Soft Computing Techniques

JNAN VIKAS MANDAL'S

Mohanlal Raichand Mehta College of Commerce
Diwali Maa College of Science
Amritlal Raichand Mehta College of Arts
Dr. R.T. Doshi College of Computer Science
NAAC Re-Accredited Grade 'A+' (CGPA : 3.31) (3rd Cycle)

DEPARTMENT OF INFORMATION TECHNOLOGY

CERTIFICATE

This is to certify that Janhavi Vijay Kharkar bearing seat no. 1310657 has done the project work/journal work in the subject of Soft Computing of semester 1 Practical Examination during the academic year 2023-24 under the guidance of **Mrs. Sharayu Kadam** being the partial requirement for the fulfilment of the curriculum of Degree in Master of Science in Information Technology under University of Mumbai.

Place: Airoli Date:

Sign of Subject in- Charge Sign of External Examiner

Sign of Coordinator

Table of Content

Practical 1A

Aim: Design a simple linear neural network model.

Code:

```
w=float(input("Enter the weight value: "))
b=float(input("Enter the bias value: "))
x=float(input("Enter the input value: "))
yin=float(b+(w*x))
print("The net input yin is:",yin)
```

Output:

```
= RESTART: C:/Users/Janhavi/AppData/Local/Programs/Python/Python38/RunAllInOneGo.py
Enter the weight value: 1
Enter the bias value: 1
Enter the input value: 0.4
The net input yin is: 1.4
>>>
```

Practical 1B

Aim: Calculate the output of neural net using both binary and bipolar sigmoidal function.

Code:

```
import math
bias = float(input("Enter the value of bias: "))
n = int(input("Enter the number of input neurons: "))
w = [ ]
x = [ ]
for i in range(0,n):
    a = float(input("Enter the input: "))
 x.append(a)

    b = float(input("Enter the weight: "))
```

```
    w.append(b)
print("The given weights are: ")
 print(w)
print("The given input are: " )
 print(x)
 y = bias
 for i in range(0, n):
      y = y + (w[i]*x[i])
 print("The calculated net input y : ")
 print(y)
 # Applying Binary Sigmoidal function on the net input i.e  y
 binary = 1/(1+ (math.exp(-y)))
 print("The output after applying binary sigmoidal function activation ")
 print (round(binary, 3))
 # Applying Bipolar Sigmoidal function on the net input i.e  y
 bipolar = -1+(2/(1+ (math.exp(-y))))
 print("The output after applying bipolar sigmoidal function activation ")
 print(round(bipolar, 3))
```

Output:

```
========== RESTART: C:/Users/Jannavi/AppData/Local/Programs/Python/Python36/RunAllInOne60.py ==========
Enter the value of bias: 1
Enter the number of input neurons: 3
Enter the input: 0.4
Enter the weight: 1
Enter the input: 0.6
Enter the weight: 1
Enter the input: 0.2
Enter the weight: 1
The given weights are:
[1.0, 1.0, 1.0]
The given input are:
[0.4, 0.6, 0.2]
The calculated net input y :
2.2
The output after applying binary sigmoidal function activation
0.9
The output after applying bipolar sigmoidal function activation
0.8
>>>
```

Practical 2A

Aim- Generate AND/NOT function using McCulloch-Pitts neural net.

Code:

```
num_ip = int(input("Enter the number of inputs : "))
theta = 1
x1 =[]
x2 =[]
for i in range(0,num_ip):
    a = int(input("Enter the input x1: "))
    x1.append(a)
    b = int(input("Enter the input x2: "))
    x2.append(b)
print("x1 = ", x1)
print("x2 = ", x2)
print("Value of theta is 1.")
print("Case 1: For calculating the net input the weight is considered as w1 = w2 = 1")
w1 = w2 = 1
case_y1 =[]
case_yin1 = []
print("x1  w1  x2  w2   case_y1     case_yin1")
for i in range(0,num_ip):
    case_y1.append(x1[i]*w1 + x2[i]*w2)
    if (case_y1[i] >= theta):
        case_yin1.append( 1 )
    else:
        case_yin1.append( 0 )
    print(x1[i]," ", w1," ",x2[i]," ",w2,"        ", case_y1[i],"           ",case_yin1[i])
print("From the calculated net inputs, it is not possible to fire the neuron form input (1, 0) only."
"\n Hence, these weights are not suitable. ")
```

```
print("Case 2: For calculating the net input the weight is considered as w1 = 1,
w2 = -1")

w1 = 1

w2 = -1

case_y2 =[]

case_yin2 = []

print("x1  w1  x2  w2   case_y2     case_yin2")

for i in range(0,num_ip):

    case_y2.append(x1[i]*w1 + x2[i]*w2)

    if (case_y2[i] >= theta):

        case_yin2.append( 1 )

    else:

        case_yin2.append( 0 )

    print(x1[i]," ", w1," ",x2[i]," ",w2,"        ", case_y2[i],"           ",case_yin2[i])

print("From the calculated net inputs, it is possible to fire the neuron form
input (1, 0) only."

"\n Hence, these weights are suitable. ")
```

