SOFT COMPUTING TECHNIQUES

PRACTICAL 1

AIM: To implement Perceptron Algorithm

CODE:

```
from sklearn import datasets
import matplotlib.pyplot as plt
import numpy as np
X, y = datasets.make_blobs(n_samples=150,n_features=2,
                           centers=2,cluster_std=1.05,
                           random_state=2)
X, y = datasets.make_blobs(n_samples=150,n_features=2,
                           centers=2,cluster_std=1.05,
                           random_state=2)
fig = plt.figure(figsize=(10,8))
plt.plot(X[:, 0][y == 0], X[:, 1][y == 0], 'r^')
plt.plot(X[:, 0][y == 1], X[:, 1][y == 1], 'bs')
plt.xlabel("feature 1")
plt.ylabel("feature 2")
plt.title('Random Classification Data with 2 classes')
```


```
def step_func(z):
        return 1.0 if (z > 0) else 0.0
# looping for every example.
        for idx, x_i in enumerate(X):

            # Insering 1 fordef perceptron(X, y, lr, epochs):

    # X --> Inputs.
    # y --> labels/target.
    # lr --> learning rate.
    # epochs --> Number of iterations.

    # m-> number of training examples
```

```
    # n-> number of features
    m, n = X.shape

    # Initializing parapeters(theta) to zeros.
    # +1 in n+1 for the bias term.
    theta = np.zeros((n+1,1))

    # Empty list to store how many examples were
    # misclassified at every iteration.
    n_miss_list = []

    # Training.
    for epoch in range(epochs):

        # variable to store #misclassified.
n_miss = 0
         bias, X0 = 1.
            x_i = np.insert(x_i, 0, 1).reshape(-1,1)

            # Calculating prediction/hypothesis.
            y_hat = step_func(np.dot(x_i.T, theta))

            # Updating if the example is misclassified.
            if (np.squeeze(y_hat) - y[idx]) != 0:
                theta += lr*((y[idx] - y_hat)*x_i)

                # Incrementing by 1.
                n_miss += 1

        # Appending number of misclassified examples
        # at every iteration.
        n_miss_list.append(n_miss)

    return theta, n_miss_list
def plot_decision_boundary(X, theta):

    # X --> Inputs
    # theta --> parameters

    # The Line is y=mx+c
    # So, Equate mx+c = theta0.X0 + theta1.X1 + theta2.X2
    # Solving we find m and c
    x1 = [min(X[:,0]), max(X[:,0])]
    m = -theta[1]/theta[2]
    c = -theta[0]/theta[2]
    x2 = m*x1 + c

    # Plotting
    fig = plt.figure(figsize=(10,8))
```

```
    plt.plot(X[:, 0][y==0], X[:, 1][y==0], "r^")
    plt.plot(X[:, 0][y==1], X[:, 1][y==1], "bs")
    plt.xlabel("feature 1")
    plt.ylabel("feature 2")
    plt.title('Perceptron Algorithm')
    plt.plot(x1, x2, 'y-')
theta, miss_l = perceptron(X, y, 0.5, 100)
plot_decision_boundary(X, theta)
```

CONCLUSION: Thus implemented the experiment successfully.

