A PRACTICAL REPORT

ON

SOFT COMPUTING TECHNIQUES

SUBMITTED BY

Mr.Shaikh Fahim Amin

UNDER THE GUIDANCE OF

Mrs. SHABANA ANSARI

Submitted in fulfillment of the requirements for qualifying

M.Sc.IT Part-1 Semester-1 Examination 2025-2026

University of Mumbai

Department of Information Technology

R.D. & S.H National College of Arts, Commerce & S.W.A.

Science College Bandra (West), Mumbai – 400 050.

R.D. & S.H. National & S.W.A. Science College

Bandra (W), Mumbai – 400050.

Department of Information Technology

M.Sc.IT (Sem-1)

Certificate

This is to certify that **Soft Computing Techniques Practical** performed at R.D. & S.H. National & S.W.A. Science College by **Ms.Uzma Jamil Khan** holding Seat No. __________ studying Masters of Science in Information Technology Semester – 1 has been satisfactorily completed as prescribed by the University of Mumbai, during the year 2025 – 2026.

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

College Stamp

INDEX

Practical 1

Aim: Design a Simple Neural Network Model.

Practical No : 1-A

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 :

```
========== RESTART: C:/Users/k
Enter value of x: 6
Enter value of weight w: 56
Enter value of bias b: 59
net= 395
output= 1
```

Practical No : 1-B

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

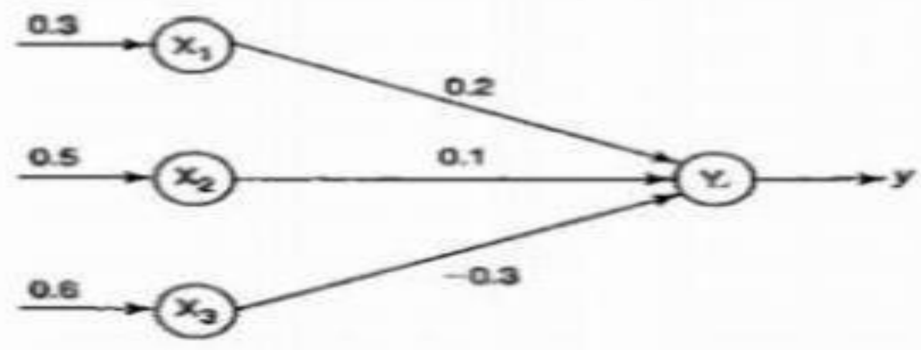


Figure 1 Neural net.

Code :

```
# number of elements as input

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

# Entering the inputs

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)

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 :

```
========= RESTART: C:/Users/kuzma/OneDrive/Desktop/Soft Compu
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

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

Practical No : 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)
```

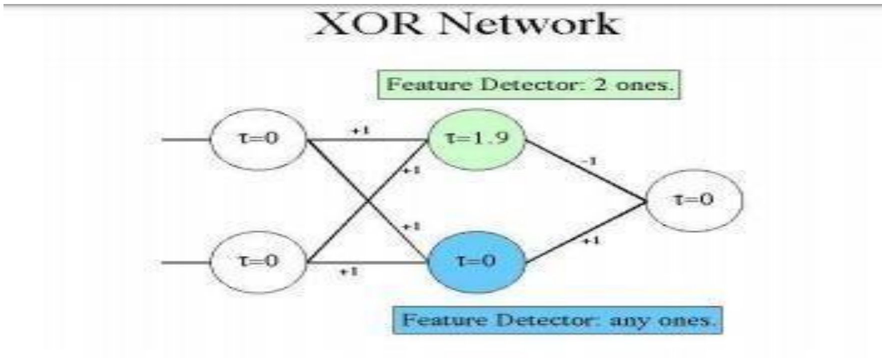
```
# Assume one weight as excitatory and the other as inhibitory
Yin = []
for i in range(0, num_ip):
Yin.append(n[i] - m[i])
print("After assuming one weight as excitatory and the other as inhibitory Yin = ", Yin)
# From the calculated net inputs, now it is possible to fire the neuron for input (1, 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
Y.append(ele)
if Yin[i] < 1:
ele = 0
Y.append(ele)
print("Y = ", Y)
```

OUTPUT :

```
========= RESTART: C:/Users/kuzma/OneDrive/Desktop/Soft Computing/2A.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 No : 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])  # Possibly for XOR logic
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 * w12 + x2 * w22
print("zin1:", zin1)
print("zin2:", zin2)
for i in range(4):
y1[i] = 1 if zin1[i] >= theta else 0
y2[i] = 1 if zin2[i] >= theta else 0
print("y1:", y1)
print("y2:", y2)
zin = y1 * v1 + y2 * v2
for i in range(4):
y[i] = 1 if zin[i] >= theta else 0
print("Final OUTPUT y:", y)
print("Target z:", z)
```

```
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 :

```
========= RESTART: C:\Users\kuzma\OneDrive
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
zin1: [ 0 -1  1  0]
zin2: [ 0  1 -1  0]
y1: [0. 0. 1. 0.]
y2: [0. 1. 0. 0.]
Final output y: [0. 1. 1. 0.]
Target z: [0 1 1 0]
z2 [ 0  1 -1  0]
yin [0. 1. 1. 0.]
Output of Net
y [0 1 1 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
```

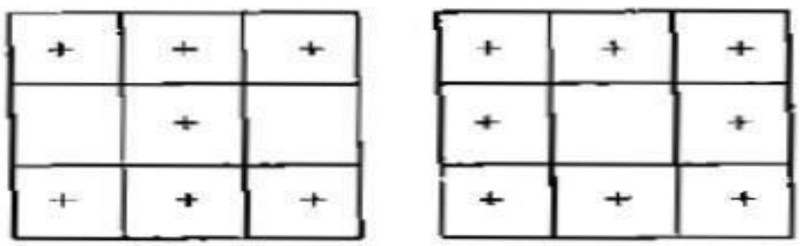
Practical 3

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

Practical No : 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

# Input feature vectors

x1 = np.array([1, 1, 1, -1, -1, 1, 1, 1, 1])

x2 = np.array([1, 1, 1, -1, 1, 1, 1, 1, 1])

# Target values for inputs

y = np.array([1, -1])

# Initial weight and bias setup

b = 0

wtold = np.zeros((9,))

wtnew = np.zeros((9,))

wtnew = wtnew.astype(int)
```

```
wtold = wtold.astype(int)
# Bias for display purposes
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 :

```
========= RESTART: C:/Users/kuzma/OneDrive/
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  0 -2  0  0  0  0]
Bias value 0
```

Practical No : 3-B

Aim : Write a program to implement Delta Rule.