Output:

```
========================================= RESTART: C:/Users/Janhavi/AppData/Local/Programs/Python/Python38/RunAllInOneGo.py =========
Enter the number of inputs : 4
Enter the input x1: 1
Enter the input x2: 1
Enter the input x1: 1
Enter the input x2: 0
Enter the input x1: 0
Enter the input x2: 1
Enter the input x1: 0
Enter the input x2: 0
x1 =  [1, 1, 0, 0]
x2 =  [1, 0, 1, 0]
Value of theta is 1.
Case 1: For calculating the net input the weight is considered as w1 = w2 = 1
x1  w1  x2  w2   case_y1     case_yin1
1   1   1   1          2            1
1   1   0   1          1            1
0   1   1   1          1            1
0   1   0   1          0            0
From the calculated net inputs, it is not possible to fire the neuron form input (1, 0) only.
 Hence, these weights are not suitable.
Case 2: For calculating the net input the weight is considered as w1 = 1, w2 = -1
x1  w1  x2  w2   case_y2     case_yin2
1   1   1   -1         0            0
1   1   0   -1         1            1
0   1   1   -1         -1            0
0   1   0   -1         0            0
From the calculated net inputs, it is possible to fire the neuron form input (1, 0) only.
 Hence, these weights are suitable.
>>>
```

Practical 2B

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

Go to command prompt:

cd C:\Users\Janhavi\AppData\Local\Programs\Python\Python38

python -m pip install numpy

Code:

```
import numpy as np
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([1, 1, 0, 0])
x2=np.array([1, 0, 1, 0])
z=np.array([0, 1, 1, 0])    #This is target output
con=1                    #covergence flag is set to 1
y1=np.zeros((4,))          #starts from 0
y2=np.zeros((4,))
y=np.zeros((4,))
while con==1:
   zin1=np.zeros((4,))
   zin2=np.zeros((4,))
   zin1=x1*w11+x2*w21       #for pair(1,0)
   zin2=x1*w21+x2*w22       #for pair(0,1)
   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)          #calculated output
    print("z",z)          #target output
    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)
```

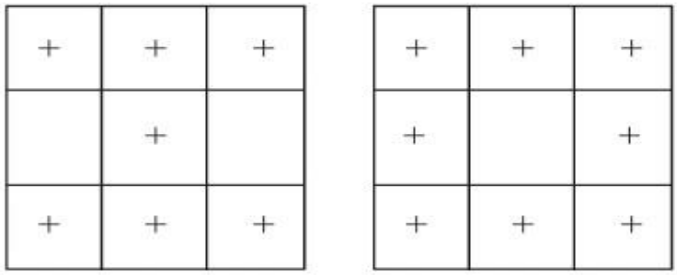
Output:

```
========================================= RESTART: C:/Users/Janhavi/AppData/Local/Programs/Python/Python38/RunAllInOneGo.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 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 3A

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).



Code:

```
import numpy as np
#first pattern
x1=np.array([1,1,1,-1,1,-1,1,1,1])
#second pattern
x2=np.array([1,1,1,1,-1,1,1,1,1])
#initialize bais value
b=0
#define target
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("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/Janhavi/AppData/Local/Programs/Python/Python38/RunAllInOneGo.py =========
First input with target =1
Second input with target =-1
new wt = [ 0  0  0 -2  2 -2  0  0  0]
Bias value 0
>>>
```

Practical 3B

Aim- Write a program to implement of 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/Janhavi/AppData/Local/Programs/Python/Python38/RunAllInOneGo.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 4A

Aim- Write a program for Back Propagation Algorithm.

Code:

```
import numpy as np
import decimal
import math
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("v1=",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/Janhavi/AppData/Local/Programs/Python/Python38/RunAllInOneGo.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
v1= [0.6 0.3]
v2= [-0.1  0.4]
w= [-0.17  0.42  0.12]
bias b1= 0.30294719716271623  b2= 0.5006117804277744
>>>
```

Practical 4B

Aim- Write a program for error Backpropagation algorithm.

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:

```
========== RESTART: C:/Users/Janhavi/AppData/Local/Programs/Python/Python38/RunAllInOneGo.py =========
Enter weight first network12
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 5A

Aim-Write a program for Hopfield Network.

Code:

```
class Neuron:
    def __init__(self, j):
        self.activation = 0
        self.weightv = [0] * 4
        for i in range(4):
            self.weightv[i] = j[i]
    def act(self, m, x):
        a = 0
        for i in range(m):
            a += x[i] * self.weightv[i]
        return a
class Network:
    def __init__(self, a, b, c, d):
        self.nrn = [None] * 4
        self.output = [0] * 4
        self.nrn[0] = Neuron(a)
        self.nrn[1] = Neuron(b)
        self.nrn[2] = Neuron(c)
        self.nrn[3] = Neuron(d)
    def threshld(self, k):
        if k >= 0:
            return 1
        else:
            return 0
    def activation(self, patrn):
        for i in range(4):
            for j in range(4):
                print("\n nrn[{}].weightv[{}] is {}".format(i, j,
self.nrn[i].weightv[j]))
            self.nrn[i].activation = self.nrn[i].act(4, patrn)
```

```
            print("\nactivation is", self.nrn[i].activation)
            self.output[i] = self.threshld(self.nrn[i].activation)
            print("\noutput value is", self.output[i])
def main():
    patrn1 = [1, 0, 1, 0]
    wt1 = [0, -3, 3, -3]
    wt2 = [-3, 0, -3, 3]
    wt3 = [3, -3, 0, -3]
    wt4 = [-3, 3, -3, 0]
    print("\nTHIS PROGRAM IS FOR A HOPFIELD NETWORK WITH A
SINGLE LAYER OF")
    print("4 FULLY INTERCONNECTED NEURONS. THE NETWORK
SHOULD RECALL THE")
    print("PATTERNS 1010 AND 0101 CORRECTLY.\n")
    # create the network by calling its constructor.
    # the constructor calls neuron constructor as many times as the number of
    # neurons in the network.
    h1 = Network(wt1, wt2, wt3, wt4)
    # present a pattern to the network and get the activations of the neurons
    h1.activation(patrn1)
    # check if the pattern given is correctly recalled and give message
    for i in range(4):
        if h1.output[i] == patrn1[i]:
            print("\n pattern=", patrn1[i], " output =", h1.output[i], " component
matches")
        else:
            print("\n pattern=", patrn1[i], " output =", h1.output[i], " discrepancy
occurred")
    print("\n\n")
    patrn2 = [0, 1, 0, 1]
    h1.activation(patrn2)
    for i in range(4):
        if h1.output[i] == patrn2[i]:
            print("\n pattern=", patrn2[i], " output =", h1.output[i], " component
matches")
```

```
        else:
            print("\n pattern=", patrn2[i], " output =", h1.output[i], " discrepancy occurred")
if __name__ == "__main__":
    main()
```