PRACTICAL 2

AIM: To implement Back Propagation Algorithm

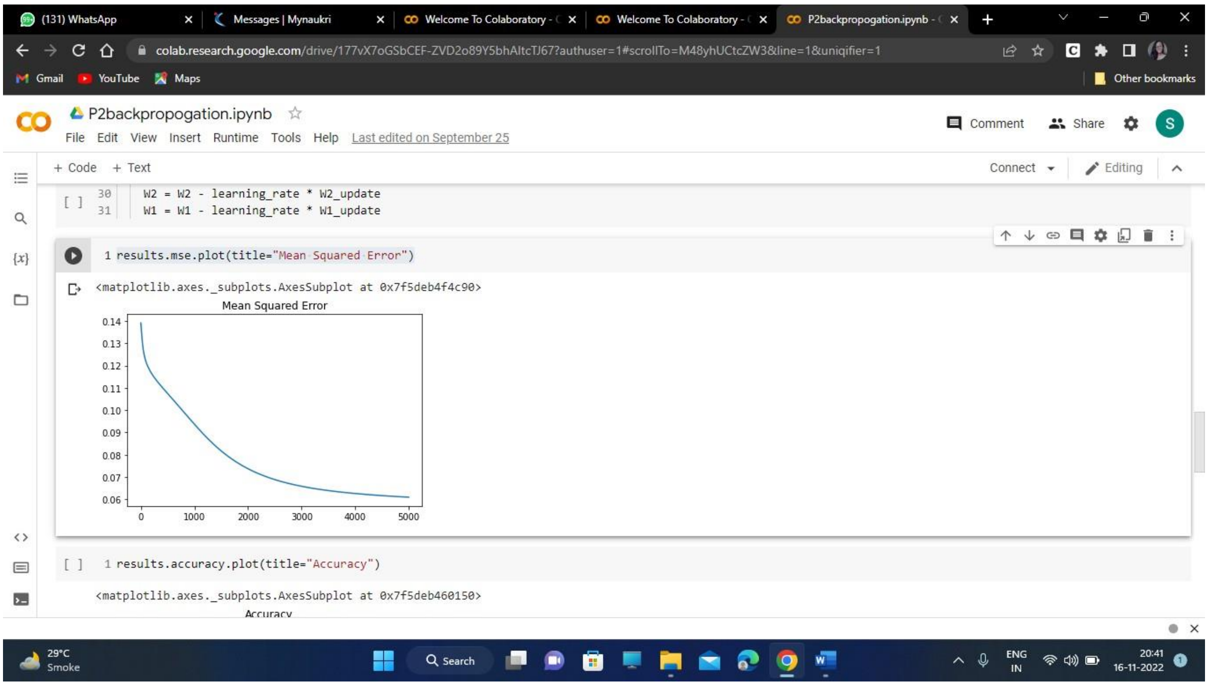
CODE:

```
# Import Libraries
import numpy as np
import pandas as pd
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt
# Load dataset
data = load_iris()

# Get features and target
X=data.data
y=data.target
# Get dummy variable
y = pd.get_dummies(y).values

y[:3]
```

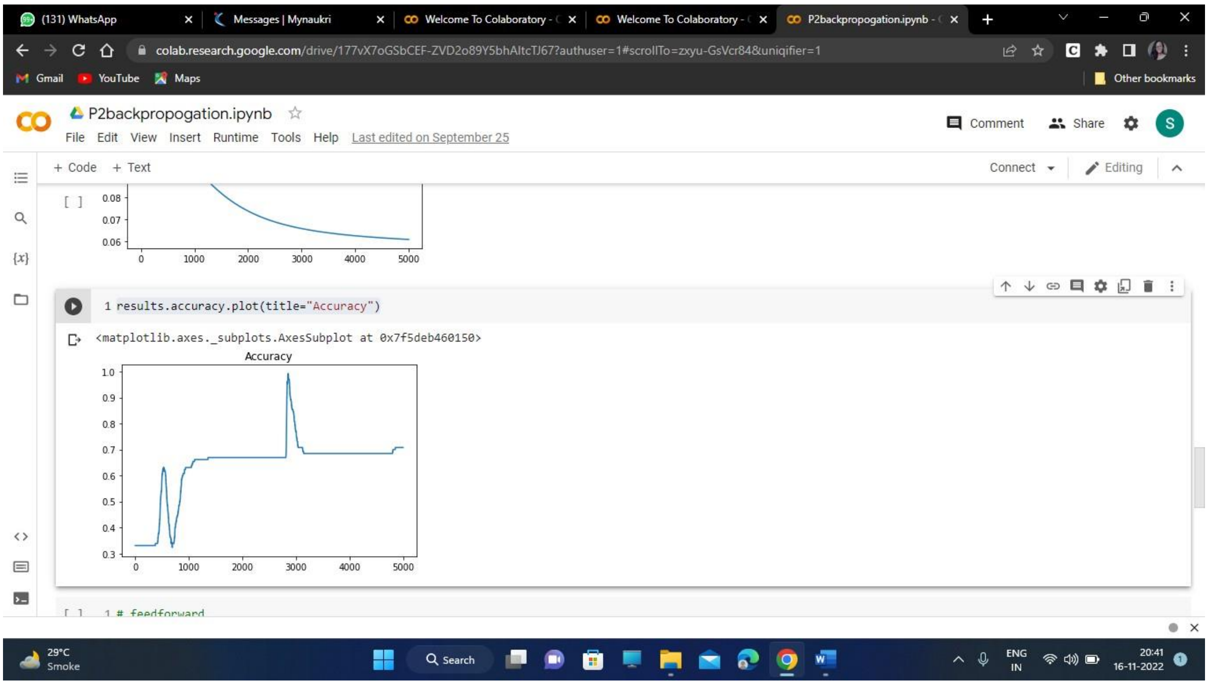
Output: array([[1, 0, 0], [1, 0, 0], [1, 0, 0]], dtype=uint8)

```
#Split data into train and test data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=20, random_state=4)
# Initialize variables
learning_rate = 0.1
iterations = 5000
N = y_train.size

# number of input features
input_size = 4

# number of hidden layers neurons
hidden_size = 2

# number of neurons at the output layer
output_size = 3

results = pd.DataFrame(columns=["mse", "accuracy"])
# Initialize weights
np.random.seed(10)

# initializing weight for the hidden layer
W1 = np.random.normal(scale=0.5, size=(input_size, hidden_size))

# initializing weight for the output layer
W2 = np.random.normal(scale=0.5, size=(hidden_size , output_size))
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def mean_squared_error(y_pred, y_true):
```

```
    return ((y_pred - y_true)**2).sum() / (2*y_pred.size)

def accuracy(y_pred, y_true):
    acc = y_pred.argmax(axis=1) == y_true.argmax(axis=1)
    return acc.mean()
for itr in range(iterations):

    # feedforward propagation
    # on hidden layer
    Z1 = np.dot(X_train, W1)
    A1 = sigmoid(Z1)

    # on output layer
    Z2 = np.dot(A1, W2)
    A2 = sigmoid(Z2)


    # Calculating error
    mse = mean_squared_error(A2, y_train)
    acc = accuracy(A2, y_train)
    results=results.append({"mse":mse, "accuracy":acc},ignore_index=True )

    # backpropagation
    E1 = A2 - y_train
    dW1 = E1 * A2 * (1 - A2)

    E2 = np.dot(dW1, W2.T)
    dW2 = E2 * A1 * (1 - A1)


    # weight updates
    W2_update = np.dot(A1.T, dW1) / N
    W1_update = np.dot(X_train.T, dW2) / N

    W2 = W2 - learning_rate * W2_update
    W1 = W1 - learning_rate * W1_update
results.mse.plot(title="Mean Squared Error")
```



