SOFT COMPUTING TECHNIQUES

A PRACTICAL REPORT

ON

SOFT COMPUTING TECHNIQUES

SUBMITTED BY

Mr. Mohd Shadik Shaikh

UNDER THE GUIDANCE OF

Mrs. SHABANA ANSARI

Submitted in fulfillment of the requirements for qualifying
M.Sc.IT Part-1 Semester-1 Examination 2024-2025

University of Mumbai

Department of Information Technology

University of Mumbai

Institute of Distance and Open Learning (IDOL)

Dr. Shankardayal Sharma bhavan, Vidyanagari, Santacruz(E)

PCP CENTER: RIZVI COLLEGE, BANDRA (W)

Certificate

This is to certify that ***Soft Computing Techniques Practical*** *performed at RIZVI COLLEGE, BANDRA (W) by Mr.* ***Mohd Shadik Jamal Akhtar Shaikh*** *holding Seat No/Application No. 1316026 /6719 studying Masters of Science in Information Technology Semester – 1 has been satisfactorily completed as prescribed by the University of Mumbai, during the year 2024 – 2025.*

Subject In-Charge **Coordinator In-Charge** **External Examiner**

INDEX

Practical 1

Aim: Design a Simple Neural Network Model.

Code:

```
x = float(input("Enter value of x: "))
w = float(input("Enter value of weight w: "))
b = float(input("Enter value of bias b: "))
net = int(w * x + b)

if (net<0):
     out = 0
elif ((net>=0) and (net<=1)):
    out = net
else:
    out = 1

print("net=",net)
print("output=",out)
```

Output:

```
PS C:\Users\raman> python .\example.py
Enter value of x: 1
Enter value of weight w: 65
Enter value of bias b: 50
net= 115
output= 1
```

Practical 1-B:-

Aim: Calculate the output of a Neural Net using both binary and bipolar sigmoidal function.

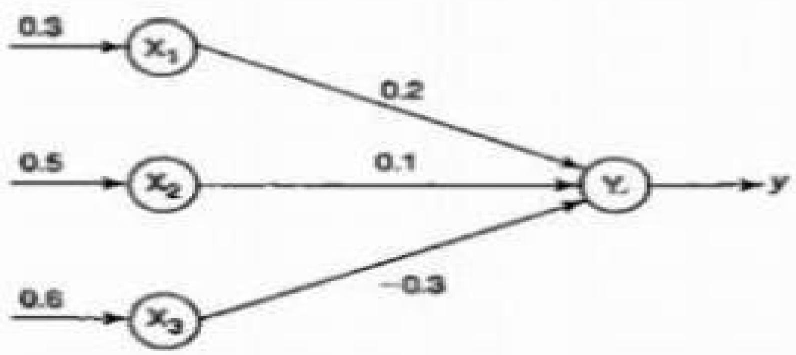


Figure 1 Neural net.

Code:

```
# number of elements as input
n = int(input("Enter number of elements : "))

# In[2]:
print("Enter the inputs")

# creating an empty list for inputs
inputs = []

# iterating till the range
for i in range(0, n):
    ele = float(input())
    inputs.append(ele)

# adding the element
print(inputs) # In[3]:
print("Enter the weights")

# creating an empty list for weights
weights = []
# iterating till the range
for i in range(0, n):
    ele = float(input())
    weights.append(ele)

# adding the element
print(weights)

# In[4]:
print("The net input can be calculated as Yin = x1w1 + x2w2 + x3w3")
```

```
# In[5]:
Yin = []
for i in range(0, n):
    Yin.append(inputs[i]*weights[i])
print(round(sum(Yin),3))
```

Output:

```
PS C:\Users\raman> python .\example.py
Enter number of elements : 3
Enter the inputs
1
2
3
[1.0, 2.0, 3.0]
Enter the weights
10
20
30
[10.0, 20.0, 30.0]
The net input can be calculated as Yin = x1w1 + x2w2 + x3w3
140.0
```

Practical 2

Practical 2-A:-

Aim: Implementation of AND.NOT function using McCulloch-Pitts neuron (use binary data representation).

Code:

```
# enter the no of inputs
num_ip = int(input("Enter the number of inputs : "))

#Set the weights with value 1
w1 = 1
w2 = 1
print("For the ", num_ip , " inputs calculate the net input using yin = x1w1 +
x2w2 ")
x1 = []
x2 = []
for j in range(0, num_ip):
    ele1 = int(input("x1 = "))
    ele2 = int(input("x2 = "))
    x1.append(ele1)
    x2.append(ele2)
print("x1 = ",x1)
print("x2 = ",x2)
n = x1 * w1
m = x2 * w2
Yin = []
for i in range(0, num_ip):
    Yin.append(n[i] + m[i])
    print("Yin = ",Yin)

#Assume one weight as excitatory and the other as inhibitory, i.e.,
Yin = []
for i in range(0, num_ip):
    Yin.append(n[i] - m[i])

print("After assuming one weight as excitatory and the other as inhibitory Yin
= ",Yin)

#From the calculated net inputs, now it is possible to fire the neuron for
input (1, 0)
#only by fixing a threshold of 1, i.e., θ ≥ 1 for Y unit.
#Thus, w1 = 1, w2 = -1; θ ≥ 1
Y=[]
for i in range(0, num_ip):
    if(Yin[i]>=1):
        ele= 1
        clear
        Y.append(ele)
```

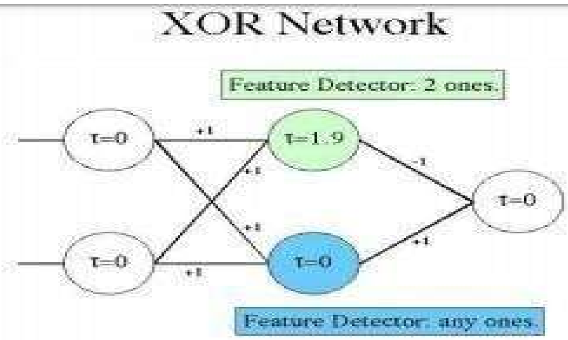
```
    if(Yin[i]<1):
        ele= 0
        Y.append(ele)
print("Y = ",Y)
```

Output:

```
PS C:\Users\raman> python example.py
Enter the number of inputs : 4
For the  4  inputs calculate the net input using yin = x1w1 + x2w2
x1 = 0
x2 = 0
x1 = 1
x2 = 0
x1 = 0
x2 = 1
x1 = 1
x2 = 1
x1 =  [0, 1, 0, 1]
x2 =  [0, 0, 1, 1]
Yin =  [0]
Yin =  [0, 1]
Yin =  [0, 1, 1]
Yin =  [0, 1, 1, 2]
After assuming one weight as excitatory and the other as inhibitory Yin =  [0, 1, -1, 0]
Y =  [0, 1, 0, 0]
```

Practical 2-B:-

Aim: Generate XOR function using McCulloch-Pitts neural net.



The XOR (exclusive or) function is defined by the following truth table:

Input1	Input2	XOR Output
0	0	0
0	1	1
1	0	1
1	1	0

Code:

```
import numpy as np

#Getting weights and threshold value
print('Enter weights')
w11=int(input('Weight w11='))
w12=int(input('weight w12='))

w21=int(input('Weight w21='))
w22=int(input('weight w22='))
v1=int(input('weight v1='))
v2=int(input('weight v2='))
print('Enter Threshold Value')
theta=int(input('theta='))
x1=np.array([0, 0, 1, 1])
x2=np.array([0, 1, 0, 1])
z=np.array([0, 1, 1, 0])
con=1
y1=np.zeros((4,))
y2=np.zeros((4,))
y=np.zeros((4,))
if con==1:
    zin1=np.zeros((4,))
    zin2=np.zeros((4,))
    zin1=x1*w11+x2*w21
    zin2=x1*w21+x2*w22
print("z1",zin1)
```

```
print("z2",zin2)
for i in range(0,4):
    if zin1[i]>=theta:
        y1[i]=1
    else:
        y1[i]=0
    if zin2[i] >= theta:
        y2[i]=1
    else:
        y2[i]=0
yin = np.array([])
yin = y1*v1+y2*v2
for i in range(0,4):
    if yin[i]>=theta:
        y[i]=1
    else:
        y[i]=0
        print("yin",yin)
        print('Output of Net')
        y = y.astype(int)
        print("y",y)
        print("z",z)
if np.array_equal(y,z):
        con=0
else:
    print("Net is not learning enter another set of weights and Threshold 
value")
    w11=input("Weight w11=")
    w12=input("weight w12=")

    w21=input("Weight w21=")
    w22=input("weight w22=")

    v1=input("weight  v1=")
    v2=input("weight  v2=")
    theta=input("theta=")
print("McCulloch-Pitts Net for XOR function")
print("Weights of Neuron Z1")
print(w11)
print(w21)
print("weights of Neuron Z2")
print(w12)
print(w22)
print("weights of Neuron Y")
print(v1)
print(v2)
print("Threshold value")
print(theta)
```

