**PAMIDI NARENDRA**

**(192224128)**

**LAB PROGRAMS WITH OUTPUTS**

**EXPERIMENT:1**

**PROGRAM:** confusion matrix

import numpy as np

from sklearn.metrics import confusion\_matrix

import seaborn as sns

import matplotlib.pyplot as plt

actual = np.array(['Dog','Dog','Dog','Not Dog','Dog','Not Dog','Dog','Dog','Not Dog','Not Dog'])

predicted = np.array(['Dog','Not Dog','Dog','Not Dog','Dog','Dog','Dog','Dog','Not Dog','Not Dog'])

cm = confusion\_matrix(actual, predicted)

sns.heatmap(cm, annot=True, fmt='g', xticklabels=['Dog','Not Dog'], yticklabels=['Dog','Not Dog'])

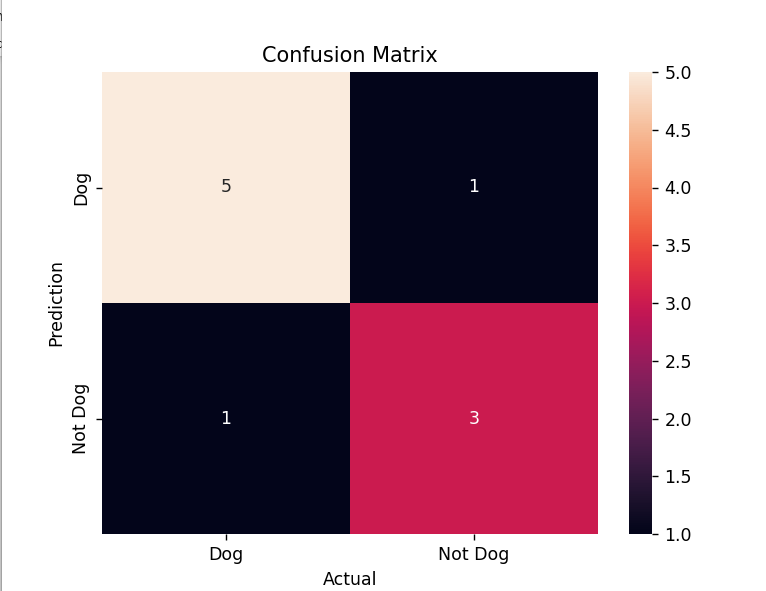
plt.ylabel('Prediction')

plt.xlabel('Actual')

plt.title('Confusion Matrix')

plt.show()

**OUTPUT:**



**EXPERIMENT:2**

**PROGRAM:** 2 class confusion matrix

from sklearn.datasets import load\_breast\_cancer

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

from sklearn.tree import DecisionTreeClassifier

from sklearn.metrics import confusion\_matrix, accuracy\_score, precision\_score, recall\_score, f1\_score

import seaborn as sns

import matplotlib.pyplot as plt

# Load the breast cancer dataset

X, y = load\_breast\_cancer(return\_X\_y=True)

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.25)

# Train the model

tree = DecisionTreeClassifier(random\_state=23)

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

# Prediction

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

# Confusion matrix

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

# Plot the confusion matrix

sns.heatmap(cm, annot=True, fmt='g', xticklabels=['malignant', 'benign'], yticklabels=['malignant', 'benign'])

plt.ylabel('Prediction')

plt.xlabel('Actual')

plt.title('Confusion Matrix')

plt.show()

# Calculate and display metrics

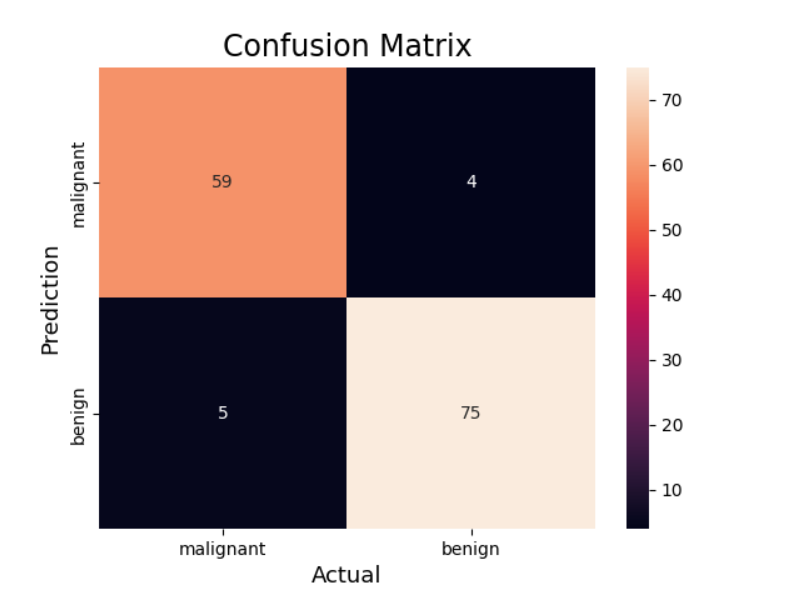
print("Accuracy :", accuracy\_score(y\_test, y\_pred))

print("Precision :", precision\_score(y\_test, y\_pred))

print("Recall :", recall\_score(y\_test, y\_pred))

print("F1-score :", f1\_score(y\_test, y\_pred))