Code :

```
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 :

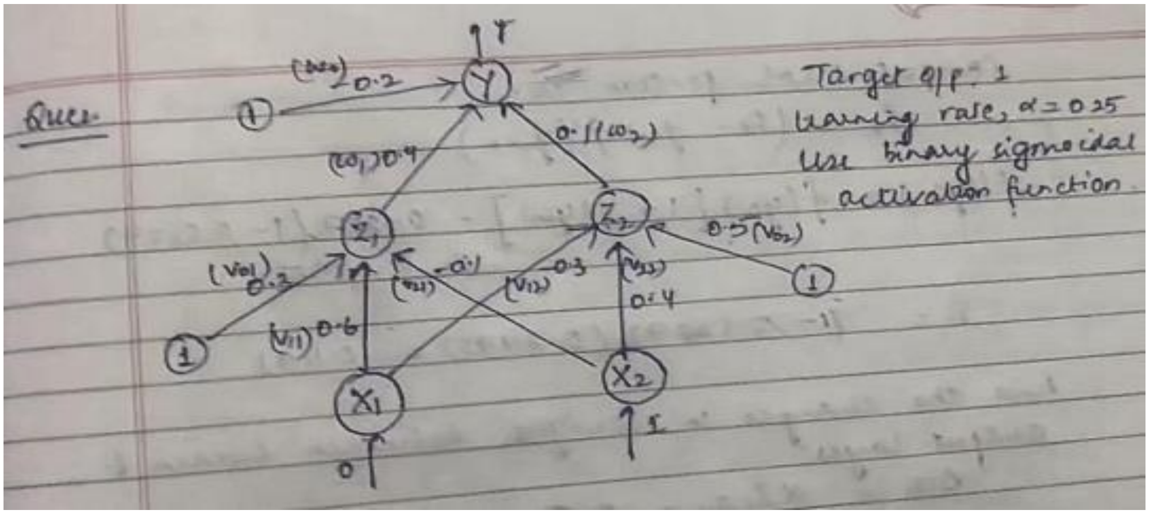
```
========= RESTART: C:/Users/kuzma/C
Initial inputs: 1
Initial inputs: 1
Initial inputs: 1
Initial weights: 1
Initial weights: 1
Initial weights: 1
Desired output: 2
Desired output: 3
Desired output: 4
Enter learning rate: 1
actual: [1. 1. 1.]
desired: [2. 3. 4.]
weights: [2. 3. 4.]
actual: [2. 3. 4.]
desired: [2. 3. 4.]
******************************
Final output
Corrected weights: [2. 3. 4.]
actual: [2. 3. 4.]
desired: [2. 3. 4.]
```

Practical 4

Aim : Write a program for Back Propagation Algorithm.

Practical No : 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(f"din1 = {din1}\ndin2 = {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 :

```
========= RESTART: C:/Users/kuzma/OneDrive/Desktop/Soft
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 No : 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 firstn/ww11=", w11)

print("The uploaded weight of secondn/ww21=", w21)

print("The updated base of firstn/wb10=", b10)

print("The updated base of secondn/wb20=", b20)

OUTPUT :

```
========= RESTART: C:/Users/kuzma/OneDrive/
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 firstn/ww11= 12.0
The uploaded weight of secondn/ww21= 23.0
The updated base of firstn/wb10= 35.0
The updated base of secondn/wb20= 45.0
```

Practical 5

Aim : Write a program for Hopfield Network.

Practical No : 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")

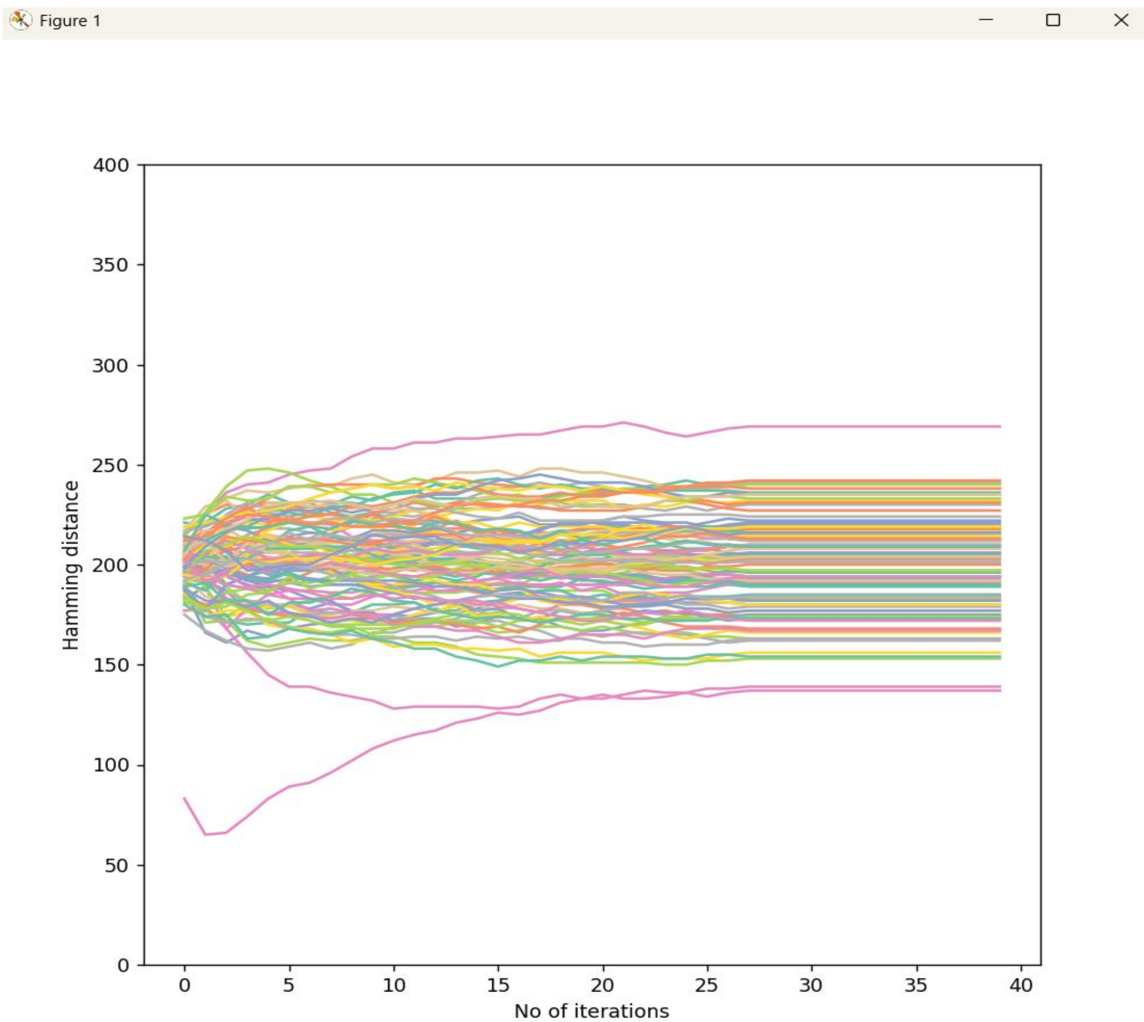
plt.ylim([0, N])

plt.show()

plt.show(block=True)