Output:

```
>>>
= RESTART: C:/Users/Janhavi/AppData/Local/Programs/Python/Python38/RunAllInOneGo.py
THIS PROGRAM IS FOR A HOPFIELD NETWORK WITH A SINGLE LAYER OF
4 FULLY INTERCONNECTED NEURONS. THE NETWORK SHOULD RECALL THE
PATTERNS 1010 AND 0101 CORRECTLY.

 nrn[0].weightv[0] is 0
 nrn[0].weightv[1] is -3
 nrn[0].weightv[2] is 3
 nrn[0].weightv[3] is -3
activation is 3
output value is 1
 nrn[1].weightv[0] is -3
 nrn[1].weightv[1] is 0
 nrn[1].weightv[2] is -3
 nrn[1].weightv[3] is 3
activation is -6
output value is 0
 nrn[2].weightv[0] is 3
 nrn[2].weightv[1] is -3
 nrn[2].weightv[2] is 0
 nrn[2].weightv[3] is -3
 nrn[2].weightv[3] is -3
activation is 3
output value is 1
 nrn[3].weightv[0] is -3
 nrn[3].weightv[1] is 3
 nrn[3].weightv[2] is -3
 nrn[3].weightv[3] is 0
activation is -6
output value is 0
 pattern= 1  output = 1  component matches
 pattern= 0  output = 0  component matches
 pattern= 1  output = 1  component matches
 pattern= 0  output = 0  component matches

 nrn[0].weightv[0] is 0
 nrn[0].weightv[1] is -3
 nrn[0].weightv[2] is 3
 nrn[0].weightv[3] is -3
activation is -6
output value is 0
```

```
 nrn[1].weightv[2] is -3
 nrn[1].weightv[3] is 3
activation is 3
output value is 1
 nrn[2].weightv[0] is 3
 nrn[2].weightv[1] is -3
 nrn[2].weightv[2] is 0
 nrn[2].weightv[3] is -3
activation is -6
output value is 0
 nrn[3].weightv[0] is -3
 nrn[3].weightv[1] is 3
 nrn[3].weightv[2] is -3
 nrn[3].weightv[3] is 0
activation is 3
output value is 1
 pattern= 0  output = 0  component matches
 pattern= 1  output = 1  component matches
 pattern= 0  output = 0  component matches
 pattern= 1  output = 1  component matches
```

Practical 5B

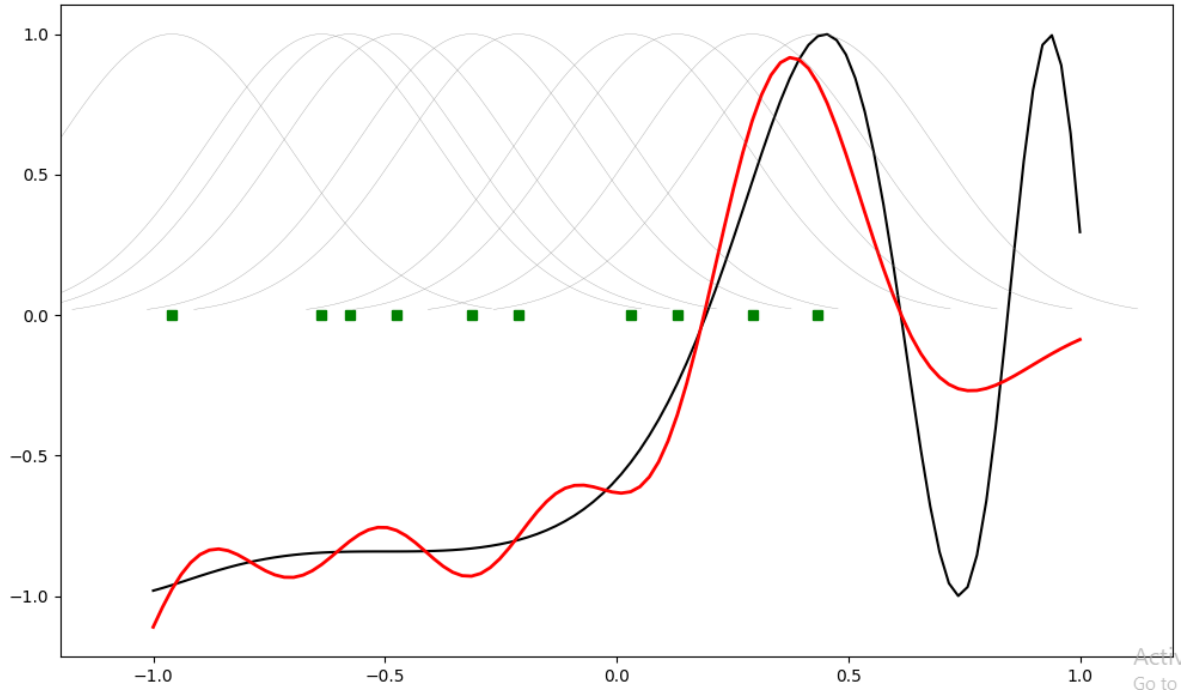
Aim-Write a program for Radial Basis function