**OUTPUT:**

****

**EXPERIMENT:3**

**PROGRAM:** multi class confusion matrix

from sklearn.datasets import load\_digits

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

from sklearn.ensemble import RandomForestClassifier

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

import seaborn as sns

import matplotlib.pyplot as plt

# Load the digits dataset

X, y = load\_digits(return\_X\_y=True)

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.25)

# Train the model

clf = RandomForestClassifier(random\_state=23)

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

# Prediction

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

# Confusion matrix

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

# Plot the confusion matrix

sns.heatmap(cm, annot=True, fmt='g')

plt.ylabel('Prediction')

plt.xlabel('Actual')

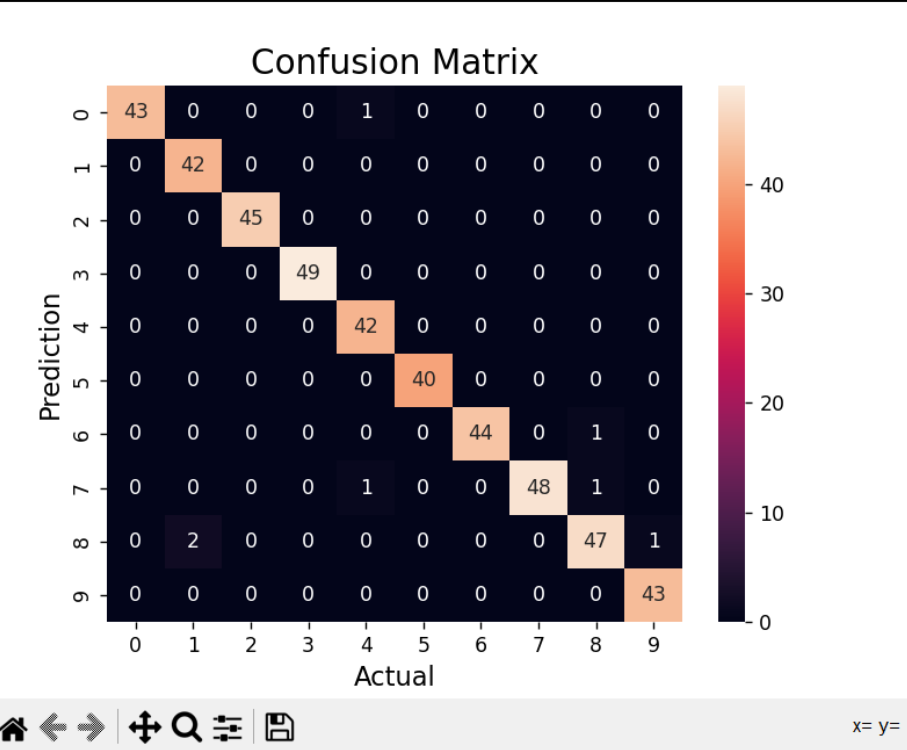
plt.title('Confusion Matrix')

plt.show()

# Calculate and display accuracy

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

**OUTPUT:**

****

**EXPERIMENT:4**

**PROGRAM:** over fitting

from sklearn.datasets import load\_digits

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

from sklearn.ensemble import RandomForestClassifier

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

import seaborn as sns

import matplotlib.pyplot as plt

# Load the digits dataset

X, y = load\_digits(return\_X\_y=True)

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.25)

# Train the model

clf = RandomForestClassifier(random\_state=23)

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

# Prediction

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

# Confusion matrix

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

# Plot the confusion matrix

sns.heatmap(cm, annot=True, fmt='g')

plt.ylabel('Prediction')

