NAME:PUGAZHENDHI S

REG.NO:192011204

EXPERIMENT 1:

AIM:

To verify linear regression .

Program:

import numpy as np

import matplotlib.pyplot as plt

def estimate\_coef(x, y):

    # number of observations/points

    n = np.size(x)

    # mean of x and y vector

    m\_x = np.mean(x)

    m\_y = np.mean(y)

    # calculating cross-deviation and deviation about x

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

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

    # calculating regression coefficients

    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):

    # plotting the actual points as scatter plot

    plt.scatter(x, y, color = "r",

               marker = "o", s = 30)

    # predicted response vector

    y\_pred = b[0] + b[1]\*x

    # plotting the regression line

    plt.plot(x, y\_pred, color = "b")

    # putting labels

    plt.xlabel('x')

    plt.ylabel('y')

    # function to show plot

    plt.show()

def main():

    # observations / data

    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])

    # estimating coefficients

    b = estimate\_coef(x, y)

    print("Estimated coefficients:\nb\_0 = {}  \

          \nb\_1 = {}".format(b[0], b[1]))

    # plotting regression line

    plot\_regression\_line(x, y, b)

if \_\_name\_\_ == "\_\_main\_\_":

    main()

Result:

Conclusion:

The Linear Regression problem is verified and also necessary outcome is recorded

EXPERIMENT 2:

AIM:

To verify Logistic Regression

Program:

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()

Result:

Conclusion:

The Logistic Regression problem is verified and also necessary outcome is recorded

EXPERIMENT 3:

AIM:

To verify 2-Confusion Matrix

Program:

#Import the necessary libraries

import numpy as np

from sklearn.metrics import confusion\_matrix

import seaborn as sns

import matplotlib.pyplot as plt

#Create the NumPy array for actual and predicted labels.

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'])

#compute the confusion matrix.

cm = confusion\_matrix(actual,predicted)

#Plot the confusion matrix.

sns.heatmap(cm,

            annot=True,

            fmt='g',

            xticklabels=['Dog','Not Dog'],

            yticklabels=['Dog','Not Dog'])

plt.ylabel('Prediction',fontsize=13)

plt.xlabel('Actual',fontsize=13)

plt.title('Confusion Matrix',fontsize=17)

plt.show()

Result:

Conclusion:

The Confusion Matrix problem is verified and also necessary outcome is recorded

EXPERIMENT 4:

AIM:

To verify multi-class confusion matrix.

Program:

#Import the necessary libraries

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

import seaborn as sns

import matplotlib.pyplot as plt

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

# Load the breast cancer 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)

# preduction

y\_pred = clf.predict(X\_test)

# compute the confusion matrix

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

#Plot the confusion matrix.

sns.heatmap(cm,

            annot=True,

            fmt='g')

plt.ylabel('Prediction',fontsize=13)

plt.xlabel('Actual',fontsize=13)

plt.title('Confusion Matrix',fontsize=17)

plt.show()

# Finding precision and recall

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

print("Accuracy   :", accuracy)

Result:

Conclusion:

The multi-class confusion matrix problem is verified and also necessary outcome is recorded

EXPERIMENT 5:

AIM:

To verify OVERFITTING

Program:

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 in range(len(degrees)):

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

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

    polynomial\_features = PolynomialFeatures(degree=degrees[i], include\_bias=False)

    linear\_regression = LinearRegression()

    pipeline = Pipeline(

        [

            ("polynomial\_features", polynomial\_features),

            ("linear\_regression", linear\_regression),

        ]

    )

    pipeline.fit(X[:, np.newaxis], y)

    # Evaluate the models using crossvalidation

    scores = cross\_val\_score(

        pipeline, X[:, np.newaxis], y, scoring="neg\_mean\_squared\_error", cv=10

    )

    X\_test = np.linspace(0, 1, 100)

    plt.plot(X\_test, pipeline.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(

        "Degree {}\nMSE = {:.2e}(+/- {:.2e})".format(

            degrees[i], -scores.mean(), scores.std()

        )

    )

plt.show()

Result:

Conclusion:

The OVERFITTING problem is verified and also necessary outcome is recorded

EXPERIMENT 6(1):

AIM:

To verify K-Nearest Neighbors.