Code:

```
from scipy import *
from scipy.linalg import norm, pinv
from matplotlib import pyplot as plt
from numpy import sin, exp, zeros, dot, array, random, arange
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 exp(-self.beta * norm(c-d)**2)
    def _calcAct(self, X):
        G = 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 = dot(pinv(G), Y)
    def test(self, X):
        """ X: matrix of dimensions n x indim """
        G = self._calcAct(X)
        Y = 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 = sin(3*(x+0.5)**3-1)
    # 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, zeros(rbf.numCenters), 'gs')
    for c in rbf.centers:
        # RF prediction lines
        cx = arange(c-0.7, c+0.7, 0.01)
        cy = [rbf._basisfunc(array([cx_]), array([c])) for cx_ in cx]
        plt.plot(cx, cy, '-', color='gray', linewidth=0.2)
    plt.xlim(-1.2, 1.2)
    plt.show()
```

Output:



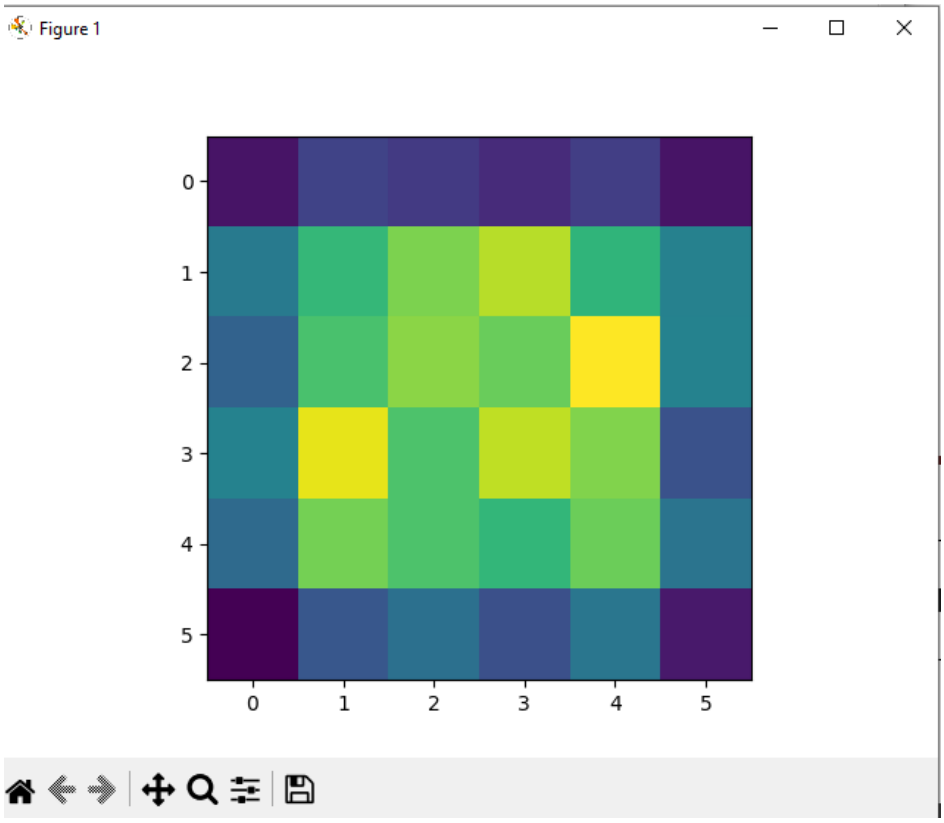
Practical 6A

Aim- Kohonen Self organizing map (SOM)

Code:

```
from minisom import MiniSom

import matplotlib.pyplot as plt

data = [[ 0.80, 0.55, 0.22, 0.03],

[ 0.82, 0.50, 0.23, 0.03],

[ 0.80, 0.54, 0.22, 0.03],

[ 0.80, 0.53, 0.26, 0.03],

[ 0.79, 0.56, 0.22, 0.03],

[ 0.75, 0.60, 0.25, 0.03],

[ 0.77, 0.59, 0.22, 0.03]]

som = MiniSom(6, 6, 4, sigma=0.3, learning_rate=0.5) # initialization of 6x6
SOM

som.train_random(data, 100) # trains the SOM with 100 iterations

plt.imshow(som.distance_map())

plt.show()
```

Output:



Practical 7A

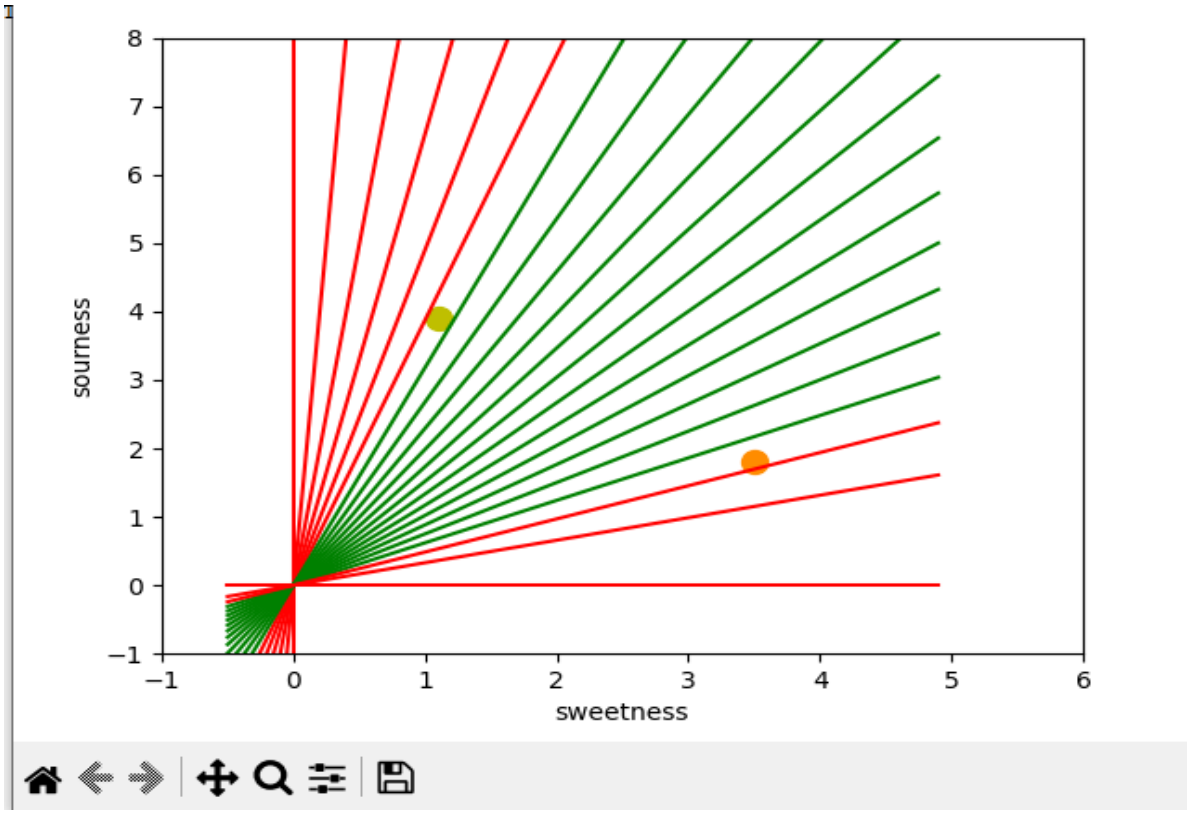
Aim- Write a program for Linear separation.

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
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)
    Y = slope * X
    results = []
    for point in points:
        results.append(dist4line1(*point))
    if (results[0][1] != results[1][1]):
        ax.plot(X, Y, "g-")
    else:
        ax.plot(X, Y, "r-")
plt.show()
```

Output:



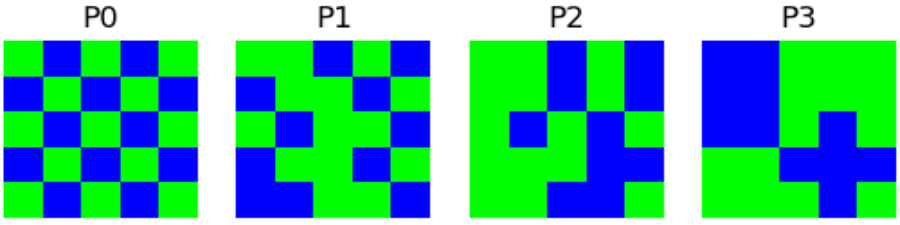
Practical 7B

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

Code:

```
import numpy as np

# Patch np.int to use the built-in int

np.int = int

# Now import the rest of the modules

from neurodynex.hopfield_network import network, pattern_tools, plot_tools

pattern_size = 5

hopfield_net = network.HopfieldNetwork(nr_neurons=pattern_size**2)

factory = pattern_tools.PatternFactory(pattern_size, pattern_size)

checkerboard = factory.create_checkerboard()

pattern_list = [checkerboard]

pattern_list.extend(factory.create_random_pattern_list(nr_patterns=3,
on_probability=0.5))

plot_tools.plot_pattern_list(pattern_list)

overlap_matrix = pattern_tools.compute_overlap_matrix(pattern_list)

plot_tools.plot_overlap_matrix(overlap_matrix)

hopfield_net.store_patterns(pattern_list)

noisy_init_state = pattern_tools.flip_n(checkerboard, nr_of_flips=4)

hopfield_net.set_state_from_pattern(noisy_init_state)

states = hopfield_net.run_with_monitoring(nr_steps=4)

states_as_patterns = factory.reshape_patterns(states)

plot_tools.plot_state_sequence_and_overlap(states_as_patterns, pattern_list,
reference_idx=0, suptitle="Network dynamics")
```

Output:



Practical 8A

Aim- Membership and Identity Operators | in, not in.

Code:

```
# 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
    return 0
list1 = [1, 2, 3, 4, 5]
list2 = [6,7,8,9]
if(overlapping(list1, list2)):
    print("overlapping")
else:
    print("not overlapping")
```

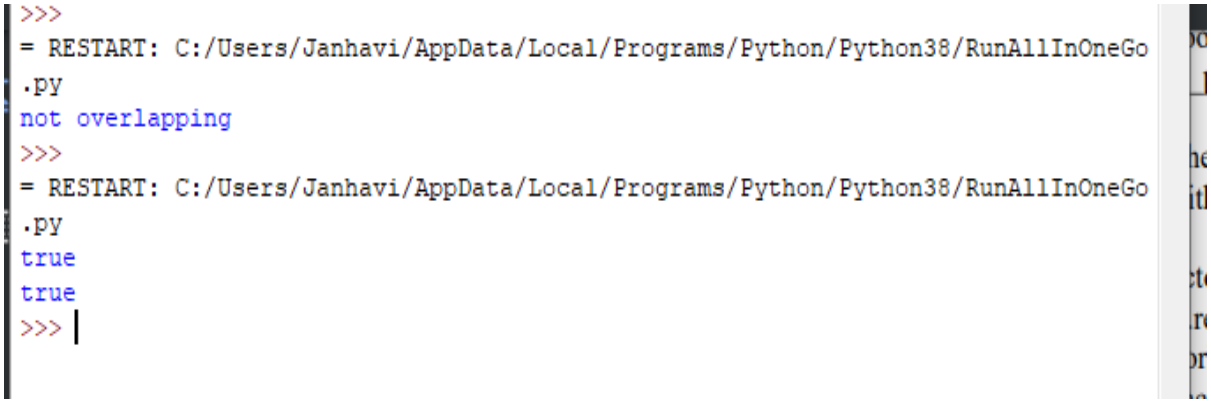
Practical 8B

Aim- Membership and Identity Operators is, is not

Code:

```
# Python program to illustrate the use
# of 'is' identity operator
x = 5
if (type(x) is int):
    print("true")
else:
    print("false")
# Python program to illustrate the
# use of 'is not' identity operator
x = 5.2
if (type(x) is not int):
    print("true")
else:
    print("false")
```

Output:



Practical 9A

Aim- Find ratios using fuzzy logic.

Code:

```
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("FuzzyWuzzyPartialRatio: ", fuzz.partial_ratio(s1, s2))
print("FuzzyWuzzyTokenSortRatio: ", fuzz.token_sort_ratio(s1, s2))
print("FuzzyWuzzyTokenSetRatio: ", fuzz.token_set_ratio(s1, s2))
print("FuzzyWuzzyWRatio: ", 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:

```
Type "help", "copyright", "credits" or "license()" for more information.
>>>
= RESTART: C:/Users/Janhavi/AppData/Local/Programs/Python/Python38/RunAllInOneGo
.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 9B

Aim- Solve Tipping problem using fuzzy logic.

install - pip install scikit-fuzzy

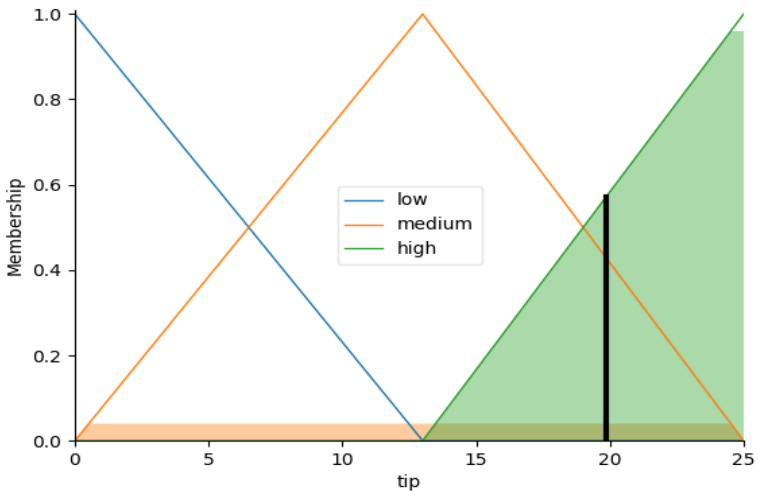
Code:

```
import numpy as np
import skfuzzy as fuzz
from skfuzzy import control as ctrl
```

```
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])
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'])
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:

```
>>>
= RESTART: C:/Users/Janhavi/AppData/Local/Programs/Python/Python38/RunAllInOneGo
.py
19.847607361963192
>>>
```



Practical 10A

Aim- Implementation of Simple genetic algorithm.

Code:

```
import random

# Number of individuals in each generation

POPULATION_SIZE = 100

# Valid genes

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

# Target string to be generated

TARGET = "Janhavi kharkar"

class Individual(object):
    ''' Class representing individual in population '''
    def __init__(self, chromosome):
        self.chromosome = chromosome
        self.fitness = self.cal_fitness()
    @classmethod
    def mutated_genes(cls):
        '''create random genes for mutation '''
        global GENES
        gene = random.choice(GENES)
        return gene
    @classmethod
    def create_gnome(cls):
        '''create chromosome or string of gene '''
        global TARGET
        gnome_len = len(TARGET)
        return [cls.mutated_genes() for _ in range(gnome_len)]
    def mate(self, par2):
        ''' Perform mating and produce new offspring '''
        # chromosome for offspring
        child_chromosome = []
        for gp1, gp2 in zip(self.chromosome, par2.chromosome):
```

```
            # random probability
            prob = random.random()
            # if prob is less than 0.45, insert gene
            # from parent 1
            if prob < 0.45:
                child_chromosome.append(gp1)
            # if prob is between 0.45 and 0.90, insert
            # gene from parent 2
            elif prob < 0.90:
                child_chromosome.append(gp2)
            # otherwise insert random gene(mutate),
            # for maintaining diversity
            else:
                child_chromosome.append(self.mutated_genes())
        # create new Individual(offspring) using
        # generated chromosome for offspring
        return Individual(child_chromosome)
    def cal_fitness(self):
        '''Calculate fittness score, it is the number of characters in string which
differ from target string. '''
        global TARGET
        fitness = 0
        for gs, gt in zip(self.chromosome, TARGET):
            if gs != gt:
                fitness += 1
        return fitness
# Driver code
def main():
    global POPULATION_SIZE
    # current generation
    generation = 1
    found = False
    population = []
    # create initial population
```

```
    for _ in range(POPULATION_SIZE):
        gnome = Individual.create_gnome()
        population.append(Individual(gnome))
    while not found:
        # sort the population in increasing order of fitness score
        population = sorted(population, key=lambda x: x.fitness)
        # if the individual having lowest fitness score ie.
        # 0 then we know that we have reached to the target
        # and break the loop
        if population[0].fitness <= 0:
            found = True
            break
        # Otherwise generate new offsprings for new generation
        new_generation = []
        # Perform Elitism, that mean 10% of fittest population
        # goes to the next generation
        s = int((10 * POPULATION_SIZE) / 100)
        new_generation.extend(population[:s])
        # From 50% of fittest population, Individuals
        # will mate to produce offspring
        s = int((90 * POPULATION_SIZE) / 100)
        for _ in range(s):
            parent1 = random.choice(population[:50])
            parent2 = random.choice(population[:50])
            child = parent1.mate(parent2)
            new_generation.append(child)
        population = new_generation
        print("Generation: {}\tString: {}\tFitness: {}".format(generation,
""".join(population[0].chromosome), population[0].fitness))
        generation += 1
    print("Generation: {}\tString: {}\tFitness: {}".format(generation,
""".join(population[0].chromosome),  population[0].fitness))
if __name__ == '__main__':
    main()
```