results.accuracy.plot(title="Accuracy")




```
# feedforward
Z1 = np.dot(X_test, W1)
A1 = sigmoid(Z1)

Z2 = np.dot(A1, W2)
A2 = sigmoid(Z2)

acc = accuracy(A2, y_test)
print("Accuracy: {}".format(acc))
```

OUTPUT: Accuracy: 0.8

CONCLUSION: Thus implemented the experiment successfully.

PRACTICAL 3

AIM: To implement Hebb’s Rule in C++

CODE:

```
#include<iostream>
using namespace std;
int main()
{
    int m,n;
    cout<<"enter no.of features and no.of training datasets: \n";
    cin>> m>>n;
    int wt1[m], wt2[m];
    int input[n][m];
    cout<<"enter the input matrix row wise "<<endl;
    for(int i=0;i<n;i++)
    {
        for(int j=0;j<m;j++)
        {
            cin>>input[i][j];
        }
    }

    int target1[n], target2[n];
    cout<<" Enter the target in binary : "<<endl;
    for(int i=0;i<n;i++)
    {
        cin>>target1[i];
    }
  cout<<"Enter the target in bipolar: "<<endl;
    for(int i=0;i<n;i++)
    {
        cin>>target2[i];
    }
    for(int i=0;i<m;i++)
    {
        wt1[i]=0; //step 1: initialise all wts to 0
        wt2[i]=0;
    }
    for(int j=0;j<n;j++)
    {
        cout<<"##########j="<<j<<endl;
        for(int i=0; i<m;i++)
        {
            wt1[i]+= (input[j][i]*target1[j]);
            cout<<"weight1 at i="<<i<<" is "<<wt1[i]<<endl;
            wt2[i]+= (input[j][i]*target2[j]);
            cout<<"wt2 at i="<<i<<" is "<<wt2[i]<<endl;
        }
    }
    cout<<"************OUTPUT**************\nafter 1 epoch: binary weights: "<<endl;
 for (int i = 0; i < m; ++i)
    {
```

```
        /* code */
        cout<<wt1[i]<<" ";
    }
    cout<<"\nafter 1 epoch: bipolar weights: "<<endl;
    for (int i = 0; i < m; ++i)
    {
        /* code */
        cout<<wt2[i]<<" ";
    }
    return 0;
}
```