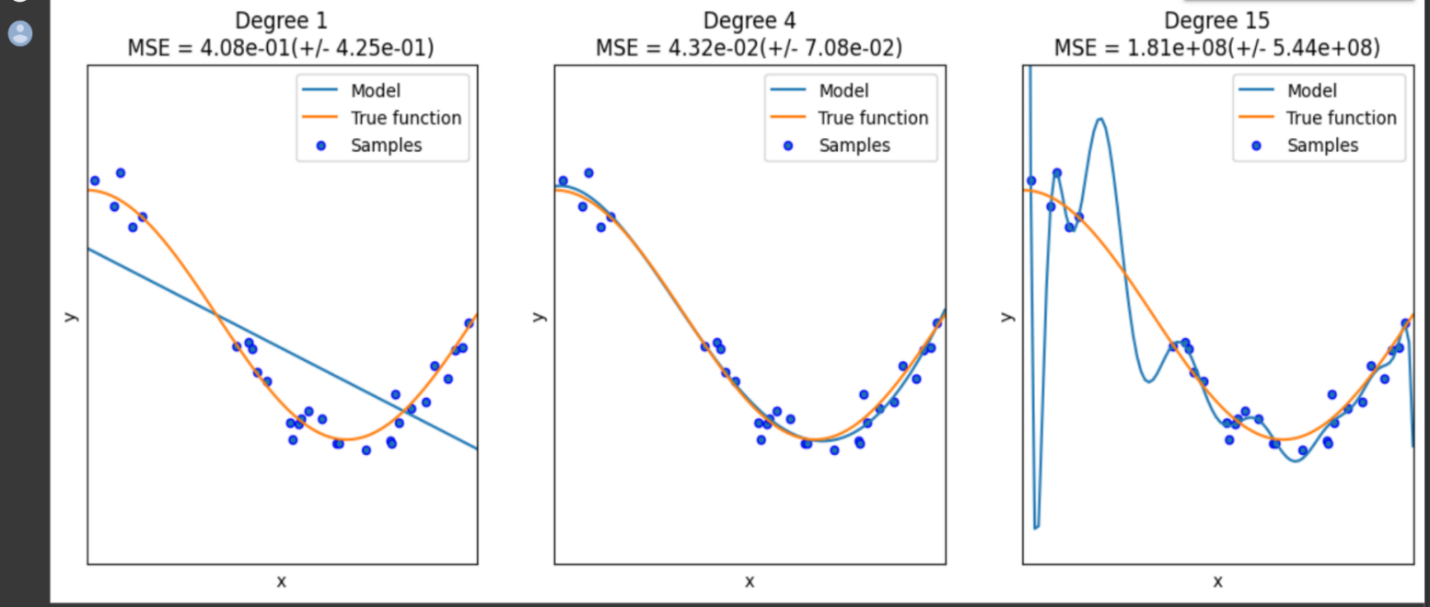
plt.xlabel('Actual')

plt.title('Confusion Matrix')

plt.show()

# Calculate and display accuracy

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

**OUTPUT: **

**EXPERIMENT:5**

**PROGRAM: LINEAR REGRESSION**

import numpy as np

import matplotlib.pyplot as plt

from sklearn.pipeline import Pipeline

from sklearn.preprocessing import PolynomialFeatures

from sklearn.linear\_model import LinearRegression

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

def true\_fun(X):

return np.cos(1.5 \* np.pi \* X)

np.random.seed(0)

n\_samples = 30

degrees = [1, 4, 15]

X = np.sort(np.random.rand(n\_samples))

y = true\_fun(X) + np.random.randn(n\_samples) \* 0.1

plt.figure(figsize=(14, 5))

for i, degree in enumerate(degrees):

ax = plt.subplot(1, len(degrees), i + 1)

plt.setp(ax, xticks=(), yticks=())

# Create and fit the model pipeline

model = Pipeline([

("polynomial\_features", PolynomialFeatures(degree=degree, include\_bias=False)),

("linear\_regression", LinearRegression())

])

model.fit(X[:, np.newaxis], y)

# Cross-validation scores

scores = cross\_val\_score(model, X[:, np.newaxis], y, scoring="neg\_mean\_squared\_error", cv=10)

# Plot model, true function, and samples

X\_test = np.linspace(0, 1, 100)

plt.plot(X\_test, model.predict(X\_test[:, np.newaxis]), label="Model")

plt.plot(X\_test, true\_fun(X\_test), label="True function")

plt.scatter(X, y, edgecolor="b", s=20, label="Samples")

plt.xlabel("x")

plt.ylabel("y")

plt.xlim((0, 1))

plt.ylim((-2, 2))

plt.legend(loc="best")

plt.title(f"Degree {degree}\nMSE = {-scores.mean():.2e}(+/- {scores.std():.2e})")

plt.show()

**OUTPUT:**

****

**EXPERIMENT:6**

**PROGRAM:** logistic regression

import numpy as np

import matplotlib.pyplot as plt

def sigmoid(z):

