SOFT COMPUTING TECHNIQUES

Practical 1

Practical 1 a: Design a simple linear neural network model.

Coding:-

```
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)&(net<=1)):
out =net
else:
out=1
print("net=",net)
print("output=",out)
```

Output:-

```
Enter value of x:1
Enter value of weight w:1
Enter value of bias b:1
net= 2
output= 1
```

Practical 1 b: Calculate the output of neural net using both binary and bipolar sigmoidal function

Coding:-

```
n = int(input("Enter number of elements : "))
print("Enter the inputs")
inputs = []
for i in range(0, n):
   ele = float(input())
   inputs.append(ele)
print(inputs)
print("Enter the weights")
weights = []
for j in range(0, n):
   ele = float(input())
   weights.append(ele)
print(weights)
print("The net input can be calculated as Yin = x1w1 + x2w2 + x3w3")
Yin = []
for i in range(0, n):
   Yin.append(inputs[i]*weights[i])
print(round(sum(Yin),3))
```

Output:-

```
Enter number of elements : 3
Enter the inputs
0.3
0.5
0.6
[0.3, 0.5, 0.6]
Enter the weights
0.2
0.1
-0.3
[0.2, 0.1, -0.3]
The net input can be calculated as Yin = x1w1 + x2w2 + x3w3
-0.07
```

Practical 2

Practical 2 a: Implement AND/NOT function using McCulloch-Pits neuron (use binary data representation).

Coding:-

```
# 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)
   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)
   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:-

```
Enter the number of inputs : 4
For the  4  inputs calculate the net input using yin = x1w1 + x2w2
x1 = 0
x2 = 0
x1 =  [0]
x2 =  [0]
x1 = 0
x2 = 1
x1 =  [0, 0]
x2 =  [0, 1]
x1 = 1
x2 = 0
x1 =  [0, 0, 1]
x2 =  [0, 1, 0]
x1 = 1
x2 = 1
x1 =  [0, 0, 1, 1]
x2 =  [0, 1, 0, 1]
Yin =  [0]
Yin =  [1]
Yin =  [1]
Yin =  [2]
After assuming one weight as excitatory and the other as inhibitory Yin =  [0]
After assuming one weight as excitatory and the other as inhibitory Yin =  [0, -1]
After assuming one weight as excitatory and the other as inhibitory Yin =  [0, -1, 1]
After assuming one weight as excitatory and the other as inhibitory Yin =  [0, -1, 1, 0]
Y =  [0]
Y =  [0, 0]
Y =  [0, 0, 1, 0]
```

Practical 2 b: Generate XOR function using McCulloch-Pitts neural net

Coding:-

```
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([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,))
while 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:-

```
Weight w11=1
weight w12=1
Weight w21=1
weight w22=1
weight v1=1
weight v2=1
theta=1
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: Write a program to implement Hebb's rule

Coding:-

```
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("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:-

```
First input with target =1
new wt = [ 1  1  1 -1  1 -1  1  1  1]
Bias value 9
Second input with target =-1
new wt = [ 0  0  0 -2  2 -2  0  0  0]
Bias value 0
```

Practical 3 b: Write a program to implement of delta rule

Coding:-

```
#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:-

```
Initial inputs:1
Initial inputs:1
Initial inputs:1
Initial weights:1
Initial weights:1
Initial weights:1
Desired output:2
Enter learning rate:1
actual [1. 1. 1.]
desired [2. 0. 0.]
Desired output:3
Enter learning rate:1
actual [1. 1. 1.]
desired [2. 3. 0.]
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: Write a program for Back Propagation Algorithm

Coding:-

```
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("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:-

```
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= 0.047627628296582778
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= 0.002947197162716259 2
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: Write a Program For Error Back Propagation Algorithm (Ebpa) Learning.

Coding:-

```
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:-

```
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: Write a program for Hopfield Network.