Output:

enter no.of features and no.of training datasets:
3 4
enter the input matrix row wise
-1 -1 1
-1 1 1
1 1 -1
1 1 1
Enter the target in binary :
0
0
0
1
Enter the target in bipolar:
-1
-1
-1
1
###########j=0
weight1 at i=0 is 0
wt2 at i=0 is 1
weight1 at i=1 is 0
wt2 at i=1 is 1
weight1 at i=2 is 0
wt2 at i=2 is -1
###########j=1
weight1 at i=0 is 0
wt2 at i=0 is 2
weight1 at i=1 is 0
wt2 at i=1 is 0
weight1 at i=2 is 0
wt2 at i=2 is -2
###########j=2
weight1 at i=0 is 0
wt2 at i=0 is 1
weight1 at i=1 is 0
wt2 at i=1 is -1
weight1 at i=2 is 0
wt2 at i=2 is -1
###########j=3

weight1 at i=0 is 1
wt2 at i=0 is 2
weight1 at i=1 is 1
wt2 at i=1 is 0
weight1 at i=2 is 1
wt2 at i=2 is 0
**************OUTPUT**************
after 1 epoch: binary weights:
1 1 1
after 1 epoch: bipolar weights:
2 0 0

**** Process exited - Return Code: 0 ****

CONCLUSION: Thus implemented the experiment successfully.

PRACTICAL 4

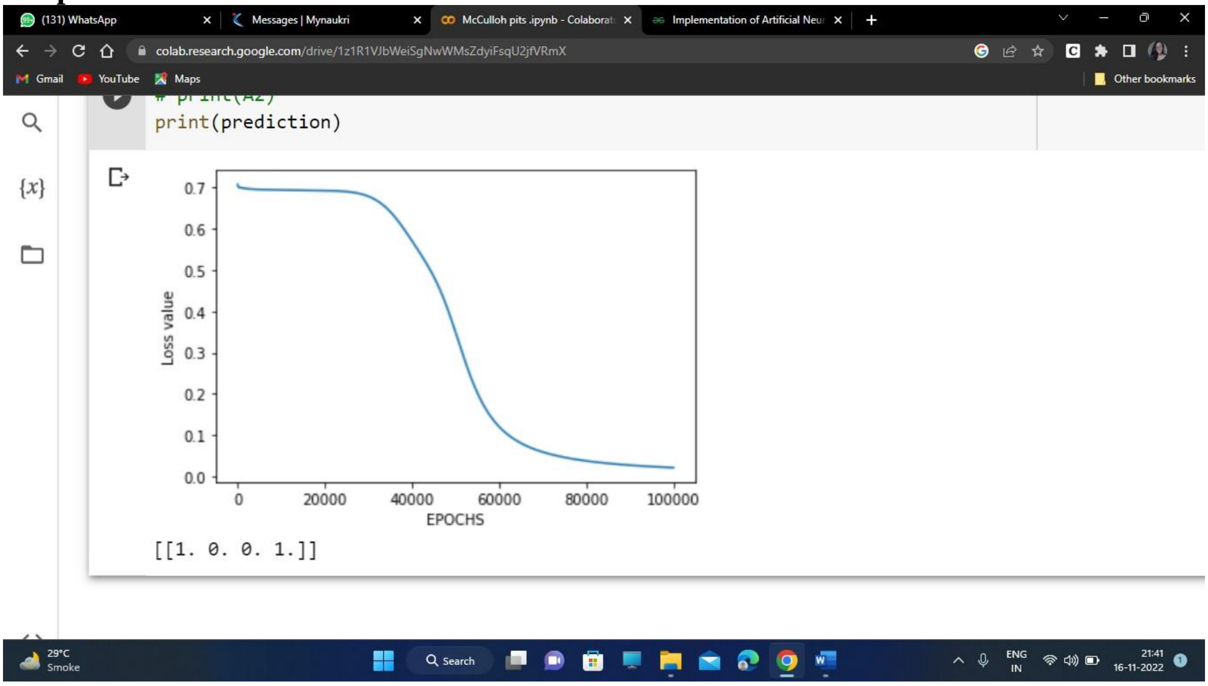
AIM: To implement McCulloh Pits model for XOR gate