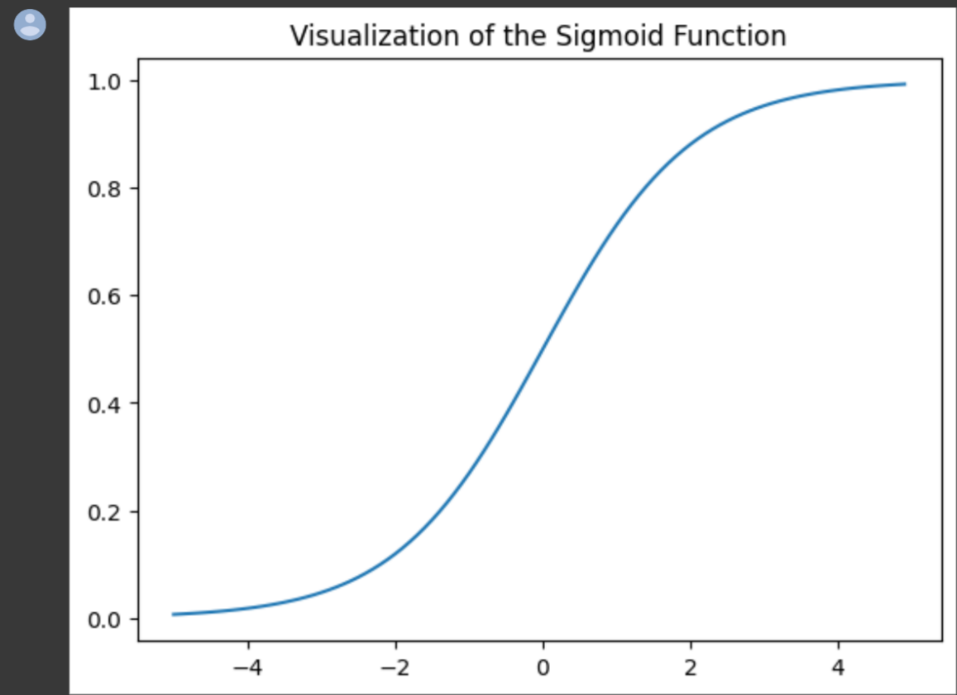
    return 1 / (1 + np.exp( - z))

plt.plot(np.arange(-5, 5, 0.1), sigmoid(np.arange(-5, 5, 0.1)))

plt.title('Visualization of the Sigmoid Function')

plt.show()

**OUTPUT:**

****

**EXPERIMENT:7**

**PROGRAM:** KNN algorithm

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.naive\_bayes import GaussianNB

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

# Load the iris dataset

iris = load\_iris()

X = iris.data

y = iris.target

# Split dataset into training and test sets

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

# Feature scaling

sc = StandardScaler()

X\_train = sc.fit\_transform(X\_train)

X\_test = sc.transform(X\_test)

# Train Naive Bayes model

classifier = GaussianNB()

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

# Predict and evaluate

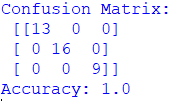
y\_pred = classifier.predict(X\_test)

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

print("Confusion Matrix:\n", cm)

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

**OUTPUT:**



**EXPERIMENT:8**

**PROGRAM:** NAVIE BAYES ALGORITHM

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.naive\_bayes import GaussianNB

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

# Load the iris dataset

iris = load\_iris()

X = iris.data

y = iris.target

# Split dataset into training and test sets

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

# Feature scaling

sc = StandardScaler()

X\_train = sc.fit\_transform(X\_train)

X\_test = sc.transform(X\_test)

# Train Naive Bayes model

classifier = GaussianNB()

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

# Predict and evaluate

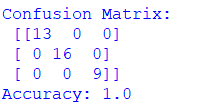
y\_pred = classifier.predict(X\_test)

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

print("Confusion Matrix:\n", cm)

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

**OUTPUT:**



**EXPERIMENT:9**

**PROGRAM:** LOGISTIC REGRESSION

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.metrics import confusion\_matrix, accuracy\_score

# Load the iris dataset

iris = load\_iris()

X = iris.data

y = iris.target

# Split the dataset into training and test sets

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.30, random\_state=2)

# Feature scaling

sc = StandardScaler()

X\_train = sc.fit\_transform(X\_train)

X\_test = sc.transform(X\_test)

# Train Logistic Regression model

classifier = LogisticRegression(random\_state=0)

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

# Predict and evaluate

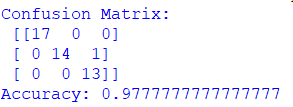
y\_pred = classifier.predict(X\_test)

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

print("Confusion Matrix:\n", cm)

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

**OUTPUT:**

****

**EXPERIMENT:10**

**PROGRAM:** DECISION TREEALGORITHM

import numpy as np

import pandas as pd

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

iris = load\_iris() X = iris.data y = iris.