Output:

```
D64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
= RESTART: C:/Users/Janhavi/AppData/Local/Programs/Python/Python38/RunAllInOneGo
.py
Generation: 1    String: XalhdKo4CAlM,$H Fitness: 13
Generation: 2    String: XalhdKo4CAlM,$H Fitness: 13
Generation: 3    String: DmnhTHq{OJaCh;% Fitness: 12
Generation: 4    String: DmnhTHq{OJaCh;% Fitness: 12
Generation: 5    String: DmnhTHq{OJaCh;% Fitness: 12
Generation: 6    String: F}khXyi Jhan%P& Fitness: 10
Generation: 7    String: 5}nhWHi JhaC_n% Fitness: 9
Generation: 8    String: Danhh@iQ&haK82r Fitness: 8
Generation: 9    String: ]aihhHi &haKk;r Fitness: 7
Generation: 10   String: ]aihhHi &haKk;r Fitness: 7
Generation: 11   String: JanhfPi &haM8gr Fitness: 6
Generation: 12   String: JanhfPi &haM8gr Fitness: 6
Generation: 13   String: JanhqPi 3haM8ar Fitness: 5
Generation: 14   String: JanhqPi 3haM8ar Fitness: 5
Generation: 15   String: JanhqPi 3haM8ar Fitness: 5
Generation: 16   String: JanhqPi 3haM8ar Fitness: 5
Generation: 17   String: Janhhviy3haMkar Fitness: 4
Generation: 18   String: Janhhviy3haMkar Fitness: 4
Generation: 19   String: JanhMvi }ha#kar Fitness: 3
Generation: 20   String: JanhMvi }ha#kar Fitness: 3
Generation: 21   String: JanhMvi }ha#kar Fitness: 3
Generation: 22   String: JanhMvi }ha#kar Fitness: 3
Generation: 23   String: Janhavi lha$kar Fitness: 2
Generation: 24   String: Janhavi lha$kar Fitness: 2
Generation: 25   String: Janhavi lha$kar Fitness: 2
Generation: 26   String: Janhavi lha$kar Fitness: 2
Generation: 27   String: Janhavi lha$kar Fitness: 2
Generation: 28   String: Janhavi lha$kar Fitness: 2
Generation: 29   String: Janhavi lha$kar Fitness: 2
Generation: 30   String: Janhavi lha$kar Fitness: 2
Generation: 31   String: Janhavi lha$kar Fitness: 2
Generation: 32   String: Janhavi lha$kar Fitness: 2
Generation: 33   String: Janhavi lha$kar Fitness: 2
```

```
Generation: 26   String: Janhavi lha$kar Fitness: 2
Generation: 27   String: Janhavi lha$kar Fitness: 2
Generation: 28   String: Janhavi lha$kar Fitness: 2
Generation: 29   String: Janhavi lha$kar Fitness: 2
Generation: 30   String: Janhavi lha$kar Fitness: 2
Generation: 31   String: Janhavi lha$kar Fitness: 2
Generation: 32   String: Janhavi lha$kar Fitness: 2
Generation: 33   String: Janhavi lha$kar Fitness: 2
Generation: 34   String: Janhavi lha$kar Fitness: 2
Generation: 35   String: Janhavi lha$kar Fitness: 2
Generation: 36   String: Janhavi lha$kar Fitness: 2
Generation: 37   String: Janhavi lha$kar Fitness: 2
Generation: 38   String: Janhavi lha$kar Fitness: 2
Generation: 39   String: Janhavi lha$kar Fitness: 2
Generation: 40   String: Janhavi lha$kar Fitness: 2
Generation: 41   String: Janhavi Rharkar Fitness: 1
Generation: 42   String: Janhavi Rharkar Fitness: 1
Generation: 43   String: Janhavi Rharkar Fitness: 1
Generation: 44   String: Janhavi Rharkar Fitness: 1
Generation: 45   String: Janhavi Rharkar Fitness: 1
Generation: 46   String: Janhavi Rharkar Fitness: 1
Generation: 47   String: Janhavi Rharkar Fitness: 1
Generation: 48   String: Janhavi Rharkar Fitness: 1
Generation: 49   String: Janhavi Rharkar Fitness: 1
Generation: 50   String: Janhavi Rharkar Fitness: 1
Generation: 51   String: Janhavi Rharkar Fitness: 1
Generation: 52   String: Janhavi Rharkar Fitness: 1
Generation: 53   String: Janhavi Rharkar Fitness: 1
Generation: 54   String: Janhavi Rharkar Fitness: 1
Generation: 55   String: Janhavi Rharkar Fitness: 1
Generation: 56   String: Janhavi Rharkar Fitness: 1
Generation: 57   String: Janhavi Rharkar Fitness: 1
Generation: 58   String: Janhavi Rharkar Fitness: 1
Generation: 59   String: Janhavi Rharkar Fitness: 1
Generation: 60   String: Janhavi Rharkar Fitness: 1
Generation: 61   String: Janhavi Rharkar Fitness: 1
Generation: 62   String: Janhavi Rharkar Fitness: 1
Generation: 63   String: Janhavi Rharkar Fitness: 1
Generation: 64   String: Janhavi kharkar Fitness: 0
```

Practical 10B

Aim- Create two classes: City and Fitness using Genetic algorithm.