CODE:

```
# import Python Libraries
import numpy as np
from matplotlib import pyplot as plt

# Sigmoid Function
def sigmoid(z):
    return 1 / (1 + np.exp(-z))

# Initialization of the neural network parameters
# Initialized all the weights in the range of between 0 and 1
# Bias values are initialized to 0
def initializeParameters(inputFeatures, neuronsInHiddenLayers, outputFeatures):
    W1 = np.random.randn(neuronsInHiddenLayers, inputFeatures)
    W2 = np.random.randn(outputFeatures, neuronsInHiddenLayers)
    b1 = np.zeros((neuronsInHiddenLayers, 1))
    b2 = np.zeros((outputFeatures, 1))

    parameters = {"W1" : W1, "b1": b1,
                  "W2" : W2, "b2": b2}
    return parameters

# Forward Propagation
def forwardPropagation(X, Y, parameters):
    m = X.shape[1]
    W1 = parameters["W1"]
    W2 = parameters["W2"]
    b1 = parameters["b1"]
    b2 = parameters["b2"]

    Z1 = np.dot(W1, X) + b1
    A1 = sigmoid(Z1)
    Z2 = np.dot(W2, A1) + b2
    A2 = sigmoid(Z2)

    cache = (Z1, A1, W1, b1, Z2, A2, W2, b2)
    logprobs = np.multiply(np.log(A2), Y) + np.multiply(np.log(1 - A2), (1 - Y))
    cost = -np.sum(logprobs) / m
    return cost, cache, A2

# Backward Propagation
def backwardPropagation(X, Y, cache):
    m = X.shape[1]
    (Z1, A1, W1, b1, Z2, A2, W2, b2) = cache

    dZ2 = A2 - Y
    dW2 = np.dot(dZ2, A1.T) / m
    db2 = np.sum(dZ2, axis = 1, keepdims = True)
```

```
    dA1 = np.dot(W2.T, dZ2)
    dZ1 = np.multiply(dA1, A1 * (1- A1))
    dW1 = np.dot(dZ1, X.T) / m
    db1 = np.sum(dZ1, axis = 1, keepdims = True) / m

    gradients = {"dZ2": dZ2, "dW2": dW2, "db2": db2,
                 "dZ1": dZ1, "dW1": dW1, "db1": db1}
    return gradients

# Updating the weights based on the negative gradients
def updateParameters(parameters, gradients, learningRate):
    parameters["W1"] = parameters["W1"] - learningRate * gradients["dW1"]
    parameters["W2"] = parameters["W2"] - learningRate * gradients["dW2"]
    parameters["b1"] = parameters["b1"] - learningRate * gradients["db1"]
    parameters["b2"] = parameters["b2"] - learningRate * gradients["db2"]
    return parameters

# Model to learn the XOR truth table
X = np.array([[0, 0, 1, 1], [0, 1, 0, 1]]) # XOR input
Y = np.array([[0, 1, 1, 0]]) # XOR output

# Define model parameters
neuronsInHiddenLayers = 2 # number of hidden layer neurons (2)
inputFeatures = X.shape[0] # number of input features (2)
outputFeatures = Y.shape[0] # number of output features (1)
parameters = initializeParameters(inputFeatures, neuronsInHiddenLayers, outputFeatures)
epoch = 100000
learningRate = 0.01
losses = np.zeros((epoch, 1))

for i in range(epoch):
    losses[i, 0], cache, A2 = forwardPropagation(X, Y, parameters)
    gradients = backwardPropagation(X, Y, cache)
    parameters = updateParameters(parameters, gradients, learningRate)

# Evaluating the performance
plt.figure()
plt.plot(losses)
plt.xlabel("EPOCHS")
plt.ylabel("Loss value")
plt.show()

# Testing
X = np.array([[1, 1, 0, 0], [0, 1, 0, 1]]) # XOR input
cost, _, A2 = forwardPropagation(X, Y, parameters)
prediction = (A2 > 0.5) * 1.0
# print(A2)
print(prediction)
```

Output:



CONCLUSION: Thus implemented the experiment successfully.