Program:

import numpy as np

import pandas as pd

dataset = pd.read\_csv("D:/GEO/BE COURSES/LAB/DATASET/knn1 (1).csv")

"""

The breast cancer dataset has the following features: Sample code number, Clump Thickness, Uniformity of Cell Size,

Uniformity of Cell Shape, Marginal Adhesion, Single Epithelial Cell Size, Bare Nuclei, Bland Chromatin,

Normal Nucleoli, Mitosis, Class.

"""

X = dataset.iloc[:, :-1].values

y = dataset.iloc[:, -1].values

dataset.shape

#splitting the dataset into the Training set and Test set

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

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size = 0.20, random\_state = 42)

#Feature Scaling

"""

Feature scaling is the process of converting the data into a given range.

In this case, the standard scalar technique is used.

"""

from sklearn.preprocessing import StandardScaler

sc = StandardScaler()

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

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

#Training the K-Nearest Neighbors (K-NN) Classification model on the Training set

"""

Once the dataset is scaled, next, the K-Nearest Neighbors (K-NN) classifier algorithm is used to create a model.

The hyperparameters such as n\_neighbors, metric, and p are set to 5, Minkowski, and 2 respectively.

The remaining hyperparameters are set to default values.

"""

from sklearn.neighbors import KNeighborsClassifier

classifier = KNeighborsClassifier(n\_neighbors = 5, metric = 'minkowski', p = 2)

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

"""

Display the results (confusion matrix and accuracy)

Here evaluation metrics such as confusion matrix and accuracy are used to evaluate the performance of the model built using a decision tree classifier.

"""

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

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

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

print(cm)

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

Result:

Conclusion:

The K-Nearest Neighborsproblem is verified and also necessary outcome is recorded

EXPERIMENT 6(2):

AIM:

To verify implement Linear Regression

Program:

import numpy as np

import matplotlib.pyplot as plt

import pandas as pd

dataset = pd.read\_csv('D:/GEO/BE COURSES/LAB/DATASET/Salary\_Data.csv')

X = dataset.iloc[:, :-1].values

y = dataset.iloc[:, -1].values

dataset.head()

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

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size = 1/3, random\_state = 0)

from sklearn.linear\_model import LinearRegression

regressor = LinearRegression()

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

LinearRegression(copy\_X=True, fit\_intercept=True, n\_jobs=None, normalize=False)

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

pd.DataFrame(data={'Actuals': y\_test, 'Predictions': y\_pred})

#Visualising the Training set results Here scatter plot is used to visualize the results.

plt.scatter(X\_train, y\_train, color = 'red')

plt.plot(X\_train, regressor.predict(X\_train), color = 'blue')

plt.title('Salary vs Experience (Training set)')

plt.xlabel('Years of Experience')

plt.ylabel('Salary')

plt.show()

Result:

Conclusion:

The implement Linear Regression problem is verified and also necessary outcome is recorded

EXPERIMENT 6(3):

AIM:

To verify Logistic Regression

Program:

import numpy as np

import pandas as pd

#"Importing the dataset

"""

After importing the necessary libraries, next, we import or read the dataset.

Click here to download the breast cancer dataset used in this implementation.

The breast cancer dataset has the following features:

Sample code number, Clump Thickness, Uniformity of Cell Size, Uniformity of Cell Shape, Marginal Adhesion,

Single Epithelial Cell Size, Bare Nuclei, Bland Chromatin, Normal Nucleoli, Mitosis, Class.

"""

# divide the dataset into concepts and targets. Store the concepts into X and targets into y.

dataset = pd.read\_csv("D:/GEO/BE COURSES/LAB/DATASET/breastcancer.csv")

X = dataset.iloc[:, :-1].values

y = dataset.iloc[:, -1].values

#Splitting the dataset into the Training set and Test

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

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

#Feature Scaling

"""

Feature scaling is the process of converting the data into a given range. In this case, the standard scalar technique is used.

from sklearn.preprocessing import StandardScaler

"""

from sklearn.preprocessing import StandardScaler

sc = StandardScaler()

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

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

"""

Training the Logistic Regression (LR) Classification model on the Training set

Once the dataset is scaled, next, the Logistic Regression (LR) classifier algorithm is used to create a model.

The hyperparameters such as random\_state to 0 respectively.