Coding:-

```
#include "hop.h"
neuron::neuron(int *j)
{
inti;
for(i=0;i<4;i++)
{
weightv[i]= *(j+i);
}
}
int neuron::act(int m, int *x)
{
inti;
int a=0;
for(i=0;i<m;i++)
{
a += x[i]*weightv[i];
}
return a;
}
int network::threshld(int k)
{
if(k>=0)
return (1);
else
return (0);
}
network::network(int a[4],int b[4],int c[4],int d[4])
{
nrn[0] = neuron(a) ;
nrn[1] = neuron(b) ;
nrn[2] = neuron(c) ;
nrn[3] = neuron(d) ;
}
void network::activation(int *patrn)
{
inti,j;
for(i=0;i<4;i++)
{
for(j=0;j<4;j++)
{
cout<<"\n nrn["<<i<<"].weightv["<<j<<"] is "
<<nrn[i].weightv[j];
}
nrn[i].activation = nrn[i].act(4,patrn);
```

```
cout<<"\nactivation is "<<nrn[i].activation;
output[i]=threshld(nrn[i].activation);
cout<<"\noutput value is "<<output[i]<<"\n";
}
}
void main ()
{
int patrn1[]= {1,0,1,0},i;
int wt1[]= {0,-3,3,-3};
int wt2[]= {-3,0,-3,3};
int wt3[]= {3,-3,0,-3};
int wt4[]= {-3,3,-3,0};
cout<<"\nTHIS PROGRAM IS FOR A HOPFIELD NETWORK WITH A SINGLE LAYER
OF";
cout<<"\n4 FULLY INTERCONNECTED NEURONS. THE NETWORK SHOULD
RECALLTHE";
cout<<"\nPATTERNS 1010 AND 0101 CORRECTLY.\n";
//create the network by calling its constructor.
// the constructor calls neuron constructor as many times as thenumber of
// neurons in the network.
network h1(wt1,wt2,wt3,wt4);
//present a pattern to the network and get the activations of theneurons
h1.activation(patrn1);
//check if the pattern given is correctly recalled and give message
for(i=0;i<4;i++)
{
if (h1.output[i] == patrn1[i])
cout<<"\n pattern= "<<patrn1[i]<<
" output = "<<h1.output[i]<<" component matches";
else
cout<<"\n pattern= "<<patrn1[i]<<
" output = "<<h1.output[i]<<
" discrepancy occurred";
}
cout<<"\n\n";
int patrn2[]= {0,1,0,1};
h1.activation(patrn2);
for(i=0;i<4;i++)
{
if (h1.output[i] == patrn2[i])
cout<<"\n pattern= "<<patrn2[i]<<
" output = "<<h1.output[i]<<" component matches";
else
cout<<"\n pattern= "<<patrn2[i]<<
" output = "<<h1.output[i]<<
" discrepancy occurred";
}
}
======== End code of main program=============
//Hop.h
//Single layer Hopfield Network with 4 neurons
```

```
#include <stdio.h>
#include <iostream.h>
#include <math.h>
class neuron
{
protected:
int activation;
friend class network;
public:
intweightv[4];
neuron() {};
neuron(int *j) ;
int act(int, int*);
};
class network
{
public:
neuron nrn[4];
int output[4];
intthreshld(int) ;
void activation(int j[4]);
network(int*,int*,int*,int*);
};
```

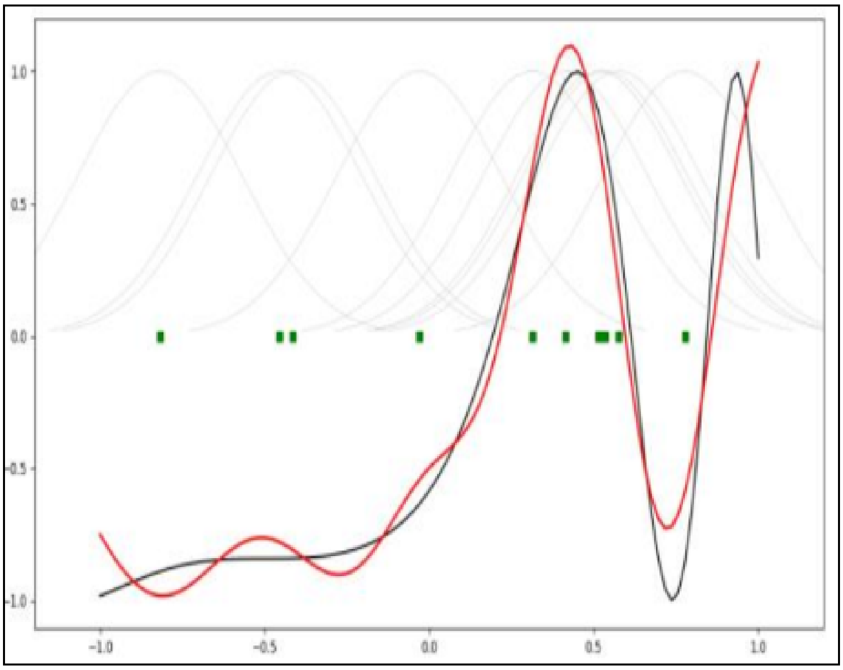
Practical 5 b: Write a program for Radial Basis function