PRACTICAL 5

AIM: To implement Maxnet

CODE:

```
def winner(mylist):
    a=[]
    for i in range(0,len(mylist)):
        if mylist[i]>0:
            a.append(i)
    return a
m=0
delta=0
yin=[0]
epoch=0
f=open("maxnet_input.txt")
linenumber=0
while True:
            line=f.readline()
            line=line.rstrip('\n')
            if len(line)==0:
                break
            linenumber+=1
            word=line.split('=')
            if linenumber==1:
                m=int(word[1])
                yin=yin*m
            if linenumber==3:
                x=word[1].split(',')
                for i in range(x._len_()):
                    yin[i]=float(x[i])
            if linenumber==2:
                delta=float(word[1])
f.close();
while True:
    epoch+=1
    yout=[]
    for i in range(0,m):
        if yin[i]>=0:
            yout.append(yin[i])
        else:
            yout.append(0)
    if len(winner(yout))==1:
        print('winner unit is : ',winner(yout)[0]+1)
        break
    for i in range(0,m):
        yin[i]=yout[i]-(sum(yout)-yout[i])*delta
    if epoch==100:
        break
```

OUTPUT: winner unit is : 4

CONCLUSION: Thus implemented the experiment successfully.

PRACTICAL 6

AIM: To implement BAM using python

CODE:

```
# Import Python Libraries
import numpy as np

# Take two sets of patterns:
# Set A: Input Pattern
x1 = np.array([1, 1, 1, 1, 1, 1]).reshape(6, 1)
x2 = np.array([-1, -1, -1, -1, -1, -1]).reshape(6, 1)
x3 = np.array([1, 1, -1, -1, 1, 1]).reshape(6, 1)
x4 = np.array([-1, -1, 1, 1, -1, -1]).reshape(6, 1)

# Set B: Target Pattern
y1 = np.array([1, 1, 1]).reshape(3, 1)
y2 = np.array([-1, -1, -1]).reshape(3, 1)
y3 = np.array([1, -1, 1]).reshape(3, 1)
y4 = np.array([-1, 1, -1]).reshape(3, 1)

'''
print("Set A: Input Pattern, Set B: Target Pattern")
print("\nThe input for pattern 1 is")
print(x1)
print("\nThe target for pattern 1 is")
print(y1)
print("\nThe input for pattern 2 is")
print(x2)
print("\nThe target for pattern 2 is")
print(y2)
print("\nThe input for pattern 3 is")
print(x3)
print("\nThe target for pattern 3 is")
print(y3)
print("\nThe input for pattern 4 is")
print(x4)
print("\nThe target for pattern 4 is")
print(y4)

print("\n.........................................")
'''
# Calculate weight Matrix: W
inputSet = np.concatenate((x1, x2, x3, x4), axis = 1)
targetSet = np.concatenate((y1.T, y2.T, y3.T, y4.T), axis = 0)
print("\nWeight matrix:")
weight = np.dot(inputSet, targetSet)
print(weight)

print("\n.........................................")
```

```
# Testing Phase
# Test for Input Patterns: Set A
print("\nTesting for input patterns: Set A")
def testInputs(x, weight):
  # Multiply the input pattern with the weight matrix
  # (weight.T X x)
  y = np.dot(weight.T, x)
  y[y < 0] = -1
  y[y >= 0] = 1
  return np.array(y)

print("\nOutput of input pattern 1")
print(testInputs(x1, weight))
print("\nOutput of input pattern 2")
print(testInputs(x2, weight))
print("\nOutput of input pattern 3")
print(testInputs(x3, weight))
print("\nOutput of input pattern 4")
print(testInputs(x4, weight))

# Test for Target Patterns: Set B
print("\nTesting for target patterns: Set B")
def testTargets(y, weight):
  # Multiply the target pattern with the weight matrix
  # (weight X y)
  x = np.dot(weight, y)
  x[x <= 0] = -1
  x[x > 0] = 1
  return np.array(x)

print("\nOutput of target pattern 1")
print(testTargets(y1, weight))
print("\nOutput of target pattern 2")
print(testTargets(y2, weight))
print("\nOutput of target pattern 3")
print(testTargets(y3, weight))
print("\nOutput of target pattern 4")
print(testTargets(y4, weight))
```

Output:

```
Weight matrix:
[[4 0 4]
 [4 0 4]
 [0 4 0]
 [0 4 0]
 [4 0 4]
 [4 0 4]]

- - - - - - - - - - - - - - - - - - - -
```

Testing for input patterns: Set A

```
Output of input pattern 1
[[1]
 [1]
 [1]]

Output of input pattern 2
[[-1]
 [-1]
 [-1]]

Output of input pattern 3
[[ 1]
 [-1]
 [ 1]]

Output of input pattern 4
[[-1]
 [ 1]
 [-1]]
```

Testing for target patterns: Set B

```
Output of target pattern 1
[[1]
 [1]
 [1]
 [1]
 [1]
 [1]]

Output of target pattern 2
[[-1]
 [-1]
 [-1]
 [-1]
 [-1]
 [-1]]

Output of target pattern 3
[[ 1]
 [ 1]
 [-1]
 [-1]
 [ 1]
 [ 1]]

Output of target pattern 4
[[-1]
 [-1]
```

```
[ 1]
[ 1]
[-1]
[-1]]
```

CONCLUSION: Thus implemented the experiment successfully.