OUTPUT :

```
========== RESTART: C:\Users\kuzma\Or
(100, 400)
67
```



Practical No : 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

from scipy.interpolate import interp1d


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)
```

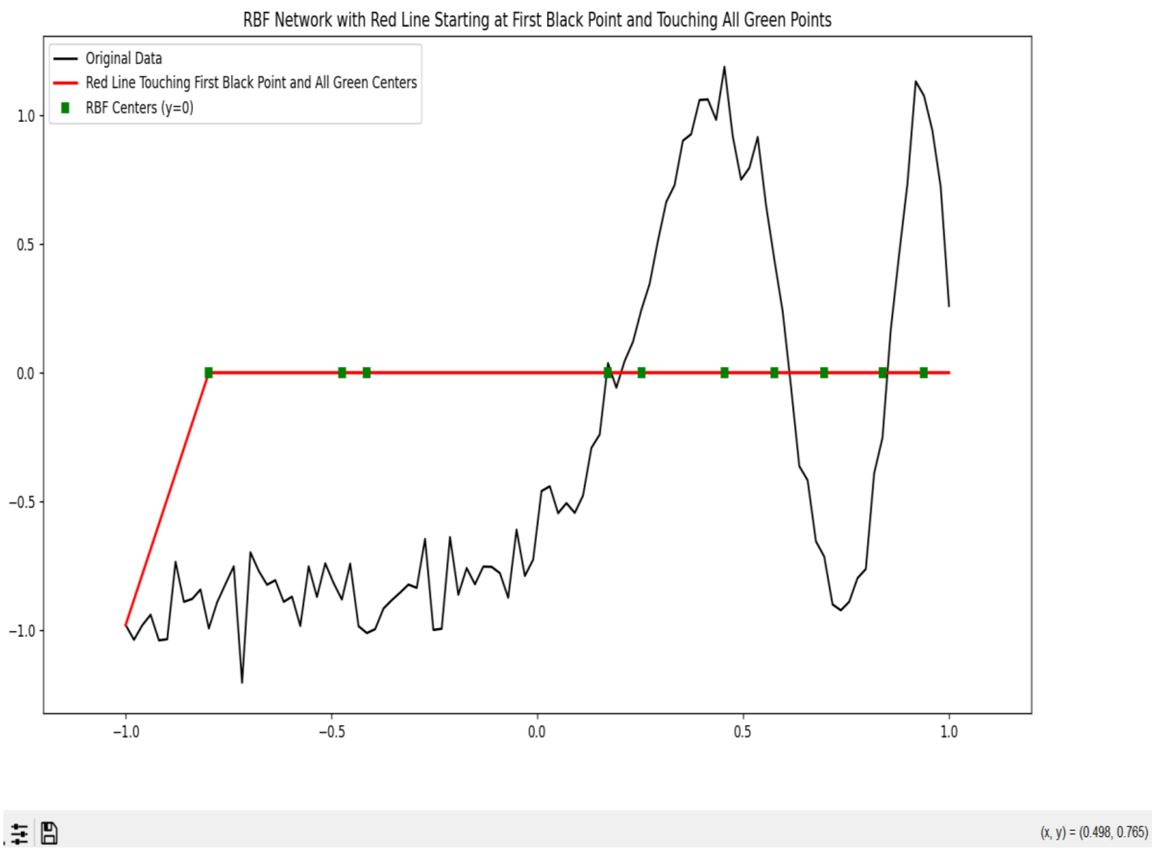
```
def _calcAct(self, X):

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):

rnd_idx = random.permutation(X.shape[0])[:self.numCenters]

self.centers = [X[i, :] for i in rnd_idx]

print("center", self.centers)

G = self._calcAct(X)

self.W = np.dot(pinv(G), Y)

def test(self, X):

G = self._calcAct(X)

Y = np.dot(G, self.W)

return Y


if __name__ == "__main__":

n = 100

x = np.mgrid[-1:1:complex(0, n)].reshape(n, 1)

y = np.sin(3 * (x + 0.5) ** 3 - 1)

y += random.normal(0, 0.1, y.shape)

rbf = RBF(1, 10, 1)
```

```
rbf.train(x, y)

z_rbf = rbf.test(x)

# Extract centers' x positions and y=0

centers_x = np.array(rbf.centers).flatten()

centers_y = np.zeros_like(centers_x)


# Start with first black data point (x[0], y[0])

start_x = x[0, 0]

start_y = y[0, 0]

# Combine start point with centers

all_x = np.concatenate(([start_x], centers_x))

all_y = np.concatenate(([start_y], centers_y))


# Sort points by x for interpolation

sorted_indices = np.argsort(all_x)

all_x_sorted = all_x[sorted_indices]

all_y_sorted = all_y[sorted_indices]

# Create piecewise linear interpolation function

interp_func = interp1d(all_x_sorted, all_y_sorted, kind='linear', fill_value="extrapolate")


# Evaluate on full x grid

z_interp = interp_func(x.flatten())

plt.figure(figsize=(12, 8))

plt.plot(x, y, "k-", label="Original Data")
```

```
plt.plot(x, z_interp, "r-", linewidth=2, label="Red Line Touching First Black Point and All Green Centers")

plt.plot(centers_x, centers_y, "gs", label="RBF Centers (y=0)")

plt.xlim(-1.2, 1.2)

plt.legend()

plt.title("RBF Network with Red Line Starting at First Black Point and Touching All Green Points")

plt.show()
```

OUTPUT :

```
========= RESTART: C:/Users/kuzma/OneDrive/Desktop/Soft Computing/5B.py ========
center [array([0.45454545]), array([0.83838384]), array([0.57575758]), array([-0.7979798
]), array([-0.41414141]), array([0.25252525]), array([0.17171717]), array([0.93939394]),
array([0.6969697]), array([-0.47474747])]
```



Practical 6

Aim : Write a program for Self-Organising Maps.

Practical No : 6-A

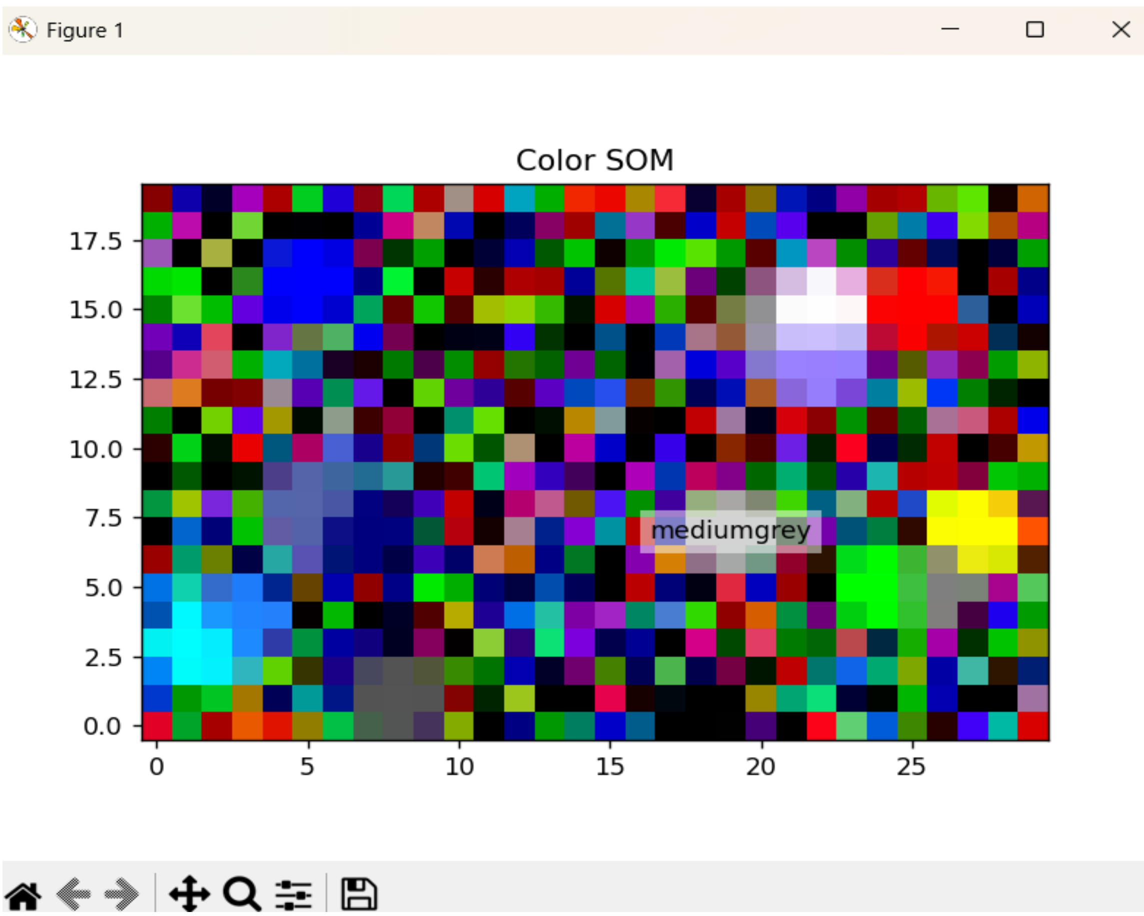
Aim : Write a program for Self-Organising Maps.

Code :