The remaining hyperparameters Logistic Regression (LR) are set to default values.

"""

from sklearn.linear\_model import LogisticRegression

classifier = LogisticRegression(random\_state = 0)

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

#Logistic Regression (LR) classifier model

LogisticRegression(C=1.0, class\_weight=None, dual=False, fit\_intercept=True,

intercept\_scaling=1, l1\_ratio=None, max\_iter=100,

multi\_class='warn', n\_jobs=None, penalty='l2',

random\_state=0, solver='warn', tol=0.0001, verbose=0,

warm\_start=False)

#Display the results (confusion matrix and accuracy)

"""

Here evaluation metrics such as confusion matrix and accuracy are used to evaluate the performance of the model

built using a decision tree classifier.

"""

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

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

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

print(cm)

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

Result:

Conclusion:

The Logistic Regression problem is verified and also necessary outcome is recorded

EXPERIMENT 6(4):

AIM:

To verify svm(breast cancer detection)

Program:

import numpy as np

import pandas as pd

# Importing the dataset

dataset = pd.read\_csv('/content/breastcancer.csv')

X = dataset.iloc[:, :-1].values

y = dataset.iloc[:, -1].values

# Splitting the dataset into the Training set and Test set

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

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

# Feature Scaling

from sklearn.preprocessing import StandardScaler

sc = StandardScaler()

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

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

# Training the SVM model on the Training set

from sklearn.svm import SVC

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

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

# Predicting the Test set results

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

# Evaluating the performance of the model using confusion matrix and accuracy

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

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

print(cm)

print('Accuracy: {:.2f}%'.format(accuracy\_score(y\_test, y\_pred) \* 100))

Result:

Conclusion:

The svm(breast cancer detection) problem is verified and also necessary outcome is recorded

EXPERIMENT 6(5):

AIM:

To verify Naïve Bayes algorithm

Program:

import numpy as np

import pandas as pd

#Importing the dataset

"""

Next, we import or read the dataset. Click here to download the breast cancer dataset used in this implementation.

After reading the dataset, divide the dataset into concepts and targets. Store the concepts into X and

targets into y.

"""

dataset = pd.read\_csv("D:/GEO/BE COURSES/LAB/DATASET/breastcancer.csv")

X = dataset.iloc[:, :-1].values

y = dataset.iloc[:, -1].values

"""

Splitting the dataset into the Training set and Test set

Once the dataset is read into the memory, next, divide the dataset into two parts, training and

testing using the train\_test\_split function from sklearn.

The test\_size and random\_state attributes are set to 0.25 and 0 respectively.

You can change these attributes as per your requirements.

"""

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

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

#Feature Scaling

"""

Feature scaling is the process of converting the data into a min-max range. In this case,

the standard scalar method is used.

"""

from sklearn.preprocessing import StandardScaler

sc = StandardScaler()

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

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

"""

Training the Naive Bayes Classification model on the Training set

Once the dataset is scaled, next, the Naive Bayes classifier algorithm is used to create a model.

The GaussianNB function is imported from sklearn.naive\_bayes library. The hyperparameters such as kernel,

and random\_state to linear, and 0 respectively. The remaining hyperparameters of the support vector machine

algorithm are set to default values.

"""

from sklearn.naive\_bayes import GaussianNB

classifier = GaussianNB()

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

#Naive Bayes classifier model

GaussianNB(priors=None, var\_smoothing=1e-09)

#Display the results (confusion matrix and accuracy)

"""

Here evaluation metrics such as confusion matrix and accuracy are used to evaluate the performance of

the model built using a decision tree classifier.

"""

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

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

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

print(cm)

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

Result:

Conclusion:

The Naïve Bayes algorithm problem is verified and also necessary outcome is recorded

EXPERIMENT 6(6):

AIM:

To verify random first.