PRACTICAL 7

AIM: Find the ratios using fuzzy logic

CODE:

```
!pip install fuzzywuzzy
!pip install python-Levenshtein

from fuzzywuzzy import fuzz
from fuzzywuzzy import process

s1 = "I love fuzzysforfuzzys"
s2 = "I am loving fuzzysforfuzzys"
print ("FuzzyWuzzy Ratio:", fuzz.ratio(s1, s2))
print ("FuzzyWuzzy PartialRatio: ", fuzz.partial_ratio(s1, s2))
print ("FuzzyWuzzy TokenSortRatio: ", fuzz.token_sort_ratio(s1, s2))
print ("FuzzyWuzzy TokenSetRatio: ", fuzz.token_set_ratio(s1, s2))
print ("FuzzyWuzzy WRatio: ", fuzz.WRatio(s1, s2),'\n\n')

# for process library,
query = 'fuzzys for fuzzys'
choices = ['fuzzy for fuzzy', 'fuzzy fuzzy', 'g. for fuzzys']
print ("List of ratios: ")
print (process.extract(query, choices), '\n')
print ("Best among the above list: ",process.extractOne(query, choices))
```

OUTPUT:

```
FuzzyWuzzy Ratio: 86
FuzzyWuzzy PartialRatio:  86
FuzzyWuzzy TokenSortRatio:  86
FuzzyWuzzy TokenSetRatio:  87
FuzzyWuzzy WRatio:  86


List of ratios:
[('g. for fuzzys', 95), ('fuzzy for fuzzy', 94), ('fuzzy fuzzy', 86)]

Best among the above list:  ('g. for fuzzys', 95)
```

CONCLUSION: Thus implemented the experiment successfully.