```
import numpy as np

from minisom import MiniSom

import matplotlib.pyplot as plt

colors = np.array([[0., 0., 0.],[0., 0., 1.],[0., 0., 0.5],[0.125, 0.519, 1.0],[0.33, 0.4, 0.67],

[0.6, 0.5, 1.0],[0., 1., 0.],[1., 0., 0.],[0., 1., 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 = MiniSom(20, 30, 3, sigma=1.0, learning_rate=0.5)

som.train(colors, 1000)

plt.imshow(som.get_weights()[:, :, :3], origin='lower')

plt.title('Color SOM')


for i, color in enumerate(colors):

w = som.winner(color)

plt.text(w[1], w[0], color_names[i], ha='center', va='center', bbox=dict(facecolor='white',
alpha=0.5, lw=0))

plt.show()
```

OUTPUT :

```
========= RESTART: C:/Users/kuzma/OneDrive/Desktop/Soft Computing/6A.py ===
=====
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers). Got range [-0.9876188274163928..0.9999999
99999957].
```



Practical No : 6-B

Aim : Write a program for Adaptive Resonance Theory.

Code :

```
import numpy as np

VIGILANCE = 0.6

LEARNING_COEF = 0.5

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

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],])


L1_neurons_cnt = len(train[0])

L2_neurons_cnt = 1

bottomUps = np.array([[1/(L1_neurons_cnt + 1) for _ in range(L1_neurons_cnt)]], np.float64)

topDowns = np.array([[1 for _ in range(L1_neurons_cnt)]], np.float64)


for tv in train:

print("\nTrain 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)

tv_sum = sum(tv)

if tv_sum == 0:

similarity = 0


else:

similarity = topDowns[winner_neuron_idx].dot(tv) / tv_sum

print(" TopDowns[Winner]: ", topDowns[winner_neuron_idx])

print(" BottomUps Weights: ", bottomUps[winner_neuron_idx])

print(" Similarity: ", similarity)


if similarity >= VIGILANCE:

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:

OUTPUTs[winner_neuron_idx] = -1

counter -= 1


if createNewNeuron:

print(" Creating a new neuron")

new_bottom_weights = np.array([[i / (LEARNING_COEF + sum(tv)) for i in tv]], np.float64)
```

```
new_top_weights = np.array([[i for i in tv]], np.float64)
print(" Weights bottomUps: ", new_bottom_weights)
print(" Weights topDown: ", 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 test vector {tv}")
```

OUTPUT :

```
========= RESTART: C:/Users/kuzma/OneDrive/Desktop/Soft Computing/6B.py ========

Train Vector:  [1 0 0 0 0 0]
 TopDowns[Winner]:  [1. 1. 1. 1. 1. 1.]
 BottomUps Weights:  [0.14285714 0.14285714 0.14285714 0.14285714 0.14285714 0.14285714]
 Similarity:  1.0
=======
Total Classes: 1
Center of masses
[[1. 0. 0. 0. 0. 0.]]

Train Vector:  [1 1 1 1 1 0]
 TopDowns[Winner]:  [1. 0. 0. 0. 0. 0.]
 BottomUps Weights:  [0.66666667 0.         0.         0.         0.         ]
 Similarity:  0.2
 Creating a new neuron
 Weights bottomUps:  [[0.18181818 0.18181818 0.18181818 0.18181818 0.18181818 0.        ]]
 Weights topDown:  [[1. 1. 1. 1. 1. 0.]]
=======
Total Classes: 2
Center of masses
[[1. 0. 0. 0. 0. 0.]
 [1. 1. 1. 1. 1. 0.]]

Train Vector:  [1 0 1 0 1 0]
 TopDowns[Winner]:  [1. 0. 0. 0. 0. 0.]
 BottomUps Weights:  [0.66666667 0.         0.         0.         0.         ]
 Similarity:  0.3333333333333333
 TopDowns[Winner]:  [1. 1. 1. 1. 1. 0.]
 BottomUps Weights:  [0.18181818 0.18181818 0.18181818 0.18181818 0.18181818 0.        ]
 Similarity:  1.0
=======
Total Classes: 2
Center of masses
[[1. 0. 0. 0. 0. 0.]
 [1. 0. 1. 0. 1. 0.]]
```

```
Train Vector:  [0 0 1 1 1 0]
 TopDowns[Winner]:  [1. 0. 1. 0. 0. 0.]
 BottomUps Weights:  [0.4 0.  0.4 0.  0.  0. ]
 Similarity:  0.3333333333333333
 TopDowns[Winner]:  [0. 1. 0. 0. 1. 1.]
 BottomUps Weights:  [0.         0.28571429 0.         0.         0.28571429 0.28571429]
 Similarity:  0.3333333333333333
 TopDowns[Winner]:  [1. 0. 0. 0. 0. 0.]
 BottomUps Weights:  [0.66666667 0.         0.         0.         0.         ]
 Similarity:  0.0
 Creating a new neuron
 Weights bottomUps:  [[0.         0.         0.28571429 0.28571429 0.28571429 0.        ]]
 Weights topDown:  [[0. 0. 1. 1. 1. 0.]]
=======
Total Classes: 4
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.]]

Train Vector:  [1 1 1 1 1 0]
 TopDowns[Winner]:  [0. 0. 1. 1. 1. 0.]
 BottomUps Weights:  [0.         0.         0.28571429 0.28571429 0.28571429 0.        ]
 Similarity:  0.6
=======
Total Classes: 4
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.]]