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

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size = 0.25, random\_state = 8)

# Feature Scaling

from sklearn.preprocessing import StandardScaler

sc = StandardScaler()

X\_train = sc.fit\_transform(X\_train)

X\_test = sc.transform(X\_test)

# Training the Decision Tree Classification model on the Training set

from sklearn.tree import DecisionTreeClassifier

classifier = DecisionTreeClassifier(criterion = 'entropy', random\_state = 5)

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

# Display the Decision Tree

from sklearn.tree import plot\_tree

import matplotlib.pyplot as plt

plt.figure(figsize=(20,10))

plot\_tree(classifier, filled=True, rounded=True, feature\_names=dataset.columns[:-1])

plt.show()

# Predicting the Test set results

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

# Display the results (confusion matrix and accuracy)

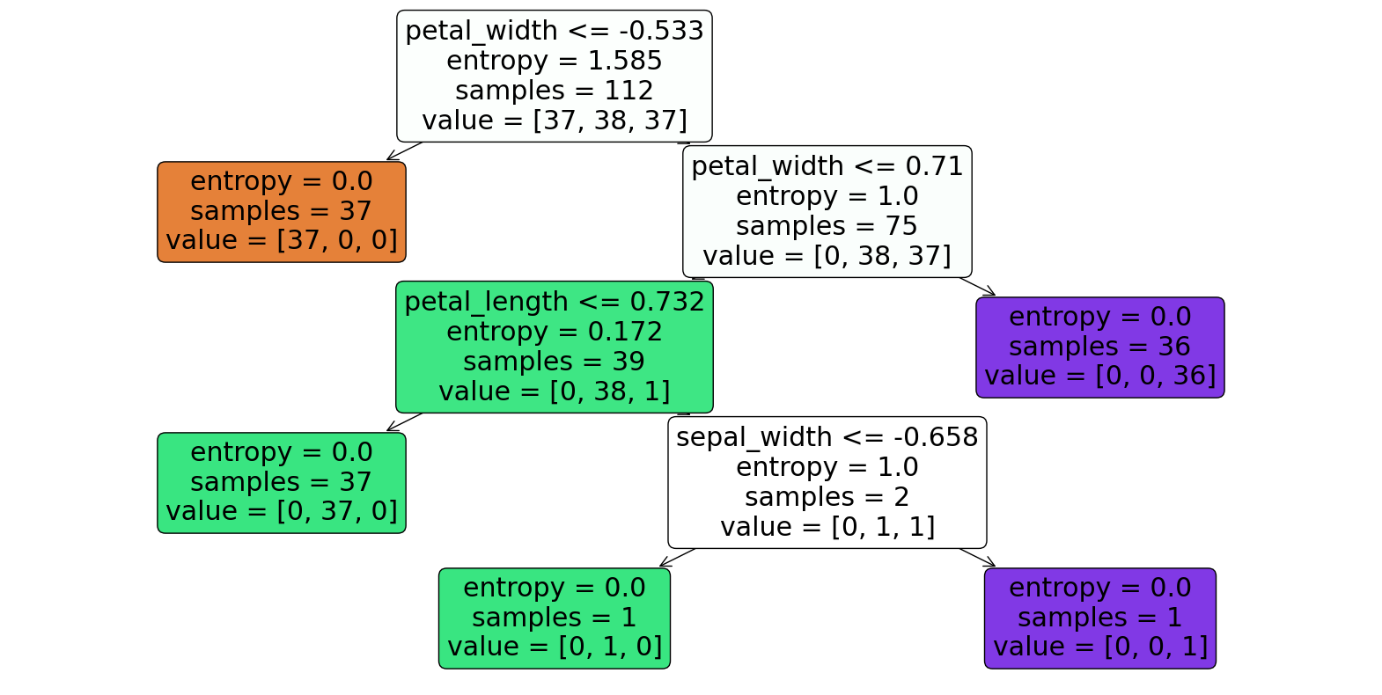
from sklearn.metrics import confusion\_matrix, accuracy\_score

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

print(cm)

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

**output:**

****

**EXPERIMENT:11**

**PROGRAM:** SVM

import numpy as np

from sklearn.datasets import load\_iris

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

from sklearn.preprocessing import StandardScaler

from sklearn.svm import SVC

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

# Load the iris dataset

iris = load\_iris()

X = iris.data

y = iris.target

# Split the dataset into training and test sets

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.25, random\_state=32)

# Feature scaling

sc = StandardScaler()