PRACTICAL 8

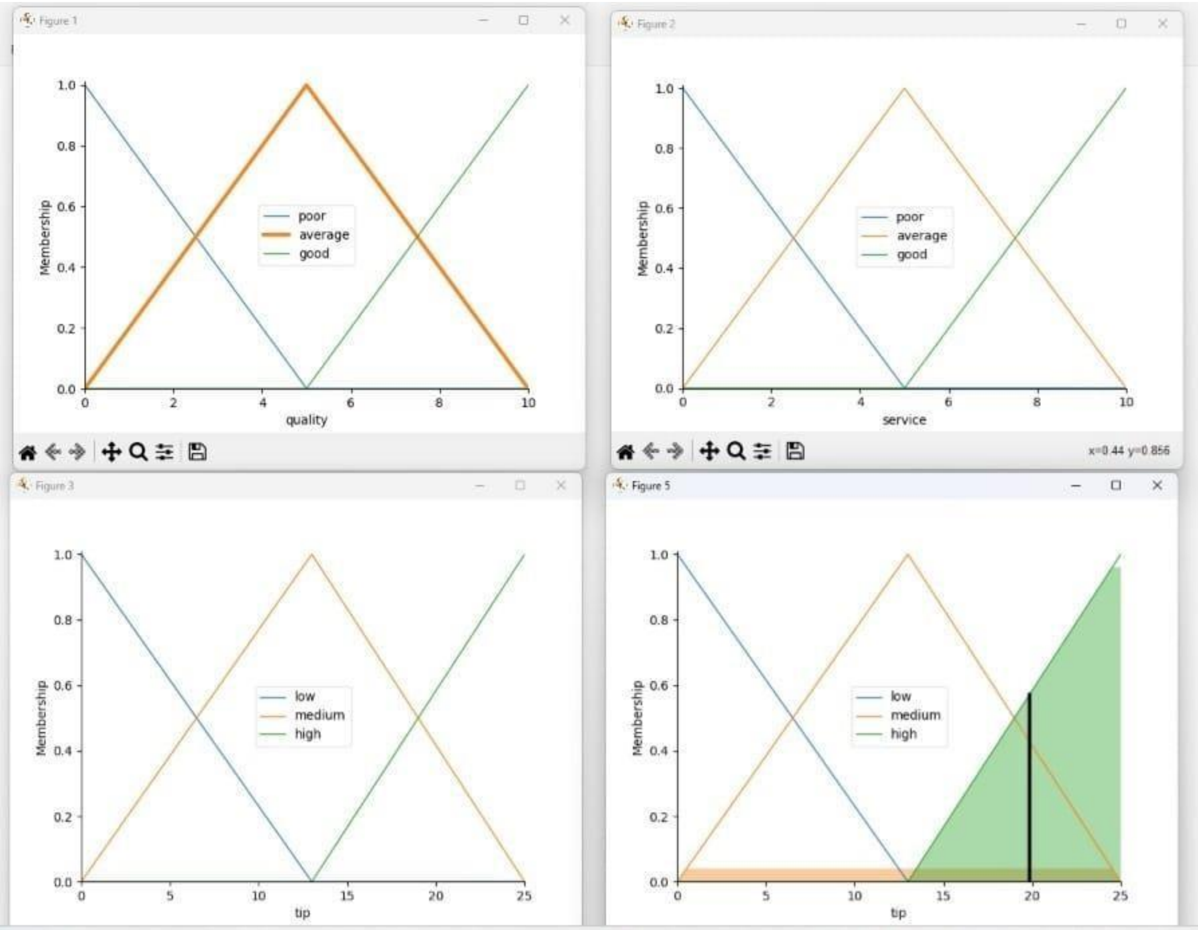
AIM: Fuzzy logic for tipping

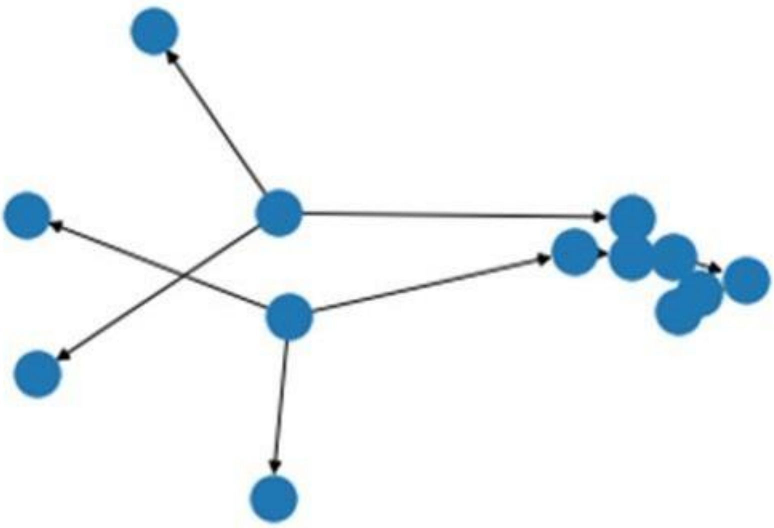
CODE:

```
!pip install scikit-fuzzy
import numpy as np
import skfuzzy as fuzz
from skfuzzy import control as ctrl

# New Antecedent/Consequent objects hold universe variables and membership
# functions
quality = ctrl.Antecedent(np.arange(0, 11, 1), 'quality')
service = ctrl.Antecedent(np.arange(0, 11, 1), 'service')
tip = ctrl.Consequent(np.arange(0, 26, 1), 'tip')

quality.automf(3)
service.automf(3)
tip['low'] = fuzz.trimf(tip.universe, [0, 0, 13])
tip['medium'] = fuzz.trimf(tip.universe, [0, 13, 25])
tip['high'] = fuzz.trimf(tip.universe, [13, 25, 25])
quality['average'].view()
service.view()
tip.view()
rule1 = ctrl.Rule(quality['poor'] | service['poor'], tip['low'])
rule2 = ctrl.Rule(service['average'], tip['medium'])
rule3 = ctrl.Rule(service['good'] | quality['good'], tip['high'])

rule1.view()

tipping_ctrl = ctrl.ControlSystem([rule1, rule2, rule3])

tipping = ctrl.ControlSystemSimulation(tipping_ctrl)
tipping.input['quality'] = 6.5
tipping.input['service'] = 9.8
tipping.compute()

print (tipping.output['tip'])
tip.view(sim=tipping)
```

OUTPUT:

19.847607361963192





CONCLUSION: Thus implemented the experiment successfully.