Train Vector:  [1 1 1 1 1 0]
 TopDowns[Winner]:  [0. 0. 1. 1. 1. 0.]
 BottomUps Weights:  [0.         0.         0.28571429 0.28571429 0.28571429 0.        ]
 Similarity:  0.6
```

```
 Similarity:  0.6
=======
Total Classes: 4
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.]]

Train Vector:  [1 1 1 1 1 1]
 TopDowns[Winner]:  [0. 1. 0. 0. 1. 1.]
 BottomUps Weights:  [0.         0.28571429 0.         0.         0.28571429 0.28571429]
 Similarity:  0.5
 TopDowns[Winner]:  [0. 0. 1. 1. 1. 0.]
 BottomUps Weights:  [0.         0.         0.28571429 0.28571429 0.28571429 0.        ]
 Similarity:  0.5
 TopDowns[Winner]:  [1. 0. 1. 0. 0. 0.]
 BottomUps Weights:  [0.4 0.  0.4 0.  0.  0. ]
 Similarity:  0.3333333333333333
 TopDowns[Winner]:  [1. 0. 0. 0. 0. 0.]
 BottomUps Weights:  [0.66666667 0.         0.         0.         0.         0.        ]
 Similarity:  0.16666666666666666
 Creating a new neuron
 Weights bottomUps:  [[0.15384615 0.15384615 0.15384615 0.15384615 0.15384615 0.15384615]]
 Weights topDown:  [[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 test vector [1 1 1 1 1 1]
Class 3 for test vector [1 1 1 1 1 0]
Class 1 for test vector [1 1 1 1 0 0]
Class 1 for test vector [1 1 1 0 0 0]
Class 0 for test vector [1 1 0 0 0 0]
Class 0 for test vector [1 0 0 0 0 0]
Class 0 for test vector [0 0 0 0 0 0]
```

Practical 7

Aim : Line Separation

Practical No : 7-A

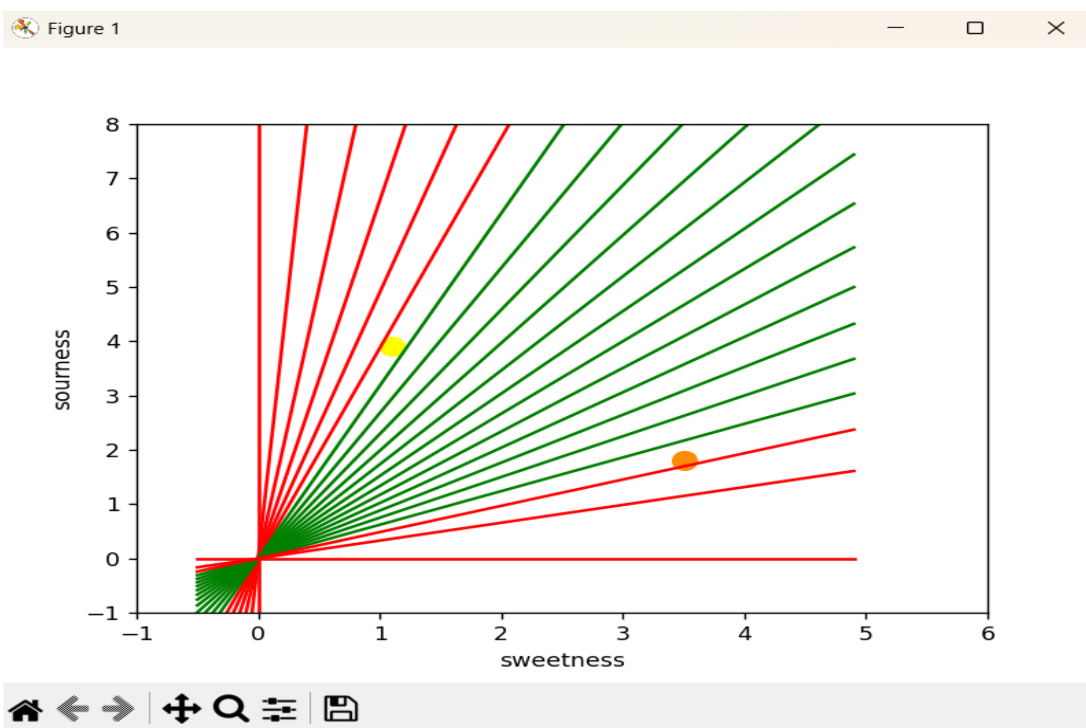
Aim : Write a program for Line Separation

Code :

```
import numpy as np

import matplotlib.pyplot as plt

def create_distance_function(a, b, c):

def distance(x, y):

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.abs(nom) / np.sqrt(a**2 + b**2), pos)
return distance

def main():
points = [(3.5, 1.8), (1.1, 3.9)]  # Example points
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)  # X values to plot line
# Plot points
size = 10
for index, (x, y) in enumerate(points):
color = "darkorange" if index == 0 else "yellow"
ax.plot(x, y, "o", color=color, markersize=size)

# Define the step for line plotting and slope generation
step = 0.05
for x_param in np.arange(0, 1 + step, step):
slope = np.tan(np.arccos(x_param))  # Slope based on x (parameterizing a line)
dist4line1 = create_distance_function(slope, -1, 0)  # Create distance function
```

```
# Generate Y values for line
Y = slope * X
# Initialize the results list
results = []
# Get results for the points
for point in points:
results.append(dist4line1(*point))  # Call the distance function for each point
print(slope, results)

# Check if the points are on different sides of the line
if results[0][1] != results[1][1]:
ax.plot(X, Y, "g-")  # Green line if points are separated
else:
ax.plot(X, Y, "r-")  # Red line if points are not separated
# Show plot
plt.show()
if __name__ == "__main__":
main()
```

OUTPUT :



Practical No : 7-B

Aim : Write a program for Hopfield network model for associative memory.

Code :

```
import numpy as np

import matplotlib.pyplot as plt

# === Pattern creation functions ===

def create_checkerboard(size):

return np.indices((size, size)).sum(axis=0) % 2

def create_random_pattern(size, on_prob=0.5):

return (np.random.rand(size, size) < on_prob).astype(int)

def create_random_pattern_list(size, nr_patterns, on_prob=0.5):

return [create_random_pattern(size, on_prob) for _ in range(nr_patterns)]

def plot_pattern_list(pattern_list, titles=None):

n = len(pattern_list)

plt.figure(figsize=(n*2, 2))


for i, p in enumerate(pattern_list):

plt.subplot(1, n, i+1)

plt.imshow(p, cmap='gray_r')

plt.axis('off')

if titles:

plt.title(titles[i])

plt.show()

def compute_overlap(p1, p2):
```

```
# patterns are binary 0/1 -> convert to bipolar -1/+1 for overlap calc
p1_bipolar = 2 * p1.flatten() - 1
p2_bipolar = 2 * p2.flatten() - 1
return np.dot(p1_bipolar, p2_bipolar) / len(p1_bipolar)
def compute_overlap_matrix(pattern_list):
n = len(pattern_list)
mat = np.zeros((n,n))
for i in range(n):
for j in range(n):
mat[i,j] = compute_overlap(pattern_list[i], pattern_list[j])
return mat


def plot_overlap_matrix(mat):
plt.imshow(mat, cmap='coolwarm', vmin=-1, vmax=1)
plt.colorbar()
plt.title("Overlap matrix")
plt.show()
# === Hopfield network class ===
class HopfieldNetwork:
def __init__(self, nr_neurons):
self.nr_neurons = nr_neurons
self.weights = np.zeros((nr_neurons, nr_neurons))
def train(self, patterns):
# Convert patterns to bipolar vectors (-1,1)
patterns_bipolar = [2 * p.flatten() - 1 for p in patterns]
```