Program:

Result:

Conclusion:

The random first problem is verified and also necessary outcome is recorded

EXPERIMENT 6(7):

AIM:

To verify decision tree

Program:

Result:

Conclusion:

The decision tree problem is verified and also necessary outcome is recorded

EXPERIMENT 7(1):

AIM:

To verify gradiant descent(actual data)

Program:

# 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()

Result:

Conclusion:

The gradiant descent problem is verified and also necessary outcome is recorded

EXPERIMENT 7(2):

AIM:

To verify gradiant descent(modified data)

Program:

# 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([31.70700585, 68.77759598, 62.5623823 , 71.54663223, 87.23092513,55.14218841, 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([

           32.50234527, 53.42680403, 61.53035803, 47.47563963, 59.81320787,78.21151827, 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()

Result:

Conclusion:

The gradiant descent (modified data)problem is verified and also necessary outcome is recorded

EXPERIMENT 8:

AIM:

To verify Image Segmentation Algorithms

Program:

# SEGMENTATION

import numpy as np

import cv2

from matplotlib import pyplot as plt

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

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

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

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

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

# noise removal

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

#opening = cv2.morphologyEx(thresh,cv2.MORPH\_OPEN,kernel, iterations = 2)

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)

# Threshold

ret, 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

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

# Add one to all labels so that sure background is not 0, but 1

markers = markers+1

# Now, mark the region of unknown with zero

markers[unknown==255] = 0

markers = cv2.watershed(img,markers)

img[markers == -1] = [255,0,0]

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

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

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

plt.imsave(r'thresh.png',thresh)

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

plt.tight\_layout()

plt.show()

Result:

Conclusion:

The Image Segmentation Algorithms problem is verified and also necessary outcome is recorded

EXPERIMENT 9:

AIM:

To verify Neural network analysis

Program:

Result:

Conclusion:

The Neural network analysis problem is verified and also necessary outcome is recorded

EXPERIMENT 10:

AIM:

To verify linear seperability

Program:

Result:

Conclusion:

The linear seperability problem is verified and also necessary outcome is recorded

EXPERIMENT 11(1):

AIM:

To verify hyper-parameter tuning(knn)

Program:

import numpy as np

import pandas as pd

dataset = pd.read\_csv("D:/GEO/BE COURSES/LAB/DATASET/knn1 (1).csv")

"""

The breast cancer dataset has the following features: Sample code number, Clump Thickness, Uniformity of Cell Size,

Uniformity of Cell Shape, Marginal Adhesion, Single Epithelial Cell Size, Bare Nuclei, Bland Chromatin,

Normal Nucleoli, Mitosis, Class.

"""

X = dataset.iloc[:, :-1].values

y = dataset.iloc[:, -1].values

dataset.shape

#splitting the dataset into the Training set and Test set

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

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size = 0.20, random\_state = 42)

#Feature Scaling

"""

Feature scaling is the process of converting the data into a given range.

In this case, the standard scalar technique is used.

"""

from sklearn.preprocessing import StandardScaler

sc = StandardScaler()

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

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

#Training the K-Nearest Neighbors (K-NN) Classification model on the Training set

"""

Once the dataset is scaled, next, the K-Nearest Neighbors (K-NN) classifier algorithm is used to create a model.

The hyperparameters such as n\_neighbors, metric, and p are set to 5, Minkowski, and 2 respectively.

The remaining hyperparameters are set to default values.

"""

from sklearn.neighbors import KNeighborsClassifier

classifier = KNeighborsClassifier(n\_neighbors = 5, metric = 'minkowski', p = 2)

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

"""

Display the results (confusion matrix and accuracy)

Here evaluation metrics such as confusion matrix and accuracy are used to evaluate the performance of the model built using a decision tree classifier.

"""

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

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

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

print(cm)

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

Result:

Conclusion:

The hyper-parameter (knn)problem is verified and also necessary outcome is recorded

EXPERIMENT 11(2):

AIM:

To verify hyper-parameter(svm)

Program:

import numpy as np

import pandas as pd

# Importing the dataset

dataset = pd.read\_csv('/content/breastcancer.csv')

X = dataset.iloc[:, :-1].values

y = dataset.iloc[:, -1].values

# Splitting the dataset into the Training set and Test set

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

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

# Feature Scaling

from sklearn.preprocessing import StandardScaler

sc = StandardScaler()

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

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

# Training the SVM model on the Training set

from sklearn.svm import SVC

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

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

# Predicting the Test set results

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

# Evaluating the performance of the model using confusion matrix and accuracy

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

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

print(cm)

print('Accuracy: {:.2f}%'.format(accuracy\_score(y\_test, y\_pred) \* 100))

Result:

Conclusion:

The hyper-parameter(svm)problem is verified and also necessary outcome is recorded