X\_train = sc.fit\_transform(X\_train)

X\_test = sc.transform(X\_test)

# Train the SVM model

classifier = SVC(kernel='linear', random\_state=0)

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

# Predict and evaluate

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

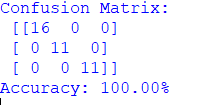
cm = confusion\_matrix(y\_test, y\_pred)

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

print("Confusion Matrix:\n", cm)

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

**OUTPUT:**

****

**EXPERIMENT:12**

**PROGRAM:**  RANDOM FOREST

import numpy as np

from sklearn.datasets import load\_iris

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

from sklearn.preprocessing import StandardScaler

from sklearn.ensemble import RandomForestClassifier

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

# Load the Iris dataset

iris = load\_iris()

X = iris.data

y = iris.target

# Split dataset into training and test sets

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.25, random\_state=39)

# Feature scaling

sc = StandardScaler()

X\_train = sc.fit\_transform(X\_train)

X\_test = sc.transform(X\_test)

# Train Random Forest model

classifier = RandomForestClassifier(n\_estimators=100, random\_state=42)

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

# Predict and evaluate

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

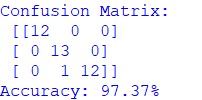
cm = confusion\_matrix(y\_test, y\_pred)

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

print("Confusion Matrix:\n", cm)

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

**OUTPUT:**

****

**EXPERIMENT:13**

**PROGRAM:** gradient descent

import numpy as np

import matplotlib.pyplot as plt

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

return np.mean((y\_true - y\_pred) \*\* 2)

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

weight, bias = 0.1, 0.01

n = len(x)

costs, weights = [], []

previous\_cost = None

for i in range(iterations):

y\_pred = weight \* x + bias

cost = mean\_squared\_error(y, y\_pred)

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

break

previous\_cost = cost

costs.append(cost)

weights.append(weight)

weight -= learning\_rate \* (-2/n \* np.sum(x \* (y - y\_pred)))

bias -= learning\_rate \* (-2/n \* np.sum(y - y\_pred))

if (i+1) % 100 == 0:

print(f"Iteration {i+1}: Cost {cost}, Weight {weight}, Bias {bias}")

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

plt.plot(weights, costs)

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

plt.title("Cost vs Weights")

plt.xlabel("Weight")

plt.ylabel("Cost")

plt.show()

return weight, bias

def main():

X = np.array([32.5, 53.4, 61.5, 47.5, 59.8, 55.1, 52.2, 39.3, 48.1, 52.6, 45.4, 54.4, 44.2, 58.2, 56.7, 49.0, 44.7, 60.3, 45.6, 38.8])

Y = np.array([31.7, 68.8, 62.6, 71.5, 87.2, 78.2, 79.6, 59.2, 75.3, 71.3, 55.2, 82.5, 62.0, 75.4, 81.4, 60.7, 82.9, 97.4, 48.8, 56.9])

weight, bias = gradient\_descent(X, Y, iterations=2000)

print(f"Estimated Weight: {weight}\nEstimated Bias: {bias}")

Y\_pred = weight \* X + bias

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

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

plt.plot(X, Y\_pred, color='blue', linestyle='--')

plt.xlabel("X")

plt.ylabel("Y")

plt.show()

if \_\_name\_\_ == "\_\_main\_\_":

main()

**output:**



**Experiment:14**

**PROGRAM:** gradient descent

import numpy as np

import matplotlib.pyplot as plt

# Function to calculate Mean Squared Error

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

return np.mean((y\_true - y\_pred) \*\* 2)

# Gradient Descent Function

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

weight, bias = 0.1, 0.01 # Initialize weight and bias

n = len(x) # Number of data points

for i in range(iterations):

y\_pred = weight \* x + bias # Calculate predictions

cost = mean\_squared\_error(y, y\_pred) # Calculate cost

# Check stopping condition

if i > 0 and abs(prev\_cost - cost) <= stopping\_threshold:

break

prev\_cost = cost

# Calculate gradients

weight\_gradient = -(2 / n) \* np.sum(x \* (y - y\_pred))

bias\_gradient = -(2 / n) \* np.sum(y - y\_pred)

# Update parameters

weight -= learning\_rate \* weight\_gradient

bias -= learning\_rate \* bias\_gradient

# Print every 1000th iteration

if i % 1000 == 0:

print(f"Iteration {i}: Cost {cost}, Weight {weight}, Bias {bias}")

# Plot cost vs weight

plt.plot(range(i+1), [mean\_squared\_error(y, weight \* x + bias) for \_ in range(i+1)])

plt.xlabel("Iteration")

plt.ylabel("Cost")

plt.show()

return weight, bias

def main():