```
for p in patterns_bipolar:
self.weights += np.outer(p, p)
np.fill_diagonal(self.weights, 0)
self.weights /= len(patterns)
def recall(self, state, steps=5):
s = 2 * state.flatten() - 1
states = [s.copy()]
for _ in range(steps):
s = np.sign(self.weights @ s)

s[s==0] = 1  # treat zeros as 1
states.append(s.copy())
return states
def state_to_pattern(self, state, shape):
# Convert bipolar state back to binary pattern
return ((state.reshape(shape) + 1) // 2).astype(int)
# === Utility ===
def flip_n_bits(pattern, n):
p_flat = pattern.flatten()
indices = np.random.choice(len(p_flat), size=n, replace=False)
p_flat[indices] = 1 - p_flat[indices]  # flip bits
return p_flat.reshape(pattern.shape)
# === Main script ===
pattern_size = 5
checkerboard = create_checkerboard(pattern_size)
```

```
pattern_list = [checkerboard]

pattern_list.extend(create_random_pattern_list(pattern_size, 3, on_prob=0.5))

# Plot original patterns

plot_pattern_list(pattern_list, titles=["Checkerboard"] + [f"Random {i+1}" for i in range(3)])

# Overlap matrix

overlap_matrix = compute_overlap_matrix(pattern_list)

plot_overlap_matrix(overlap_matrix)


# Train Hopfield network

hp_net = HopfieldNetwork(nr_neurons=pattern_size**2)

hp_net.train(pattern_list)

# Flip 4 bits in checkerboard pattern (add noise)

noisy_init = flip_n_bits(checkerboard, 4)

# Run recall (network dynamics)

states = hp_net.recall(noisy_init, steps=6)

# Convert states to patterns for plotting

states_as_patterns = [hp_net.state_to_pattern(s, (pattern_size, pattern_size)) for s in states]

# Plot recall sequence

plot_pattern_list(states_as_patterns, titles=[f"Step {i}" for i in range(len(states_as_patterns))])
```

OUTPUT :


Figure 1
Checkerboard
Random 1
Random 2
Random 3

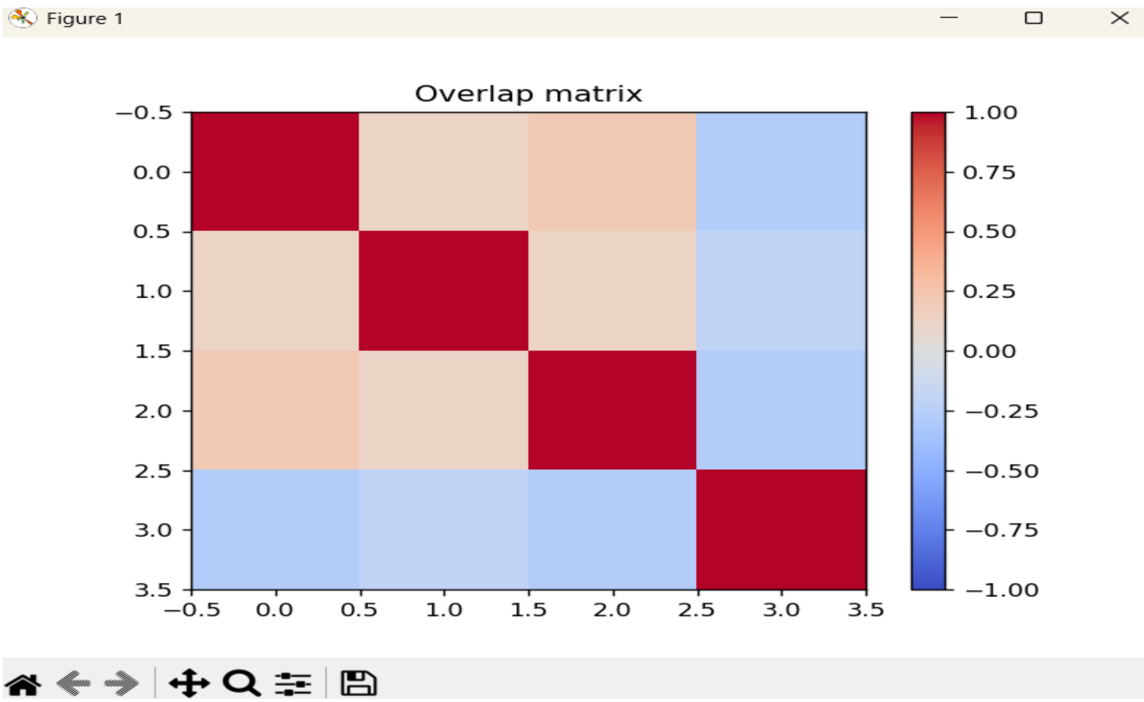


Figure 1
Overlap matrix
−0.5
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00

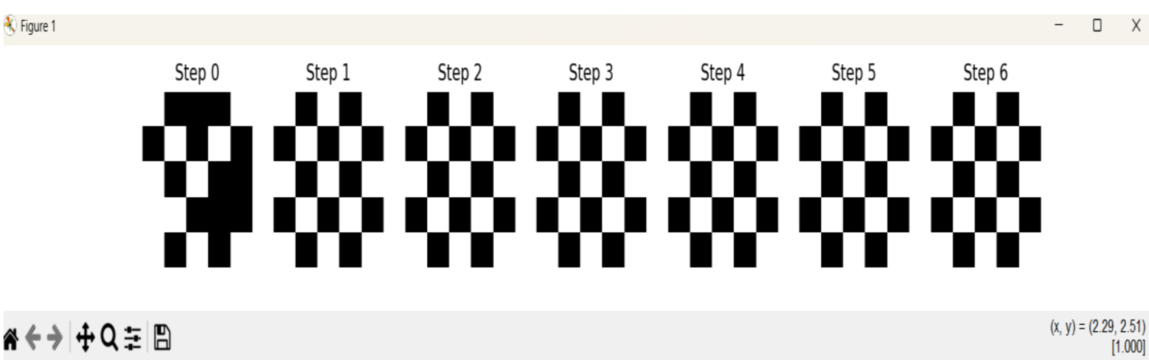


Figure 1
Step 0
Step 1
Step 2
Step 3
Step 4
Step 5
Step 6
(x, y) = (2.29, 2.51)
[1.000]

Practical 8

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

Practical No : 8-A

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

Code :

(in) :

```
def overlapping(list1, list2):
for i in list1:
for j in list2:
if i == j:
return 1
return 0  # Only return 0 if no overlapping element found

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) :

```
======= RESTART: C:/Users/kuzma/OneDrive/Desktop/Soft Computing/8A_in.py
Overlapping
```

(Not in) :

```
======= RESTART: C:/Users/kuzma/OneDrive/Desktop/
Not overlapping
```

Practical No : 8-B

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

Code :

(Is) :

```
x= 5
if(type(x) is not float):
print("true")
else:
print("false")
```

(Is not) :

```
y = 5.2
if(type(y) is not float):
print("true")
else:
print("false")
```

OUTPUT :

(is):

```
========= RESTART: C:/Users/kuzma/OneDrive/Desktop/Sc
true
```

(Is not):

```
========= RESTART: C:/Users/kuzma/OneDrive/Desktop/Soft Computing/8B.py
false
```

Practical 9

Aim : Find the ratios using Fuzzy Logic.