Output:

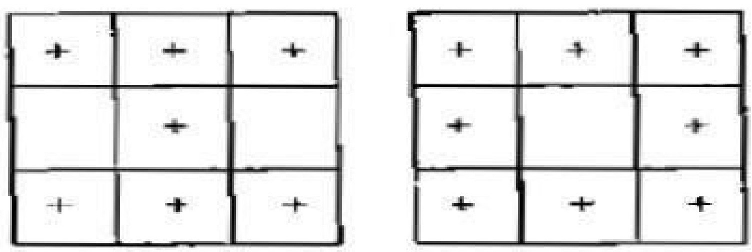
```
PS C:\Users\raman> python example.py
Enter weights
Weight w11=1
weight w12=-1
Weight w21=-1
weight w22=1
weight v1=1
weight v2=1
Enter Threshold Value
theta=1
z1 [ 0 -1  1  0]
z2 [ 0  1 -1  0]
yin [0. 1. 1. 0.]
Output of Net
y [0 0 0 0]
z [0 1 1 0]
yin [0. 1. 1. 0.]
Output of Net
y [0 1 1 0]
z [0 1 1 0]
McCulloch-Pitts Net for XOR function
Weights of Neuron Z1
1
-1
weights of Neuron Z2
-1
1
weights of Neuron Y
1
1
Threshold value
1
```

Practical 3

Practical 3-A:-

Aim: Write a program to implement Hebb's Rule.

Using the Hebb rule, find the weights required to perform the following classifications of the given input patterns shown in Figure 16. The pattern is shown as 3 × 3 matrix form in the squares. The "+" symbols represent the value "1" and empty squares indicate "−1." Consider "I" belongs to the members of class (so has target value 1) and "O" does not belong to the members of class (so has target value −1).



'I' 'O'

Code:

```
import numpy as np

x1 = np.array([1,1,1,-1,1,-1,1,1,1])
x2 = np.array([1,1,1,1,-1,1,1,1,1])
b = 0
y = np.array([1,-1])
wtold = np.zeros((9,))
wtnew = np.zeros((9,))
wtnew = wtnew.astype(int)
wtold = wtold.astype(int)
bais = 0
print("First input with target =1")

for i in range(0,9):
    wtold[i] = wtold[i] + x1[i] * y[0]

wtnew = wtold
b = b + y[0]

print("new wt =", wtnew)
print("Bias value",b)

print("Second input with target =-1")
```

```
for i in range(0,9):
    wtnew[i] = wtold[i] + x2[i] * y[1]
b = b + y[1]
print("new wt =", wtnew)
print("Bias value",b)
```

Output:

```
PS C:\Users\raman> python .\Hebb.py
First input with target =1
new wt = [ 1  1  1 -1  1 -1  1  1  1]
Bias value 1
Second input with target =-1
new wt = [ 0  0  0 -2  2 -2  0  0  0]
Bias value 0
```

Practical 3-B:-

Aim: Write a program to implement Delta Rule.

Code:

```python
# Supervised Learning
import numpy as np
import time
np.set_printoptions(precision=2)
x=np.zeros((3,))
weights=np.zeros((3,))
desired=np.zeros((3,))
actual=np.zeros((3,))
for i in range(0,3):
    x[i]=float(input("Initial inputs:"))
for i in range(0,3):
    weights[i]=float(input("Initial weights:"))
for i in range(0,3):
    desired[i]=float(input("Desired output:"))

a = float(input("Enter learning rate:"))
actual = x*weights
print("actual",actual)
print("desired",desired)

while True:
    if np.array_equal(desired,actual):
        break #no change
    else:
        for i in range(0,3):
            weights[i]=weights[i]+a*(desired[i]-actual[i])
            actual=x*weights
print("weights",weights)
print("actual",actual)
print("desired",desired)
print("*"*30)
print("Final output")
print("Corrected weights",weights)
print("actual",actual)
print("desired",desired)
```

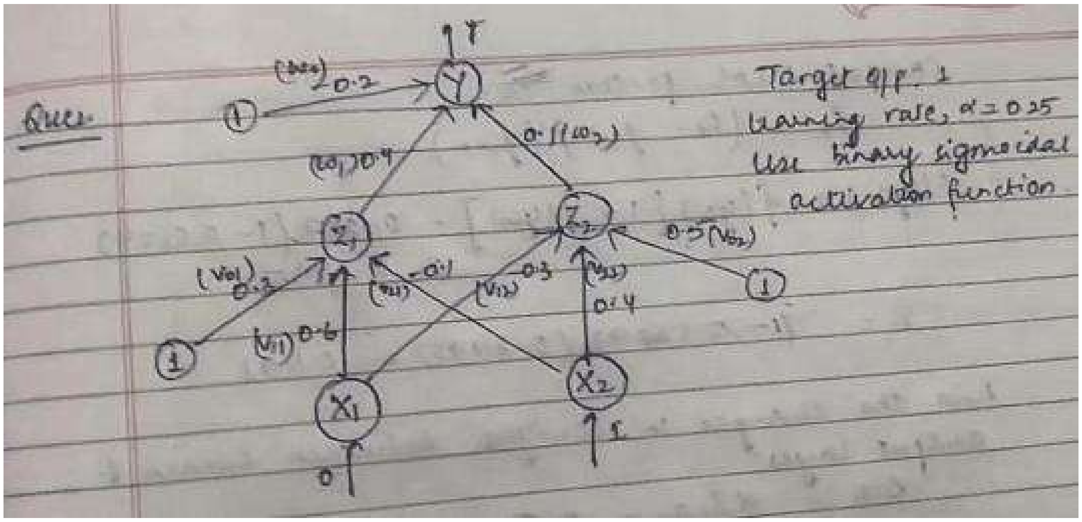
Output:

```
PS C:\Users\raman> python .\Delta.py
Initial inputs:1
Initial inputs:1
Initial inputs:1
Initial weights:1
Initial weights:1
Initial weights:1
Desired output:2
Desired output:3
Desired output:4
Enter learning rate:1
actual [1. 1. 1.]
desired [2. 3. 4.]
weights [2. 3. 4.]
actual [2. 3. 4.]
desired [2. 3. 4.]
******************************
Final output
Corrected weights [2. 3. 4.]
actual [2. 3. 4.]
desired [2. 3. 4.]
```

Practical 4

Practical 4-A:-

Aim: Write a program for Back Propagation Algorithm.



Code:

```
import numpy as np
import math
import decimal

np.set_printoptions(precision=2)

v1=np.array([0.6, 0.3])
v2=np.array([-0.1, 0.4])
w=np.array([-0.2,0.4,0.1])
b1=0.3
b2=0.5
x1=0
x2=1
alpha=0.25

print("calculate net input to z1 layer")
zin1=round(b1+ x1*v1[0]+x2*v2[0],4)

print("z1=",round(zin1,3))
print("calculate net input to z2 layer")
zin2=round(b2+ x1*v1[1]+x2*v2[1],4)

print("z2=",round(zin2,4))
print("Apply activation function to calculate output")
z1=1/(1+math.exp(-zin1))
z1=round(z1,4)
z2=1/(1+math.exp(-zin2))
z2=round(z2,4)
```

```
print("z1=",z1)
print("z2=",z2)
print("calculate net input to output layer")
yin=w[0]+z1*w[1]+z2*w[2]
print("yin=",yin)
print("calculate net output")
y=1/(1+math.exp(-yin))

print("y=",y)
fyin=y *(1- y)
dk=(1-y)*fyin

print("dk",dk)
dw1= alpha * dk * z1
dw2= alpha * dk * z2
dw0= alpha * dk

print("compute error portion in delta")
din1=dk* w[1]
din2=dk* w[2]

print("din1=",din1)
print("din2=",din2)

print("error in delta")
fzin1= z1 *(1-z1)

print("fzin1",fzin1)
d1=din1* fzin1
fzin2= z2 *(1-z2)

print("fzin2",fzin2)
d2=din2* fzin2

print("d1=",d1)
print("d2=",d2)

print("Changes in weights between input and hidden layer")
dv11=alpha * d1 * x1

print("dv11=",dv11)
dv21=alpha * d1 * x2

print("dv21=",dv21)
dv01=alpha * d1

print("dv01=",dv01)
dv12=alpha * d2 * x1
```

```
print("dv12=",dv12)
dv22=alpha * d2 * x2

print("dv22=",dv22)
dv02=alpha * d2

print("dv02=",dv02)
print("Final weights of network")
v1[0]=v1[0]+dv11
v1[1]=v1[1]+dv12

print("v=",v1)
v2[0]=v2[0]+dv21
v2[1]=v2[1]+dv22

print("v2",v2)
w[1]=w[1]+dw1
w[2]=w[2]+dw2
b1=b1+dv01
b2=b2+dv02
w[0]=w[0]+dw0
print("w=",w)
print("bias b1=",b1, " b2=",b2)
```