# Sample data

X = np.array([52.5, 63.4, 81.5, 47.5, 89.8, 55.1, 52.2, 39.3, 48.1, 52.5, 45.4, 54.3, 44.2, 58.2, 56.7, 48.9, 44.7, 60.3, 45.6, 38.8])

Y = np.array([41.7, 78.8, 82.6, 91.5, 77.2, 78.2, 79.6, 59.2, 75.3, 71.3, 55.2, 82.5, 62.0, 75.4, 81.4, 60.7, 82.9, 97.4, 48.8, 56.9])

# Run gradient descent

weight, bias = gradient\_descent(X, Y)

print(f"Estimated Weight: {weight}, Estimated Bias: {bias}")

# Plot regression line

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

plt.plot(X, weight \* X + bias, color='blue', linestyle='dashed')

plt.xlabel("X")

plt.ylabel("Y")

plt.show()

if \_\_name\_\_ == "\_\_main\_\_":

main()

**OUTPUT:**





**EXPERIMENT:15**

**PROGRAM:** image processing

import numpy as np

import cv2

from matplotlib import pyplot as plt

# Load the image

img = cv2.imread('C33P1thinF\_IMG\_20150619\_114756a\_cell\_181.png')

b, g, r = cv2.split(img)

rgb\_img = cv2.merge([r, g, b])

# Convert to grayscale

gray = cv2.cvtColor(img, cv2.COLOR\_BGR2GRAY)

# Apply Otsu's thresholding

\_, thresh = cv2.threshold(gray, 0, 255, cv2.THRESH\_BINARY\_INV + cv2.THRESH\_OTSU)

# Noise removal using morphological closing

kernel = np.ones((2, 2), np.uint8)

closing = cv2.morphologyEx(thresh, cv2.MORPH\_CLOSE, kernel, iterations=2)

# Sure background area

sure\_bg = cv2.dilate(closing, kernel, iterations=3)

# Finding sure foreground area

dist\_transform = cv2.distanceTransform(sure\_bg, cv2.DIST\_L2, 3)

\_, sure\_fg = cv2.threshold(dist\_transform, 0.1 \* dist\_transform.max(), 255, 0)

# Finding unknown region

sure\_fg = np.uint8(sure\_fg)

unknown = cv2.subtract(sure\_bg, sure\_fg)

# Marker labelling

\_, markers = cv2.connectedComponents(sure\_fg)

markers = markers + 1 # Increment all labels so background is 1

markers[unknown == 255] = 0 # Mark unknown regions as 0

# Apply watershed algorithm

markers = cv2.watershed(img, markers)

img[markers == -1] = [255, 0, 0] # Mark boundaries in red

# Plotting the results

plt.subplot(211), plt.imshow(rgb\_img)

plt.title('Input Image'), plt.xticks([]), plt.yticks([])

plt.subplot(212), plt.imshow(thresh, 'gray')

plt.title("Otsu's binary threshold"), plt.xticks([]), plt.yticks([])

plt.tight\_layout()

plt.show()

**OUTPUT:**



**EXPERIMENT:16**

**PROGRAM:** image processing by using water shed database

import numpy as np

import cv2

from matplotlib import pyplot as plt

# Load the image

img = cv2.imread('C33P1thinF\_IMG\_20150619\_114756a\_cell\_181.png')

b, g, r = cv2.split(img)

rgb\_img = cv2.merge([r, g, b])

# Convert to grayscale

gray = cv2.cvtColor(img, cv2.COLOR\_BGR2GRAY)

# Apply Otsu's thresholding

\_, thresh = cv2.threshold(gray, 0, 255, cv2.THRESH\_BINARY\_INV + cv2.THRESH\_OTSU)

# Noise removal using morphological closing

kernel = np.ones((2, 2), np.uint8)

closing = cv2.morphologyEx(thresh, cv2.MORPH\_CLOSE, kernel, iterations=2)

# Sure background area using dilation

sure\_bg = cv2.dilate(closing, kernel, iterations=3)

# Plot the results

plt.subplot(211), plt.imshow(closing, 'gray')

plt.title("MorphologyEx: Closing (2x2)"), plt.xticks([]), plt.yticks([])

plt.subplot(212), plt.imshow(sure\_bg, 'gray')

plt.title("Dilation"), plt.xticks([]), plt.yticks([])

# Save the dilated image

plt.imsave('dilation.png', sure\_bg)

plt.tight\_layout()

plt.show()

**OUTPUT:**

****

**EXPERIMENT:17**

**PROGRAM:**TANH ACTIVATION

import numpy as np

import matplotlib.pyplot as plt

# Define the Tanh activation function

def tanh(x):

return np.tanh(x)