Practical No : 9-A

Aim : Find the ratios using Fuzzy Logic.

Code :

```
from fuzzywuzzy import fuzz
from fuzzywuzzy import process
s1 = "I love fuzzyforfuzzys"
s2 = "I am loveing fuzzyforfuzzys"

print ("\nFuzzywuzzy Ratio:", fuzz.ratio(s1,s2))
print ("\nFuzzywuzzyParialRatio:" ,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))

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

OUTPUT :

```
========= RESTART: C:/Users/kuzma/OneDrive/Desktop/Soft Computing/9A.py

Fuzzywuzzy Ratio: 88

FuzzywuzzyParialRatio: 86

FuzzywuzzyTokenSortRatio: 88

FuzzywuzzyTokenSetRatio: 88

FuzzywuzzyWRatio: 88
list of ratio:
[('fuzzy for fuzzy', 97), ('fuzzy fuzzy', 95), ('g. for fuzzys', 86)]

best among the above list: ('fuzzy for fuzzy', 97)
```

Practical No : 9-B

Aim : Solve Tipping Problem using Fuzzy Logic.

Code :

```
import numpy as np

import skfuzzy as fuzz

from skfuzzy import control as ctrl

import matplotlib.pyplot as plt


# Create fuzzy variables

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')

# Automatically generate fuzzy membership functions quality.automf(3)

service.automf(3)


# Define custom membership functions for 'tip' 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])

# Visualize the membership functions for the quality, service, and tip quality['average'].view()

service.view()

tip.view()

# Keep the plots open

 plt.show()
```

OUTPUT :

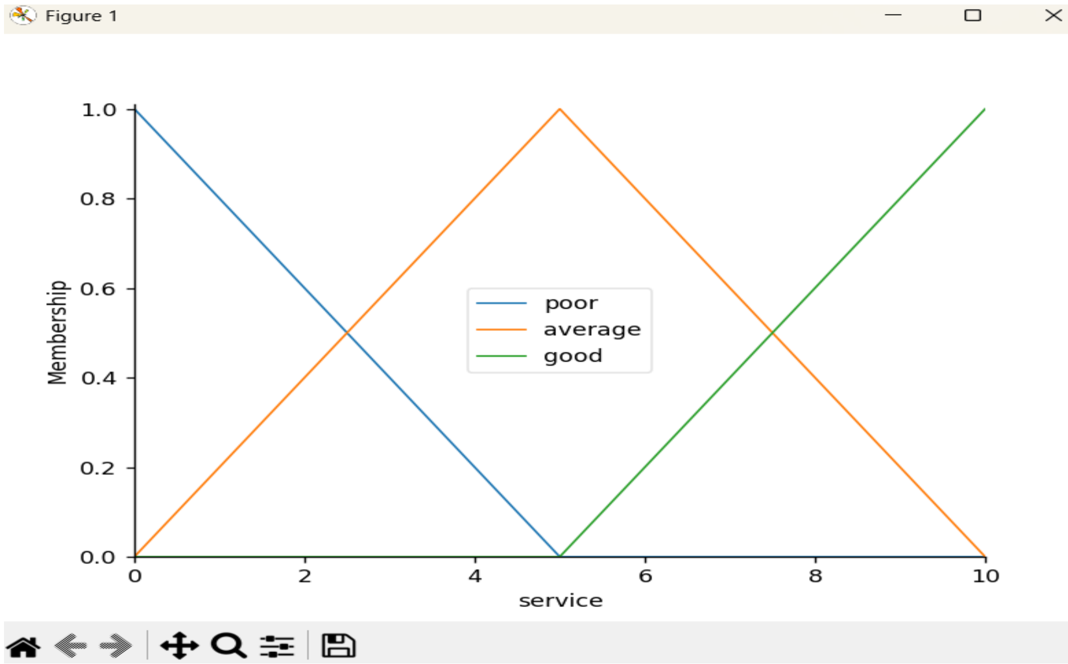


Figure 1
1.0
0.8
0.6
0.4
0.2
0.0
Membership
poor
average
good
0
2
4
6
8
10
service

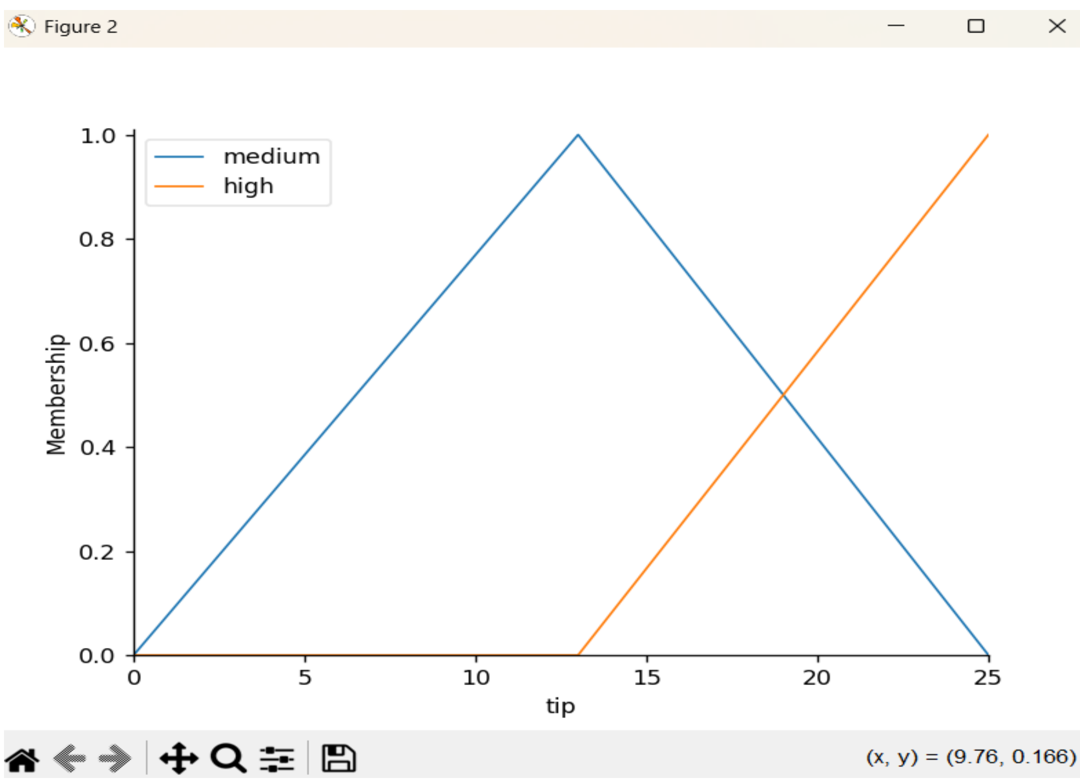


Figure 2
1.0
0.8
0.6
0.4
0.2
0.0
Membership
medium
high
0
5
10
15
20
25
tip
(x, y) = (9.76, 0.166)

Practical 10

Aim : Implementation of Simple Genetic Algorithm.

Practical No : 10-A

Aim : Implementation of Simple Genetic Algorithm.

Code :