Coding:-

```
from scipy import *
from scipy.linalg import norm, pinv
from matplotlib import pyplot as plt
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):
# calculate activations of RBFs
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 =mgrid[-1:1:complex(0,n)].reshape(n, 1)
# set y and add random noise
y =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, 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 :



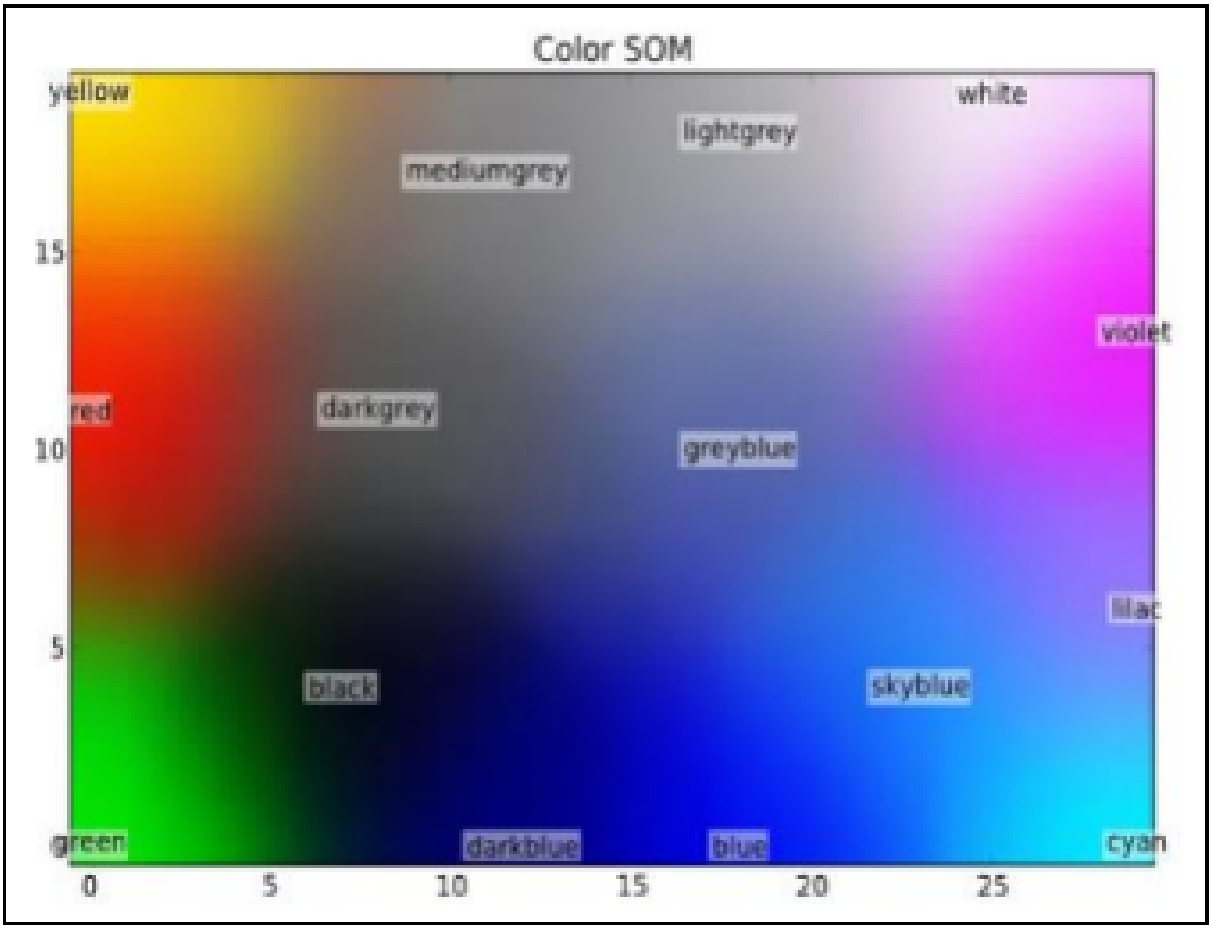
Practical 6

Practical 6 a: Self-Organizing Maps

Coding:-