Code:

```
import numpy as np
import random
import operator
import pandas as pd
import matplotlib.pyplot as plt
from tkinter import Tk, Canvas, Frame, BOTH, Text
import math
class City:
    def __init__(self, x, y):
        self.x = x
        self.y = y
    def distance(self, city):
        xDis = abs(self.x - city.x)
        yDis = abs(self.y - city.y)
        distance = np.sqrt((xDis ** 2) + (yDis ** 2))
        return distance
    def __repr__(self):
        return "(" + str(self.x) + "," + str(self.y) + ")"
class Fitness:
    def __init__(self, route):
        self.route = route
        self.distance = 0
        self.fitness= 0.0
    def routeDistance(self):
        if self.distance == 0:
            pathDistance = 0
            for i in range(0, len(self.route)):
                fromCity = self.route[i]
                toCity = None
```

```
                if i + 1 < len(self.route):
                    toCity = self.route[i + 1]
                else:
                    toCity = self.route[0]
                pathDistance += fromCity.distance(toCity)
            self.distance = pathDistance
        return self.distance
    def routeFitness(self):
        if self.fitness == 0:
            self.fitness = 1 / float(self.routeDistance())
        return self.fitness
def createRoute(cityList):
    route = random.sample(cityList, len(cityList))
    return route
def initialPopulation(popSize, cityList):
    population = []
    for i in range(0, popSize):
        population.append(createRoute(cityList))
    return population
def rankRoutes(population):
    fitnessResults = {}
    for i in range(0, len(population)):
        fitnessResults[i] = Fitness(population[i]).routeFitness()
    return sorted(fitnessResults.items(), key=operator.itemgetter(1),
reverse=True)
def selection(popRanked, eliteSize):
    selectionResults = []
    df = pd.DataFrame(np.array(popRanked), columns=["Index", "Fitness"])
    df['cum_sum'] = df.Fitness.cumsum()
    df['cum_perc'] = 100 * df.cum_sum / df.Fitness.sum()
    for i in range(0, eliteSize):
        selectionResults.append(popRanked[i][0])
    for i in range(0, len(popRanked) - eliteSize):
        pick = 100 * random.random()
```

```
        for i in range(0, len(popRanked)):
            if pick <= df.iat[i, 3]:
                selectionResults.append(popRanked[i][0])
                break
    return selectionResults
def matingPool(population, selectionResults):
    matingpool = []
    for i in range(0, len(selectionResults)):
        index = selectionResults[i]
        matingpool.append(population[index])
    return matingpool
def breed(parent1, parent2):
    child = []
    childP1 = []
    childP2 = []
    geneA = int(random.random() * len(parent1))
    geneB = int(random.random() * len(parent1))
    startGene = min(geneA, geneB)
    endGene = max(geneA, geneB)
    for i in range(startGene, endGene):
        childP1.append(parent1[i])
    childP2 = [item for item in parent2 if item not in childP1]
    child = childP1 + childP2
    return child
def breedPopulation(matingpool, eliteSize):
    children = []
    length = len(matingpool) - eliteSize
    pool = random.sample(matingpool, len(matingpool))
    for i in range(0, eliteSize):
        children.append(matingpool[i])
    for i in range(0, length):
        child = breed(pool[i], pool[len(matingpool) - i - 1])
        children.append(child)
```

```
    return children
def mutate(individual, mutationRate):
    for swapped in range(len(individual)):
        if(random.random() < mutationRate):
            swapWith = int(random.random() * len(individual))
            city1 = individual[swapped]
            city2 = individual[swapWith]
            individual[swapped] = city2
            individual[swapWith] = city1
    return individual
def mutatePopulation(population, mutationRate):
    mutatedPop = []
    for ind in range(0, len(population)):
        mutatedInd = mutate(population[ind], mutationRate)
        mutatedPop.append(mutatedInd)
    return mutatedPop
def nextGeneration(currentGen, eliteSize, mutationRate):
    popRanked = rankRoutes(currentGen)
    selectionResults = selection(popRanked, eliteSize)
    matingpool = matingPool(currentGen, selectionResults)
    children = breedPopulation(matingpool, eliteSize)
    nextGeneration = mutatePopulation(children, mutationRate)
    return nextGeneration
def geneticAlgorithm(population, popSize, eliteSize, mutationRate,
generations):
    pop = initialPopulation(popSize, population)
    print("Initial distance: " + str(1 / rankRoutes(pop)[0][1]))
    for i in range(0, generations):
        pop = nextGeneration(pop, eliteSize, mutationRate)
    print("Final distance: " + str(1 / rankRoutes(pop)[0][1]))
    bestRouteIndex = rankRoutes(pop)[0][0]
    bestRoute = pop[bestRouteIndex]
    return bestRoute
```

```
def geneticAlgorithmPlot(population, popSize, eliteSize, mutationRate,
generations):
    pop = initialPopulation(popSize, population)
    progress = []
    progress.append(1 / rankRoutes(pop)[0][1])
    for i in range(0, generations):
        pop = nextGeneration(pop, eliteSize, mutationRate)
        progress.append(1 / rankRoutes(pop)[0][1])
    plt.plot(progress)
    plt.ylabel('Distance')
    plt.xlabel('Generation')
    plt.show()
def main():
    cityList = []
    for i in range(0, 25):
        cityList.append(City(x=int(random.random() * 200),
y=int(random.random() * 200)))
    geneticAlgorithmPlot(population=cityList, popSize=100, eliteSize=20,
mutationRate=0.01, generations=500)
 if __name__ == '__main__':
 main()
```

Output: