARIES):

**CODE:**

import random

import matplotlib.pyplot as plt

# Load Iris dataset manually iris\_data = [

[1.4, 0.2], [1.4, 0.2], [1.3, 0.2], [1.5, 0.2], [1.4, 0.2],

[4.7, 1.4], [4.5, 1.5], [4.9, 1.5], [4.0, 1.3], [4.6, 1.5],

[5.1, 1.9], [5.9, 2.1], [5.7, 2.3], [5.2, 2.0], [5.0, 1.9]

]

X = iris\_data # Using petal length and petal width for clustering

# Euclidean distance function def euclidean\_distance(p1, p2):

return sum((x - y) \*\* 2 for x, y in zip(p1, p2)) \*\* 0.5

# K-Means implementation without libraries def kmeans\_manual(data, k, max\_iters=100):

centroids = random.sample(data, k) for \_ in range(max\_iters):

clusters = [[] for \_ in range(k)] for point in data:

distances = [euclidean\_distance(point, centroid) for centroid in centroids] cluster\_idx = distances.index(min(distances))

clusters[cluster\_idx].append(point) new\_centroids = []

for i, cluster in enumerate(clusters):

if cluster:

new\_centroids.append([sum(dim) / len(cluster) for dim in zip(\*cluster)]) else:

new\_centroids.append(centroids[i]) # Retain previous centroid if cluster is empty if new\_centroids == centroids:

break

centroids = new\_centroids return clusters, centroids

# Running K-Means k = 3

clusters, centroids = kmeans\_manual(X, k)

# Visualizing clusters

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

for i, cluster in enumerate(clusters):

if cluster: # Ensure cluster is not empty x\_vals, y\_vals = zip(\*cluster)

plt.scatter(x\_vals, y\_vals, s=100, c=colors[i], label=f'Cluster {i+1}')

# Plot centroids

x\_centroids, y\_centroids = zip(\*centroids)

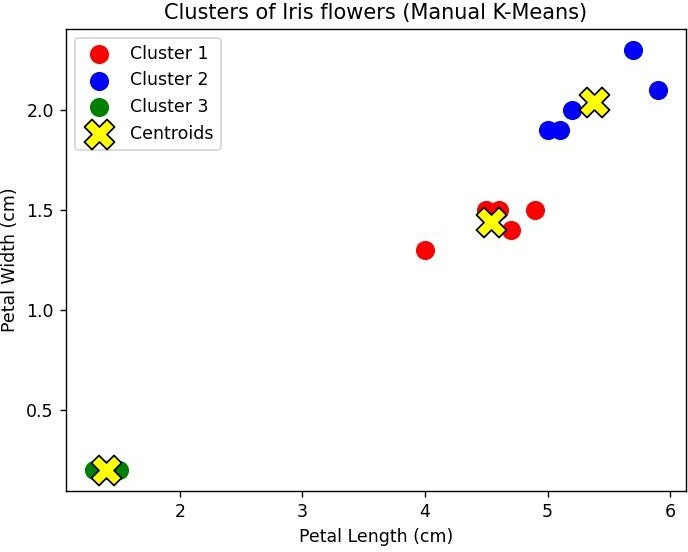
plt.scatter(x\_centroids, y\_centroids, s=300, c='yellow', edgecolors='black', marker='X', label='Centroids')

plt.title('Clusters of Iris flowers (Manual K-Means)') plt.xlabel('Petal Length (cm)')

plt.ylabel('Petal Width (cm)') plt.legend()

plt.show()

**CODE OUTPUT:**

****

**LEARNING OUTCOME:**

# EXPERIMENT 11

**AIM:** Write a program for showing comparison of machine learning algorithms based on different-different parameters

**THEORY:**

## CODE:

import numpy as np import pandas as pd

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

from sklearn.metrics import accuracy\_score, precision\_score, recall\_score, f1\_score from sklearn.ensemble import RandomForestClassifier

from sklearn.svm import SVC

from sklearn.linear\_model import LogisticRegression from sklearn.datasets import load\_iris

# Load dataset data = load\_iris()

X = data.data y = data.target

# Split data

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

# Define models models = {

"Random Forest": RandomForestClassifier(), "SVM": SVC(),

"Logistic Regression": LogisticRegression(max\_iter=200)

}

# Evaluate models results = []

for name, model in models.items(): model.fit(X\_train, y\_train)

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

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

precision = precision\_score(y\_test, y\_pred, average='weighted') recall = recall\_score(y\_test, y\_pred, average='weighted')

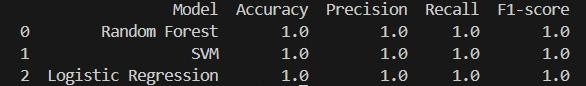
f1 = f1\_score(y\_test, y\_pred, average='weighted')

results.append([name, accuracy, precision, recall, f1])

# Convert to DataFrame and display

results\_df = pd.DataFrame(results, columns=["Model", "Accuracy", "Precision", "Recall", "F1-score"]) print(results\_df)

**CODE OUTPUT:**

****

**LEARNING OUTCOME:**