```
import random

POPULATION_SIZE=250

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

TARGET="HELLO"


class Individual(object):

def __init__(self,chromosome):

self.chromosome=chromosome

self.fitness=self.cal_fitness()

Classmethod


@staticmethod

def mutated_genes():

global GENES

gene = random.choice(GENES)

return gene


@classmethod

def create_gnome(self):

global TARGET

gnome_len = len(TARGET)
```

```
return [Individual.mutated_genes() for _ in range(gnome_len)]

def cal_fitness(self):

global TARGET

fitness = 0

for gs, gt in zip(self.chromosome, TARGET):

if gs != gt: fitness += 1

return fitness


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)


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 = []
#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:  642          String: H.LLO   Fitness: 1
Generations:  643          String: H.LLO   Fitness: 1
Generations:  644          String: H.LLO   Fitness: 1
Generations:  645          String: H.LLO   Fitness: 1
Generations:  646          String: H.LLO   Fitness: 1
Generations:  647          String: H.LLO   Fitness: 1
Generations:  648          String: H.LLO   Fitness: 1
Generations:  649          String: H.LLO   Fitness: 1
Generations:  650          String: H.LLO   Fitness: 1
Generations:  651          String: H.LLO   Fitness: 1
Generations:  652          String: H.LLO   Fitness: 1
Generations:  653          String: H.LLO   Fitness: 1
Generations:  654          String: H.LLO   Fitness: 1
Generations:  655          String: H.LLO   Fitness: 1
Generations:  656          String: H.LLO   Fitness: 1
Generations:  657          String: H.LLO   Fitness: 1
Generations:  658          String: H.LLO   Fitness: 1
Generations:  659          String: H.LLO   Fitness: 1
Generations:  660          String: H.LLO   Fitness: 1
Generations:  661          String: H.LLO   Fitness: 1
Generations:  662          String: H.LLO   Fitness: 1
Generations:  663          String: H.LLO   Fitness: 1
Generations:  664          String: H.LLO   Fitness: 1
Generations:  665          String: H.LLO   Fitness: 1
Generations:  666          String: H.LLO   Fitness: 1
Generations:  667          String: H.LLO   Fitness: 1
Generations:  668          String: H.LLO   Fitness: 1
Generations:  669          String: H.LLO   Fitness: 1
Generations:  670          String: H.LLO   Fitness: 1
Generations:  671          String: H.LLO   Fitness: 1
Generations:  672          String: H.LLO   Fitness: 1
Generations:  673          String: H.LLO   Fitness: 1
Generations:  674          String: H.LLO   Fitness: 1
Generations:  675          String: H.LLO   Fitness: 1
Generations:  676          String: HELLO   Fitness: 0
```

Practical No : 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 np.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, size=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  # Mutation mask
y['position'][flag] += sigma * np.random.randn(np.sum(flag))
return y

def bounds(c, varmin, varmax):
c['position'] = np.maximum(c['position'], varmin)
c['position'] = np.minimum(c['position'], varmax)
def sort_population(arr):
# Sort population by cost ascending
return sorted(arr, key=lambda x: x['cost'])
def ga(costfunc, num_var, varmin, varmax, maxit, npop, num_children, mu, sigma, beta):
# Initial population
population = []
```

```
for _ in range(npop):
ind = {}
ind['position'] = np.random.uniform(varmin, varmax, num_var)
ind['cost'] = costfunc(ind['position'])
population.append(ind)
# Sort initial population
population = sort_population(population)
# Best solution
bestsol = copy.deepcopy(population[0])
# Array to hold best cost values
bestcost = np.zeros(maxit)
for it in range(maxit):
# Calculate selection probabilities based on cost (lower cost = higher prob)
costs = np.array([ind['cost'] for ind in population])

# To avoid zero probabilities, add a small constant and invert costs
max_cost = np.max(costs)
fitness = np.exp(-beta * costs / max_cost)
probs = fitness / np.sum(fitness)
children = []
# Number of children must be even
for _ in range(num_children // 2):
# Selection

p1 = population[roulette_wheel_selection(probs)]
p2 = population[roulette_wheel_selection(probs)]
# Crossover
c1, c2 = crossover(p1, p2)
# Mutation
c1 = mutate(c1, mu, sigma)
c2 = mutate(c2, mu, sigma)
# Apply bounds
bounds(c1, varmin, varmax)
bounds(c2, varmin, varmax)
# Evaluate cost
c1['cost'] = costfunc(c1['position'])
c2['cost'] = costfunc(c2['position'])
children.append(c1)
children.append(c2)

# Merge population and children
```

```
population.extend(children)
# Sort combined population and select the best npop individuals
population = sort_population(population)[:npop]
# Update best solution found so far

if population[0]['cost'] < bestsol['cost']:
bestsol = copy.deepcopy(population[0])
bestcost[it] = bestsol['cost']
# Print iteration info every 50 iterations
if (it % 50 == 0) or (it == maxit - 1):
print(f"Iteration {it}: Best Cost = {bestcost[it]}")
return population, bestsol, bestcost

# Problem definition
costfunc = sphere
num_var = 5  # Number of decision variables
varmin = -10  # Lower bound
varmax = 10   # Upper bound
# GA Parameters
maxit = 501  # Number of iterations
npop = 20    # Population size
beta = 1     # Selection pressure

prop_children = 1  # Proportion of children to population size
num_children = int(np.round(prop_children * npop / 2) * 2)  # Even number of children
mu = 0.2     # Mutation rate
sigma = 0.1  # Mutation step size

# Run GA

population, bestsol, bestcost = ga(costfunc, num_var, varmin, varmax, maxit, npop,
num_children, mu, sigma, beta)
# Plot results
plt.plot(bestcost)
plt.xlim(0, maxit)
plt.xlabel('Generations')
plt.ylabel('Best Cost')
plt.title('Genetic Algorithm Optimization')
plt.grid(True)
plt.show()
```

OUTPUT :

```
======== RESTART: C:\Users\kuzma\OneDrive\Desktop\Soft Computing\10_B.py
Iteration 0: Best Cost = 38.84355800416189
Iteration 50: Best Cost = 0.001244405006380954
Iteration 100: Best Cost = 3.962687796361074e-05
Iteration 150: Best Cost = 1.1053871493938471e-05
Iteration 200: Best Cost = 4.670447897161459 6e-07
Iteration 250: Best Cost = 4.666143382418269e-07
Iteration 300: Best Cost = 4.666131714190804e-07
Iteration 350: Best Cost = 4.0059227614810357e-07
Iteration 400: Best Cost = 4.0059227614810357e-07
Iteration 450: Best Cost = 4.0059227614810357e-07
Iteration 500: Best Cost = 3.962150780658552e-07
```