# Define the derivative of the Tanh function

def tanh\_derivative(x):

return 1 - np.tanh(x) \*\* 2

# Generate an array of input values

x = np.linspace(-5, 5, 100)

# Compute the Tanh activation values and their derivatives

y\_tanh = tanh(x)

y\_tanh\_deriv = tanh\_derivative(x)

# Plot the Tanh activation function

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

plt.plot(x, y\_tanh, label='tanh(x)', color='blue')

plt.plot(x, y\_tanh\_deriv, label="tanh'(x)", color='red', linestyle='dashed')

plt.title('Tanh Activation Function and its Derivative')

plt.xlabel('Input')

plt.ylabel('Output')

plt.axhline(0, color='black', linewidth=0.5)

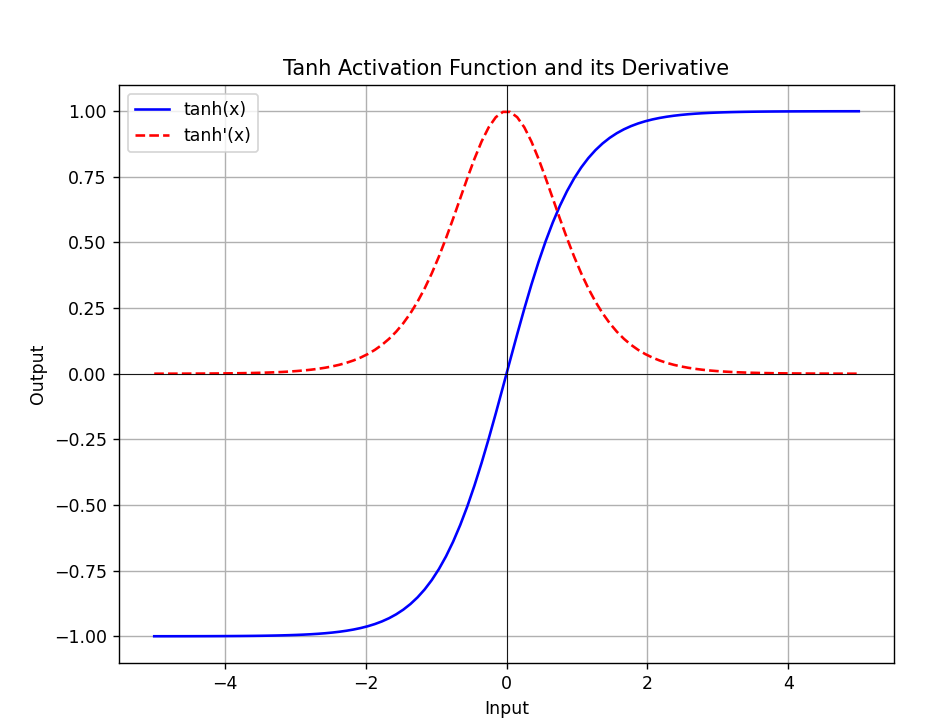
plt.axvline(0, color='black', linewidth=0.5)

plt.legend()

plt.grid(True)

plt.show()

**OUTPUT:**

****

**EXPERIMENT:18**

**PROGRAM:** SIGMOID ACTIVATION

**OUTPUT: **

**EXPERIMENT:19**

**PROGRAM:** LINEAR ACTIVATION

**OUTPUT: **

**EXPERIMENT:20**

**PROGRAM:** Neural network analysis using ReLU ACTIVATION

**OUTPUT:**



**EXPERIMENT:21**

**PROGRAM:** linear separability

import numpy as np

import matplotlib.pyplot as plt

def estimate\_coef(x, y):

n = np.size(x)

m\_x, m\_y = np.mean(x), np.mean(y)

SS\_xy = np.sum(y\*x) - n\*m\_y\*m\_x

SS\_xx = np.sum(x\*x) - n\*m\_x\*m\_x

b\_1 = SS\_xy / SS\_xx

b\_0 = m\_y - b\_1\*m\_x

return b\_0, b\_1

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

plt.scatter(x, y, color="r", marker="o")

plt.plot(x, b[0] + b[1]\*x, color="b")

plt.xlabel('x')

plt.ylabel('y')

plt.show()

x = np.array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

y = np.array([1, 3, 2, 5, 7, 8, 8, 9, 10, 12])

b = estimate\_coef(x, y)

print(f"Estimated coefficients:\nb\_0 = {b[0]} \nb\_1 = {b[1]}")

plot\_regression\_line(x, y, b)

**OUTPUT:**