Output:

```
PS C:\Users\raman> python .\BackPropogation.py
calculate net input to z1 layer
z1= 0.2
calculate net input to z2 layer
z2= 0.9
Apply activation function to calculate output
z1= 0.5498
z2= 0.7109
calculate net input to output layer
yin= 0.09101
calculate net output
y= 0.5227368084248941
dk 0.11906907074145694
compute error portion in delta
din1= 0.04762762829658278
din2= 0.011906907074145694
error in delta
fzin1 0.24751996
fzin2 0.20552119000000002
d1= 0.011788788650865037
d2= 0.0024471217110978417
Changes in weights between input and hidden layer
dv11= 0.0
dv21= 0.0029471971627162592
dv01= 0.0029471971627162592
dv12= 0.0
dv22= 0.0006117804277744604
dv02= 0.0006117804277744604
Final weights of network
v= [0.6 0.3]
v2 [-0.1  0.4]
w= [-0.17  0.42  0.12]
bias b1= 0.30294719716271623  b2= 0.5006117804277744
```

Practical 4-B:-

Aim: Write a program for Error Back Propagation Algorithm(EBPA).

Code:

```
import math

a0 = -1
t = -1

w10 = float(input("Enter weight first network: "))
b10 = float(input("Enter base first network: "))
w20 = float(input("Enter weight second network: "))
b20 = float(input("Enter base second network: "))
c = float(input("Enter learning coefficient: "))

n1 = float(w10*c+b10)
a1 = math.tanh(n1)
n2 = float(w20*a1+b20)
a2 = math.tanh(float(n2))

e = t-a2
s2 = -2*(1-a2*a2)*e
s1 = (1-a1*a1)*w20*s2
w21 = w20-(c*s2*a1)
w11 = w10-(c*s1*a0)
b21 = b20-(c*s2)
b11 = b10-(c*s1)

print("The updated weight of first n/w w11= ",w11)
print("The uploaded weight of second n/w w21= ",w21)
print("The updated base of first n/w b10= ",b10)
print("The updated base of second n/w b20= ",b20)
```

Output:

```
PS C:\Users\raman> python .\ErrorBackPropogation.py
Enter weight first network: 12
Enter base first network: 35
Enter weight second network: 23
Enter base second network: 45
Enter learning coefficient: 11
The updated weight of first n/w w11=  12.0
The uploaded weight of second n/w w21=  23.0
The updated base of first n/w b10=  35.0
The updated base of second n/w b20=  45.0
```

Practical 5

Practical 5-A:-

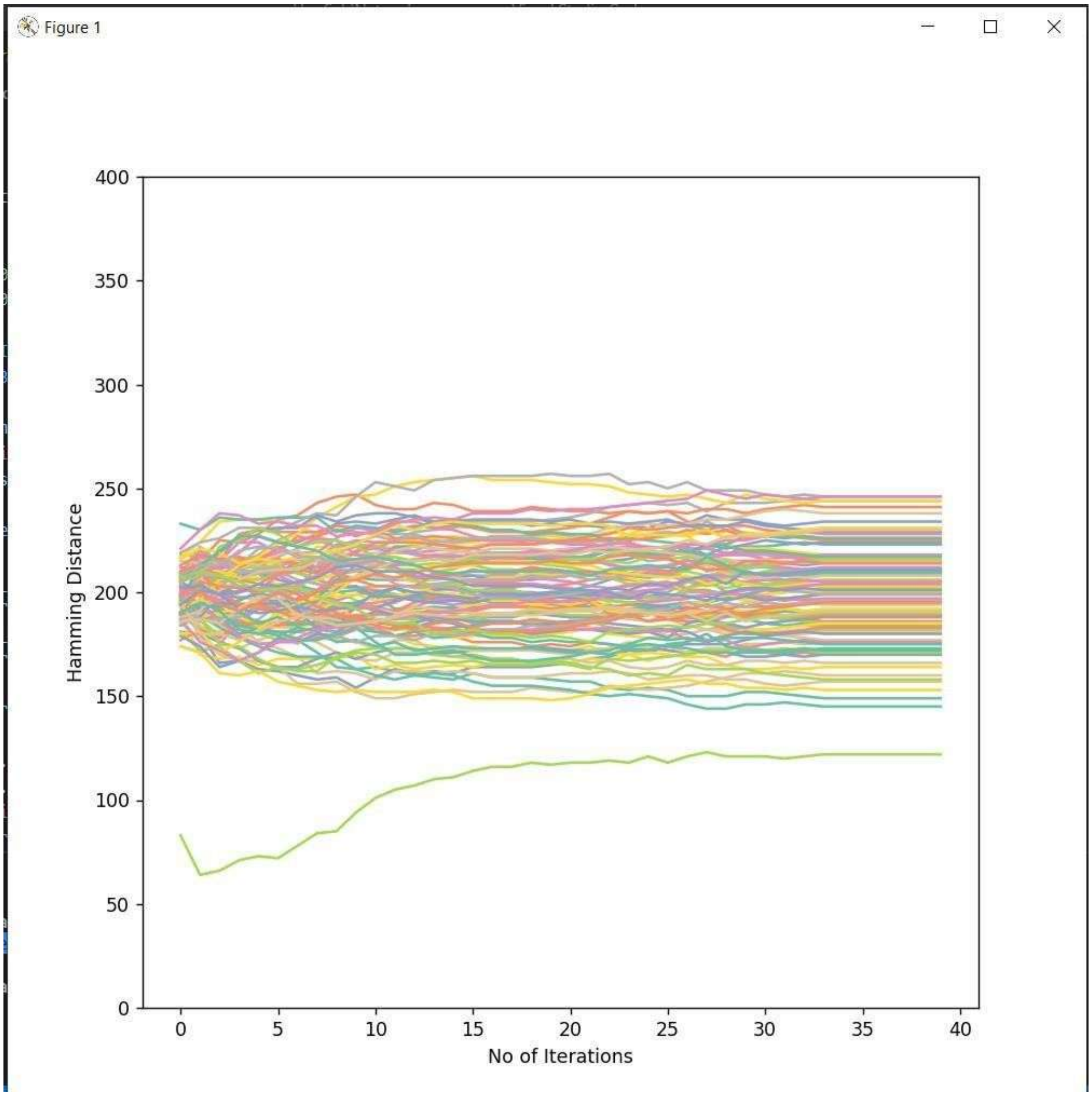
Aim: Write a program for Hopfield Network.

Code:

```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
sns.set_palette('Set2')
N = 400
P = 100
N_sqrt = np.sqrt(N).astype('int32')
NO_OF_ITERATIONS = 40
NO_OF_BITS_TO_CHANGE = 200

epsilon = np.asarray([np.random.choice([1, -1], size=N)])
for i in range(P-1):
    epsilon = np.append(epsilon, [np.random.choice([1, -1], size=N)], axis=0)

print(epsilon.shape)

random_pattern = np.random.randint(P)
test_array = epsilon[random_pattern]
random_pattern_test = np.random.choice([1, -1], size=NO_OF_BITS_TO_CHANGE)
test_array[:NO_OF_BITS_TO_CHANGE] = random_pattern_test

print(random_pattern)

w = np.zeros((N, N))
h = np.zeros(N)
for i in range(N):
    for j in range(N):
        for p in range(P):
            w[i, j] += (epsilon[p, i]*epsilon[p, j]).sum()
        if i==j:
            w[i, j] = 0
w /= N
hamming_distance = np.zeros((NO_OF_ITERATIONS, P))
for iteration in range(NO_OF_ITERATIONS):
    for _ in range(N):
        i = np.random.randint(N)
        h[i] = 0
        for j in range(N):
            h[i] += w[i, j]*test_array[j]
    test_array = np.where(h<0, -1, 1)

    for i in range(P):
```

```
        hamming_distance[iteration, i] = ((epsilon - test_array)[i]!=0).sum()

fig = plt.figure(figsize = (8, 8))
plt.plot(hamming_distance)
plt.xlabel('No of Iterations')
plt.ylabel('Hamming Distance')
plt.ylim([0, N])
plt.show()
```

Output:



Practical 5-B:-

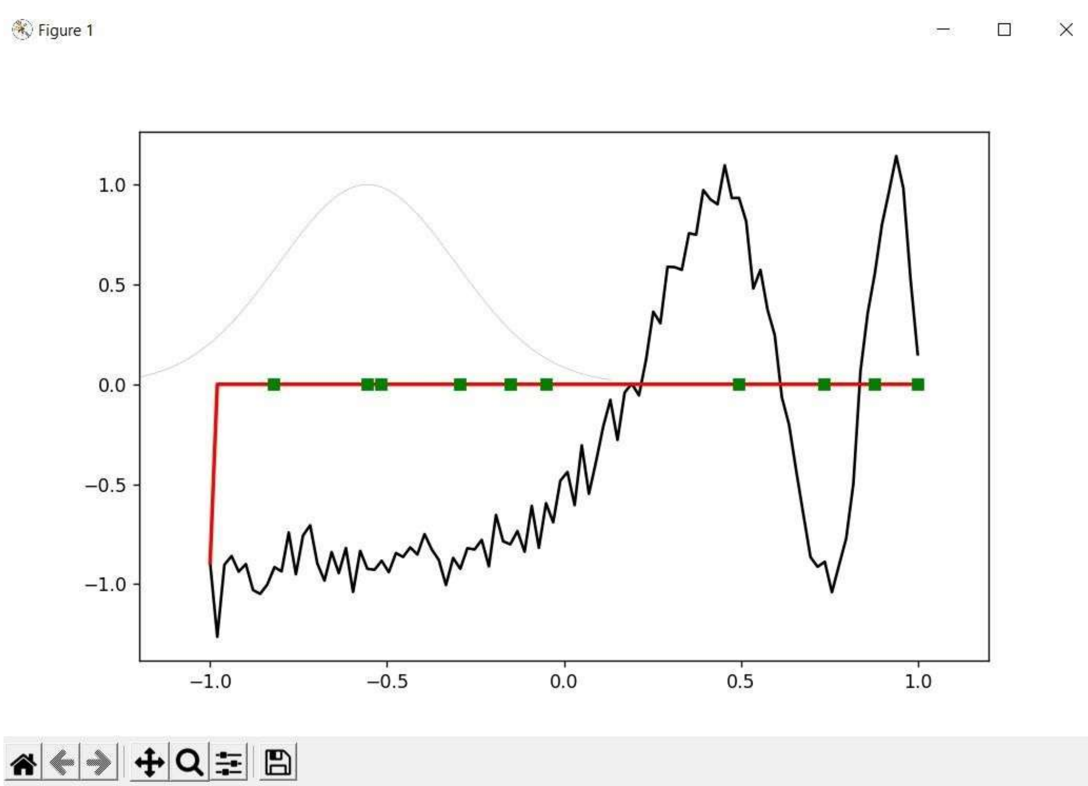
Aim: Write a program for Radial Basis Function.

Code:

```
from scipy import *
from scipy.linalg import norm, pinv
from matplotlib import pyplot as plt
from numpy import random as random
import numpy as np
class RBF:
    def _init_(self, indim, numCenters, outdim):
        self.indim =indim
        self.outdim =outdim
        self.numCenters =numCenters
        self.centers = [random.uniform(-1, 1, indim)
        for i in range(numCenters)]
        self.beta = 8
        self.W =random.random((self.numCenters, self.outdim))
    def _basisfunc(self, c, d):
        assert len(d) ==self.indim
        return np.exp(-self.beta *norm(c-d)**2)
    def _calcAct(self, X):
        # calculate activations of RBFs
        G =np.zeros((X.shape[0], self.numCenters), float)
        for ci, c in enumerate(self.centers):
            for xi, x in enumerate(X):
                G[xi,ci] = self._basisfunc(c, x)
                return G
    def train(self, X, Y):
        """ X: matrix of dimensions n x indim
            y: column vector of dimension n x 1 """
        # choose random center vectors from training set
        rnd_idx = random.permutation(X.shape[0])[:self.numCenters]
        self.centers =[X[i,:] for i in rnd_idx]
        print("center", self.centers)
        # calculate activations of RBFs
        G =self._calcAct(X)
        print (G)
        # calculate output weights (pseudoinverse)
        self.W = np.dot(pinv(G), Y)
    def test(self, X):
        """ X: matrix of dimensions n x indim """
        G = self._calcAct(X)
        Y = np.dot(G, self.W)
        return Y
if __name__=='__main_':
    # -------- 1D Example ------------------------------------------------------------------
    n =100
```

```
    x = np.mgrid [-1:1:complex(0,n)].reshape(n, 1)
    # set y and add random noise
    y = np.sin(3*(x+0.5)**3-1)
    y += random.normal(0, 0.1, y.shape)
    # rbf regression
    rbf = RBF(1, 10, 1)
    rbf.train(x, y)
    z = rbf.test(x)
    # plot original data
    plt.figure(figsize=(12, 8))
    plt.plot(x, y, 'k-')
    # plot learned model
    plt.plot(x, z, 'r-', linewidth=2)
    # plot rbfs
    plt.plot(rbf.centers, np.zeros(rbf.numCenters), 'gs')
    for i in range(1):
        # RF prediction lines
        ix = np.arange(i-0.7, i+0.7, 0.01)
        iy = [rbf._basisfunc(np.array([ix_]), np.array([i])) for ix_ in ix]
        plt.plot(ix, iy, '-', color = 'gray', linewidth = 0.2)
        plt.xlim(-1.2, 1.2)
        plt.show()
```

Output:



Practical 6

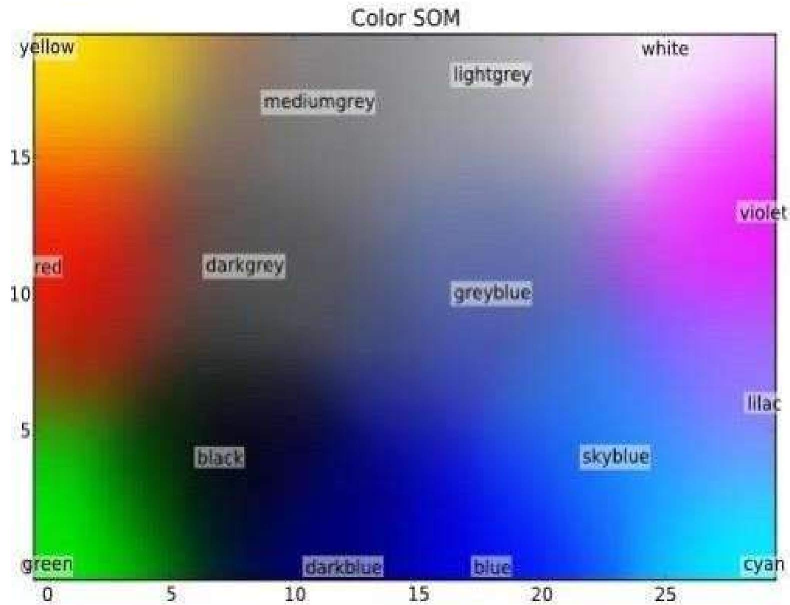
Practical 6-A:-

Aim: Write a program for Self-Organising Maps.

Code:

```
from mvpa2.suite import *

 colors=np.array( [[0.,0.,0.],
                    [0.,0.,1.],
                    [0.,0.,0.5],
                    [0.125,0.529,1.0],
                    [0.33,0.4,0.67],
                    [0.6,0.5,1.0],
                    [0.,1.,0.],
                    [1.,0.,0.],
                    [0.,1.,1.],
                    [1.,0.,1.],
                    [1.,1.,0.],
                    [1.,1.,1.],
                    [.33,.33,.33],
                    [.5,.5,.5],
                    [.66,.66,.66]])

color_names =  ['black','blue','darkblue','skyblue',
                'greyblue','lilac','green','red',
                'cyan','violet','yellow','white',
                'darkgrey','mediumgrey','lightgrey']

som = SimpleSOMMapper((20,30),400,learning_rate=0.05)

som.train(colors)
pl.imshow(som.K,origin='lower')
mapped = som(colors)
pl.title('Color SOM')

# SOM's kshape is (rows x columns), while matplotlib wants (X x Y)
for i,minenumerate(mapped):
    pl.text(m[1],m[0],color_names[i],ha='center',va='center',
bbox=dict(facecolor='white',alpha=0.5,lw=0))
```

Output:


Color SOM
yellow
white
lightgrey
mediumgrey
violet
red
darkgrey
greyblue
lilac
black
skyblue
green
darkblue
blue
cyan
0
5
10
15
20
25
15
10
5

Practical 6-B:-

Aim: Write a program for Adaptive Resonance Theory.

Code:

```
import numpy as np
VIGILANCE = 0.6 # trashhold 0 - 1.0
LEARNING_COEF = 0.5  # standard
train = np.array([[1,0,0,0,0,0],
                  [1,1,1,1,1,0],
                  [1,0,1,0,1,0],
                  [0,1,0,0,1,1],
                  [1,1,1,0,0,0],
                  [0,0,1,1,1,0],
                  [1,1,1,1,1,0],
                  [1,1,1,1,1,1]], np.float)

 test = np.array([[1,1,1,1,1,1],
                  [1,1,1,1,1,0],
                  [1,1,1,1,0,0],
                  [1,1,1,0,0,0],
                  [1,1,0,0,0,0],
                  [1,0,0,0,0,0],
                 [0,0,0,0,0,0]], np.float)

L1_neurons_cnt = len(train[0])
L2_neurons_cnt = 1
# Init weights from the first neuron
bottomUps = np.array([[1/(L1_neurons_cnt + 1) for _ in
range(L1_neurons_cnt)]], np.float)
topDowns = np.array([[1 for _ in range(L1_neurons_cnt)]], np.float)

for tv in train:
    print(" ------ ")
    print('Train vector:', tv)
    createNewNeuron = True
    outputs = [bottomUps[i].dot(tv) for i in range(L2_neurons_cnt)]
    counter = L2_neurons_cnt
    while  counter  >  0:
        winning_output = max(outputs)
        winner_neuron_idx = outputs.index(winning_output)
        # NOTE!!! Sometimes there can be more than one winning neurons
        # Then we should select them randomly. For sake of simplicity,
        # this was not implemented for sake of simplicity

        # Because `sum(tv)` can be 0 and we can not divide by zero :(
        tv_sum = sum(tv)
        if tv_sum == 0:
            similarity = 0
```

```
        else:
            similarity = topDowns[winner_neuron_idx].dot(tv)/(sum(tv))
        print(" ", topDowns[winner_neuron_idx])
        print("    Bottom Ups Weights:", bottomUps[winner_neuron_idx])
        print("    Similartiy:", similarity)
        if similarity >= VIGILANCE:
            # Found similar neuron -> update their weights
            createNewNeuron = False
            new_bottom_weights = tv *
topDowns[winner_neuron_idx]/(LEARNING_COEF+tv.dot(topDowns[winner_neuron_idx])
)
            new_top_weights = tv * topDowns[winner_neuron_idx]
            topDowns[winner_neuron_idx] = new_top_weights
            bottomUps[winner_neuron_idx] = new_bottom_weights
            break
        else:
            # Didn't find similar neuron
            outputs[winner_neuron_idx] = -1  # So it won't be selected in the
next iteration
            counter -= 1

    if createNewNeuron:
        print("  Creating a new new neuron")
        new_bottom_weights = np.array([[i/(LEARNING_COEF + sum(tv)) for i in
tv]], np.float)
        new_top_weights = np.array([[i for i in tv]], np.float)
        print("    Weights bottomUps:", new_bottom_weights)
        print("    Weights topDowns:", new_top_weights)
        bottomUps = np.append(bottomUps, new_bottom_weights, axis=0)
        topDowns = np.append(topDowns, new_top_weights, axis=0)
        L2_neurons_cnt += 1

print("=====")
print(f"Total Classes: {L2_neurons_cnt}")
print("Center of masses")
print(topDowns)
for tv in test:
    A = list(range(L2_neurons_cnt))
    createNewNeuron = True
    outputs = [bottomUps[i].dot(tv) for i in A]
    winning_weight = max(outputs)
    winner_neuron_idx = outputs.index(winning_weight)
    print(f"Class {winner_neuron_idx} for train vector {tv}")
```

Output:

```
-------
Train vector: [1. 0. 0. 0. 0. 0.]
  [1. 1. 1. 1. 1. 1.]
    Bottom Ups Weights: [0.14285714 0.14285714 0.14285714 0.14285714 0.14285714 0.14285714]
    Similartiy: 1.0
-------
Train vector: [1. 1. 1. 1. 1. 0.]
  [1. 0. 0. 0. 0. 0.]
    Bottom Ups Weights: [0.66666667 0.         0.         0.         0.         0.        ]
    Similartiy: 0.2
  Creating a new new neuron
```

```
-------
Train vector: [1. 0. 1. 0. 1. 0.]
  [1. 0. 0. 0. 0. 0.]
    Bottom Ups Weights: [0.66666667 0.         0.         0.         0.         0.        ]
    Similartiy: 0.3333333333333333
  [1. 1. 1. 1. 1. 0.]
    Bottom Ups Weights: [0.18181818 0.18181818 0.18181818 0.18181818 0.18181818 0.        ]
    Similartiy: 1.0
-------
Train vector: [0. 1. 0. 0. 1. 1.]
  [1. 0. 1. 0. 1. 0.]
    Bottom Ups Weights: [0.28571429 0.         0.28571429 0.         0.28571429 0.        ]
    Similartiy: 0.3333333333333333
  [1. 0. 0. 0. 0. 0.]
    Bottom Ups Weights: [0.66666667 0.         0.         0.         0.         0.        ]
    Similartiy: 0.0
  Creating a new new neuron
    Weights bottomUps: [[0.         0.28571429 0.         0.         0.28571429 0.28571429]]
    Weights topDowns: [[0. 1. 0. 0. 1. 1.]]
```

```
-------
Train vector: [1. 1. 1. 0. 0. 0.]
  [1. 0. 0. 0. 0. 0.]
    Bottom Ups Weights: [0.66666667 0.         0.         0.         0.         0.        ]
    Similartiy: 0.3333333333333333
  [1. 0. 1. 0. 1. 0.]
    Bottom Ups Weights: [0.28571429 0.         0.28571429 0.         0.28571429 0.        ]
    Similartiy: 0.6666666666666666
-------
Train vector: [0. 0. 1. 1. 1. 0.]
  [1. 0. 1. 0. 0. 0.]
    Bottom Ups Weights: [0.4 0.  0.4 0.  0.  0. ]
    Similartiy: 0.3333333333333333
  [0. 1. 0. 0. 1. 1.]
    Bottom Ups Weights: [0.         0.28571429 0.         0.         0.28571429 0.28571429]
    Similartiy: 0.3333333333333333
  [1. 0. 0. 0. 0. 0.]
    Bottom Ups Weights: [0.66666667 0.         0.         0.         0.         0.        ]
    Similartiy: 0.0
  Creating a new new neuron
```

```
  Creating a new new neuron
    Weights bottomUps: [[0.         0.         0.28571429 0.28571429 0.28571429 0.        ]]
    Weights topDowns: [[0. 0. 1. 1. 1. 0.]]
-------
Train vector: [1. 1. 1. 1. 1. 0.]
  [0. 0. 1. 1. 1. 0.]
    Bottom Ups Weights: [0.         0.         0.28571429 0.28571429 0.28571429 0.        ]
    Similartiy: 0.6
-------
Train vector: [1. 1. 1. 1. 1. 1.]
  [0. 1. 0. 0. 1. 1.]
    Bottom Ups Weights: [0.         0.28571429 0.         0.         0.28571429 0.28571429]
    Similartiy: 0.5
  [0. 0. 1. 1. 1. 0.]
    Bottom Ups Weights: [0.         0.         0.28571429 0.28571429 0.28571429 0.        ]
    Similartiy: 0.5
  [1. 0. 1. 0. 0. 0.]
  [1. 0. 0. 0. 0. 0.]
    Bottom Ups Weights: [0.66666667 0.         0.         0.         0.         0.        ]
    Similartiy: 0.16666666666666666
  Creating a new new neuron
    Weights bottomUps: [[0.15384615 0.15384615 0.15384615 0.15384615 0.15384615 0.15384615]]
    Weights topDowns: [[1. 1. 1. 1. 1. 1.]]
=====
```

```
=====
Total Classes: 5
Center of masses
[[1. 0. 0. 0. 0. 0.]
 [1. 0. 1. 0. 0. 0.]
 [0. 1. 0. 0. 1. 1.]
 [0. 0. 1. 1. 1. 0.]
 [1. 1. 1. 1. 1. 1.]]
Class 4 for train vector [1. 1. 1. 1. 1. 1.]
Class 3 for train vector [1. 1. 1. 1. 1. 0.]
Class 1 for train vector [1. 1. 1. 1. 0. 0.]
Class 1 for train vector [1. 1. 1. 0. 0. 0.]
Class 0 for train vector [1. 1. 0. 0. 0. 0.]
Class 0 for train vector [1. 0. 0. 0. 0. 0.]
Class 0 for train vector [0. 0. 0. 0. 0. 0.]
```

Practical 7

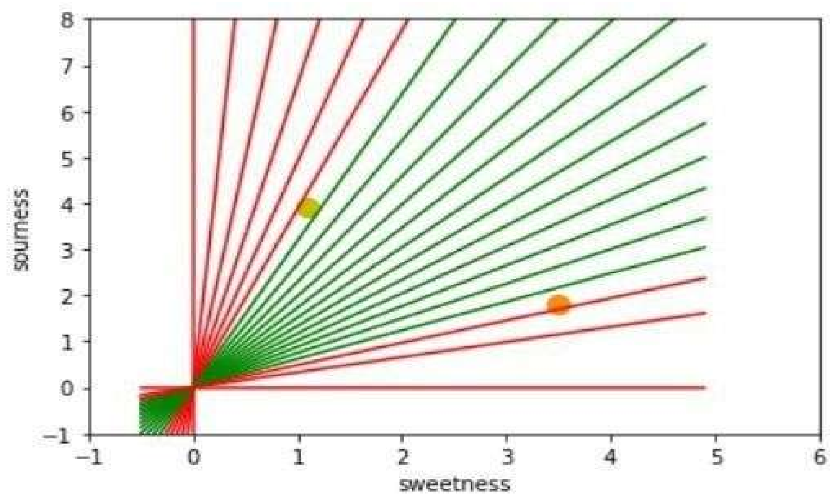
Aim: Line Separation

Code:

```
import numpy as np
import matplotlib.pyplot as plt
def create_distance_function(a, b, c):
    """ 0 = ax + by + c """
    def distance(x, y):
        """ returns tuple (d, pos)
        d is the distance
        If pos == -1 point is below the line,
        0 on the line and +1 if above the line
        """
        nom = a * x + b * y + c
        if nom == 0:
            pos = 0
        elif (nom<0 and b<0) or (nom>0 and b>0):
            pos = -1
        else:
            pos = 1
        return (np.absolute(nom) / np.sqrt( a ** 2 + b ** 2), pos)
    return distance
def main():
    points = [ (3.5, 1.8), (1.1, 3.9) ]
    fig, ax = plt.subplots()
    ax.set_xlabel("sweetness")
    ax.set_ylabel("sourness")
    ax.set_xlim([-1, 6])
    ax.set_ylim([-1, 8])
    X = np.arange(-0.5, 5, 0.1)
    colors = ["r", ""] # for the samples
    size = 10
    for (index, (x, y)) in enumerate(points):
        if index == 0:
            ax.plot(x, y, "o", color="darkorange", markersize=size)
        else:
            ax.plot(x, y, "oy", markersize=size)
        step = 0.05
    for x in np.arange(0, 1+step, step):
        slope = np.tan(np.arccos(x))
        dist4line1 = create_distance_function(slope, -1, 0)
        #print("x: ", x, "slope: ", slope)
        Y = slope * X
        results = [(3.9,-1)]
    for point in points:
        results.append(dist4line1(*point))
        print(slope, results)

        if (results[0][1] != results[1][1]):
```

```
            ax.plot(X, Y, "g-")
        else:
            ax.plot(X, Y, "r-")
    plt.show()

if _name_== "_main_":
    main()
```

Output:



Practical 8

Practical 8-A:-

Aim: Implementation of Membership and Identity operators (in, not in).

Code:

(in):

```
# Python program to illustrate
# Finding common member in list
# without using 'in' operator
# Define a function() that takes two lists
def overlapping(list1,list2):
    c=0
    d=0
    for i in list1:
        c+=1
    for i in list2:
        d+=1
    for i in range(0,c):
        for j in range(0,d):
            if(list1[i]==list2[j]):
                return 1
            else:
                return 0
def main():
    list1=[1,2,3,4,5]
    list2=[1,2,3,4,5,6,7,8,9]
    if(overlapping(list1,list2)):
        print("overlapping")
    else:
        print("not overlapping")

if _name_== "_main_":
    main()
```

(not in):

```
def main():
    x = 14
    list = [1,2,3,4,5,6,7,8,9]
    if x  not  in list:
        print("not overlapping")
    else:
        print("overlapping")

if _name_== "_main_":
    main()
```

Output:

(in):

```
PS C:\Users\raman> python example.py
overlapping
```

(Not in):

```
PS C:\Users\raman> python example.py
not overlapping
```

Practical 8-B:-

Aim: Implementation of Membership and Identity operators (is, is not).

Code:

(Is):

```
# Python program to illustrate the use
# of 'is' identity operator
x = 5
if (type(x) is int):
    print ("true")
else:
    print ("false")
```

(Is not):

```
# Python program to illustrate the
# use of 'is not' identity operator
x = 5.2
if (type(x) is not float):
    print ("true")
else:
    print ("false")
```

Output:

(is):

```
PS C:\Users\raman> python example.py
true
```

(Is not):

```
PS C:\Users\raman> python example.py
false
```

Practical 9

Practical 9-A:-

Aim: Find the ratios using Fuzzy Logic.

Code:

```
from fuzzywuzzy import fuzz
from fuzzywuzzy import process
s1 = "I love fuzzysforfuzzys"
s2 = "I am loving fuzzysforfuzzys"
print ("\nFuzzyWuzzy Ratio:", fuzz.ratio(s1, s2))
print ("\nFuzzyWuzzyPartialRatio: ", fuzz.partial_ratio(s1, s2))
print ("\nFuzzyWuzzyTokenSortRatio: ", fuzz.token_sort_ratio(s1, s2))
print ("\nFuzzyWuzzyTokenSetRatio: ", fuzz.token_set_ratio(s1, s2))
print ("\nFuzzyWuzzyWRatio: ", fuzz.WRatio(s1, s2),'\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:

```
PS C:\Users\raman> python .\Fuzzy.py

FuzzyWuzzy Ratio: 86

FuzzyWuzzyPartialRatio:  86

FuzzyWuzzyTokenSortRatio:  86

FuzzyWuzzyTokenSetRatio:  87

FuzzyWuzzyWRatio:  86

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

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

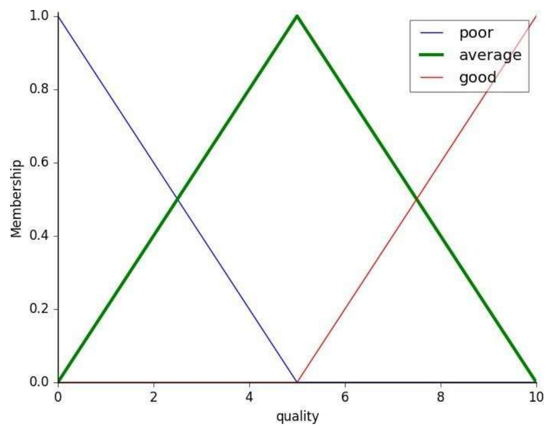
Practical 9-B:-

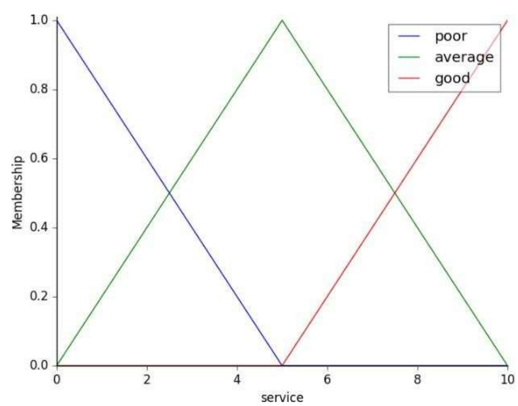
Aim: Solve Tipping Problem using Fuzzy Logic.

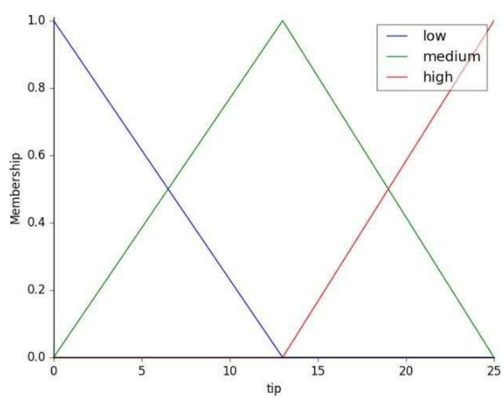
Code:

```
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')

# Auto-membership function population is possible with .automf(3, 5, or 7)
quality.automf(3)
service.automf(3)

# Custom membership functions can be built interactively with a familiar,
# Pythonic API
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])

""" To help understand what the membership looks like, use the ``view``
methods. """
# You can see how these look with .view()
quality['average'].view()

""" .. image:: PLOT2RST.current_figure """
(service.view())

""" .. image:: PLOT2RST.current_figure """
tip.view()

""" .. image:: PLOT2RST.current_figure"""
tip['medium'].view()
```

Output:


poor
average
good
Membership
1.0
0.8
0.6
0.4
0.2
0.0
0
2
4
6
8
10
quality


poor
average
good
Membership
1.0
0.8
0.6
0.4
0.2
0.0
0
2
4
6
8
10
service


low
medium
high
Membership
1.0
0.8
0.6
0.4
0.2
0.0
0
5
10
15
20
25
tip

Practical 10

Aim: Implementation of Simple Genetic Algorithm.

Code:

```
import random

#Number of population
POPULATION_SIZE = 250

# Random Genes
GENES = '''abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ 1234567890, .-
;:_!"#%&/()=?@${[]}'''

# Target string
TARGET = "Hello"

# Class to represent individuals in population
class Individual(object):

    # Intialize the chromosome and calculate the fitness
    def _init_(self, chromosome):
        self.chromosome = chromosome
        self.fitness = self.cal_fitness()

    #Create random genes for mutation
    @classmethod
    def mutated_genes(self):
        global GENES
        gene = random.choice(GENES)
        return gene

    #Create a chromosome or string of genes till target length
    @classmethod
    def create_gnome(self):
        global TARGET
        gnome_len = len(TARGET)
        return [self.mutated_genes() for _ in range(gnome_len)]

    #Calculate Fitness
    def cal_fitness(self):

        global TARGET
        fitness = 0
        for gs, gt in zip(self.chromosome, TARGET):
            if gs != gt: fitness += 1
        return fitness

    #Perform reproduction and create new offspring
```

```
    def crossover(self, par2):

        #offspring
        child_chromosome = []

        for gp1, gp2 in zip(self.chromosome, par2.chromosome):

            #prob
            prob = random.random()

            if prob < 0.50:
                child_chromosome.append(gp1)

            elif prob < 0.90:
                child_chromosome.append(gp2)

            else:
                child_chromosome.append(self.mutated_genes())

        return Individual(child_chromosome)

    #Selection operation
    def selection(population):

        population = sorted(population, key = lambda x:x.fitness)
        return population

def main():

    global POPULATION_SIZE

    #Current Generation
    generation = 1

    #Booleans for solution
    found = False
    population = []

    for _ in range(POPULATION_SIZE):
            gnome = Individual.create_gnome()
            population.append(Individual(gnome))

    while not found:

        population = Individual.selection(population)

        if population[0].fitness <= 0:
            found = True
```

```
            break

        #New Generation
        new_generation = []

        #Perform Elitism, selecting 10% fittest from the population
        #This will go to the next generation
        #so we dnt destroy our solution

        s = int((10 * POPULATION_SIZE) / 100)
        new_generation.extend(population[:s])

        s = int((90 * POPULATION_SIZE)/100)
        for _ in range(s):
            parent1 = random.choice(population[:50])
            parent2 = random.choice(population[:50])

            #Perform crossover
            child = parent1.crossover(parent2)

            #Append the new generation
            new_generation.append(child)

            #Population will have the new generation
            population = new_generation

            #Print the generations
            print("Generations: {}\tString: {}\tFitness: {}".\
                format(generation, "".join(population[0].chromosome),
population[0].fitness))

            generation += 1

    #Print the generations
    print("Generations: {}\tString: {}\tFitness: {}".\
        format(generation, "".join(population[0].chromosome),
population[0].fitness))

if _name_== '_main_':
    main()
```

Output:

```
Generations: 643        String: He{lo   Fitness: 1
Generations: 644        String: He{lo   Fitness: 1
Generations: 645        String: He{lo   Fitness: 1
Generations: 646        String: He{lo   Fitness: 1
Generations: 647        String: He{lo   Fitness: 1
Generations: 648        String: He{lo   Fitness: 1
Generations: 649        String: He{lo   Fitness: 1
Generations: 650        String: He{lo   Fitness: 1
Generations: 651        String: He{lo   Fitness: 1
Generations: 652        String: He{lo   Fitness: 1
Generations: 653        String: He{lo   Fitness: 1
Generations: 654        String: He{lo   Fitness: 1
Generations: 655        String: He{lo   Fitness: 1
Generations: 656        String: He{lo   Fitness: 1
Generations: 657        String: He{lo   Fitness: 1
Generations: 658        String: He{lo   Fitness: 1
Generations: 659        String: He{lo   Fitness: 1
Generations: 660        String: He{lo   Fitness: 1
Generations: 661        String: He{lo   Fitness: 1
Generations: 662        String: He{lo   Fitness: 1
Generations: 663        String: He{lo   Fitness: 1
Generations: 664        String: He{lo   Fitness: 1
Generations: 665        String: He{lo   Fitness: 1
Generations: 666        String: He{lo   Fitness: 1
Generations: 667        String: He{lo   Fitness: 1
Generations: 668        String: He{lo   Fitness: 1
Generations: 669        String: He{lo   Fitness: 1
Generations: 670        String: He{lo   Fitness: 1
Generations: 671        String: He{lo   Fitness: 1
Generations: 672        String: He{lo   Fitness: 1
Generations: 673        String: He{lo   Fitness: 1
Generations: 674        String: He{lo   Fitness: 1
Generations: 675        String: He{lo   Fitness: 1
Generations: 676        String: Hello   Fitness: 0
PS C:\Users\raman>
```

Practical 10-B:-

Aim: Implementation of Simple Genetic Algorithm.

Code:

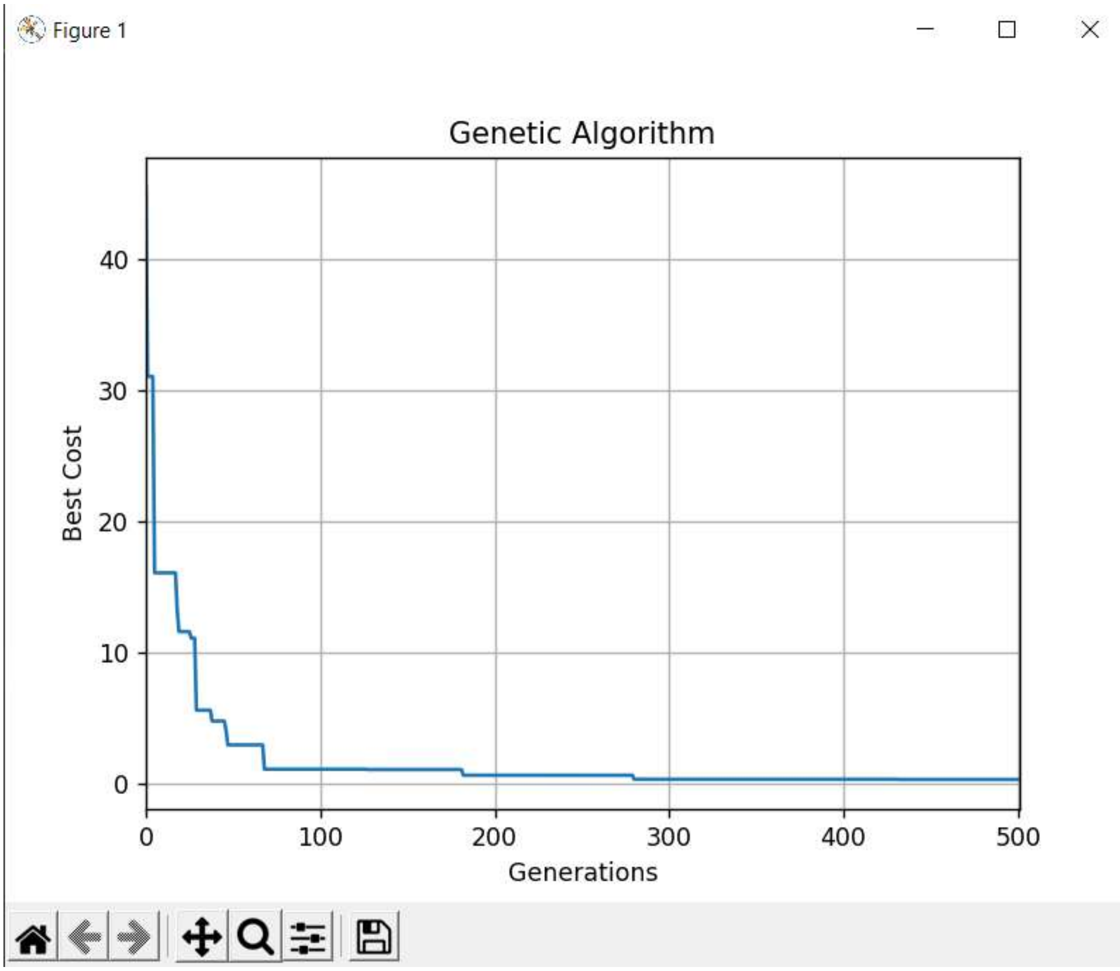
```
import numpy as np
import matplotlib.pyplot as plt
import copy

# cost function
def sphere(x):
  return sum(x**2)

def roulette_wheel_selection(p):
  c = np.cumsum(p)
  r = sum(p) * np.random.rand()
  ind = np.argwhere(r <= c)
  return ind[0][0]

def crossover(p1, p2):
  c1 = copy.deepcopy(p1)
  c2 = copy.deepcopy(p2)

  # Uniform crossover
  alpha = np.random.uniform(0, 1, *(c1['position'].shape))
  c1['position'] = alpha*p1['position'] + (1-alpha)*p2['position']
  c2['position'] = alpha*p2['position'] + (1-alpha)*p1['position']

  return c1, c2

def mutate(c, mu, sigma):
  y = copy.deepcopy(c)
  flag = np.random.rand(*(c['position'].shape)) <= mu  # array of True and
Flase, indicating at which position to perform mutation
  ind = np.argwhere(flag)
  y['position'][ind] += sigma * np.random.randn(*ind.shape)
  return y

def bounds(c, varmin, varmax):
  c['position'] = np.maximum(c['position'], varmin)
  c['position'] = np.minimum(c['position'], varmax)

def sort(arr):
  n = len(arr)
  for i in range(n-1):
    for j in range(0, n-i-1):
            if arr[j]['cost'] > arr[j+1]['cost'] :
                arr[j], arr[j+1] = arr[j+1], arr[j]
    return arr
```

```
def ga(costfunc, num_var, varmin, varmax, maxit, npop, num_children, mu,
sigma, beta):
  population = {}
  for i in
range(npop):                                                      # each
inidivdual has position(chromosomes) and cost,
    population[i] = {'position': None, 'cost':
None}                          # create individual as many as population
size(npop)
  # Best solution found
  bestsol = copy.deepcopy(population)
  bestsol_cost =
np.inf                                                      # initial best
cost is infinity
  # Initialize population - 1st Gen
  for i in range(npop):
      population[i]['position'] = np.random.uniform(varmin, varmax,
num_var)    # randomly initialize the chromosomes and cost
      population[i]['cost'] = costfunc(population[i]['position'])
      if population[i]['cost'] <
bestsol_cost:                            # if cost of an individual is
less(best) than best cost,
        bestsol =
copy.deepcopy(population[i])                               # replace the
best solution with that individual
  # Best cost of each generation/iteration
  bestcost = np.empty(maxit)
  # Main loop
  for it in range(maxit):
    # Calculating probability for roulette wheel selection
    costs = []
    for i in range(len(population)):
      costs.append(population[i]['cost'])
  # list of all the population cost
    costs = np.array(costs)
    avg_cost =
np.mean(costs)                                                 # taking
average of the costs
    if avg_cost != 0:
      costs = costs/avg_cost
    probs = np.exp(-
beta*costs)                                                # probability is
exponensial of -ve beta times costs
    for _ in
range(num_children//2):                                          # we will
be having two off springs for each crossover
```

```
  # hence divide number of children by 2
      # Roulette wheel selection
      p1 = population[roulette_wheel_selection(probs)]
      p2 = population[roulette_wheel_selection(probs)]
      # crossover two parents
      c1, c2 = crossover(p1, p2)
      # Perform mutation
      c1 = mutate(c1, mu, sigma)
      c2 = mutate(c2, mu, sigma)
      # Apply bounds
      bounds(c1, varmin, varmax)
      bounds(c2, varmin, varmax)

      # Evaluate first off spring
      c1['cost'] =
costfunc(c1['position'])                              # calculate cost
function of child 1

      if type(bestsol_cost) == float:
        if c1['cost'] <
bestsol_cost:                                        # replacing best
solution in every generation/iteration
          bestsol_cost = copy.deepcopy(c1)
      else:
        if c1['cost'] <
bestsol_cost['cost']:                                # replacing best
solution in every generation/iteration
          bestsol_cost = copy.deepcopy(c1)


      # Evaluate second off spring
      if c2['cost'] <
bestsol_cost['cost']:                                  # replacing best
solution in every generation/iteration
        bestsol_cost = copy.deepcopy(c2)

    # Merge, Sort and Select
    population[len(population)] = c1
    population[len(population)] = c2

    population = sort(population)

    # Store best cost
    bestcost[it] = bestsol_cost['cost']

    # Show generation information
    print('Iteration {}: Best Cost = {}'. format(it, bestcost[it]))
```

```
  out = population
  Bestsol = bestsol
  bestcost = bestcost
  return (out, Bestsol, bestcost)

# Problem definition
costfunc = sphere
num_var = 5       # number of decicion variables
varmin = -10      # lower bound
varmax = 10       # upper bound

# GA Parameters
maxit = 501                                          # number of
iterations
npop = 20                                            # initial population
size
beta = 1
prop_children = 1                                    # proportion of
children to population
num_children = int(np.round(prop_children * npop/2)*2)    # making sure it
always an even number
mu = 0.2                                             # mutation rate 20%,
205 of 5 is 1, mutating 1 gene
sigma = 0.1                                          # step size of
mutation


# Run GA
out = ga(costfunc, num_var, varmin, varmax, maxit, npop, num_children, mu,
sigma, beta)

# Resultsimport numpy as np
import matplotlib.pyplot as plt import copy

# cost function def sphere(x):
return sum(x**2)

def roulette_wheel_selection(p): c = np.cumsum(p)
r = sum(p) * np.random.rand() ind = np.argwhere(r <= c) return ind[0][0]

def crossover(p1, p2): c1 = copy.deepcopy(p1) c2 = copy.deepcopy(p2)
```

Output:




```
Iteration 484: Best Cost = 0.31722424870157534
Iteration 485: Best Cost = 0.31722424870157534
Iteration 486: Best Cost = 0.31722424870157534
Iteration 487: Best Cost = 0.31722424870157534
Iteration 488: Best Cost = 0.31722424870157534
Iteration 489: Best Cost = 0.31722424870157534
Iteration 490: Best Cost = 0.31722424870157534
Iteration 491: Best Cost = 0.31722424870157534
Iteration 492: Best Cost = 0.31722424870157534
Iteration 493: Best Cost = 0.31722424870157534
Iteration 494: Best Cost = 0.31722424870157534
Iteration 495: Best Cost = 0.31722424870157534
Iteration 496: Best Cost = 0.31722424870157534
Iteration 497: Best Cost = 0.31722424870157534
Iteration 498: Best Cost = 0.31722424870157534
Iteration 499: Best Cost = 0.31722424870157534
Iteration 500: Best Cost = 0.31722424870157534
```