```
frommvpa2.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]])
# store the names of the colors for visualization later on
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)
fori,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
15
violet
red
darkgrey
10
greyblue
lilac
5
black
skyblue
green
darkblue
blue
cyan
0
5
10
15
20
25

Practical 6 b: ADAPTIVE RESONANCE THEORY

Coding:-

```
from__future__import
division
importnumpyas np
fromneupy.utilsimportformat_data
fromneupy.core.propertiesimport (ProperFractionProperty,
IntProperty)
fromneupy.algorithms.baseimportBaseNetwork
__all__= ('ART1',)
classART1(BaseNetwork):
"""
Adaptive Resonance Theory (ART1) Network for binary
data clustering.
Notes
-----
- Weights are not random, so the result will be
always reproduceble.
Parameters
----------
rho : float
Control reset action in training process. Value must be
between ``0`` and ``1``, defaults to ``0.5``.
n_clusters : int
Number of clusters, defaults to ``2``. Min value is also
``2``.
{BaseNetwork.Parameters}
Methods
-------
train(X)
ART trains until all clusters are found.
predict(X)
Each prediction trains a new network. It's an alias to
the ``train`` method.
{BaseSkeleton.fit}
Examples
--------
>>>import numpy as np
>>>from neupy import algorithms
>>>
>>>data = np.array([
... [0, 1, 0],
... [1, 0, 0],
... [1, 1, 0],
... ])
>>>>
>>>artnet = algorithms.ART1(
... step=2,
... rho=0.7,
... n_clusters=2,
```

```
... verbose=False
... )
>>>artnet.predict(data)
array([ 0., 1., 1.])
"""
rho =ProperFractionProperty(default=0.5)
n_clusters=IntProperty(default=2, minval=2)
deftrain(self, X):
X =format_data(X)
ifX.ndim!=2:
raiseValueError("Input value must be 2 dimensional, got "
"{}".format(X.ndim))
nsamples, n_features=X.shape
n_clusters=self.n_clusters
step =self.step
rho =self.rho
ifnp.any((X !=0) & (X !=1)):
raiseValueError("ART1 Network works only with binary
matrices")
ifnothasattr(self, 'weight_21'):
self.weight_21 =np.ones((n_features, n_clusters))
ifnothasattr(self, 'weight_12'):
scaler = step / (step +n_clusters-1)
self.weight_12 = scaler *self.weight_21.T
weight_21 =self.weight_21
weight_12 =self.weight_12
ifn_features!= weight_21.shape[0]:
raiseValueError("Input data has invalid number of features. "
"Got {} instead of {}"
"".format(n_features, weight_21.shape[0]))
classes =np.zeros(n_samples)
# Train network
fori, p inenumerate(X):
disabled_neurons= []
reseted_values= []
reset =True
while reset:
output1 = p
input2 = np.dot(weight_12, output1.T)
output2 =np.zeros(input2.size)
input2[disabled_neurons] =-np.inf
winner_index= input2.argmax()
output2[winner_index] =1
expectation = np.dot(weight_21, output2)
output1 =np.logical_and(p, expectation).astype(int)
reset_value= np.dot(output1.T, output1) / np.dot(p.T, p)
reset =reset_value< rho
if reset:
disabled_neurons.append(winner_index)
reseted_values.append((reset_value, winner_index))
iflen(disabled_neurons) >=n_clusters:
```

```
# Got this case only if we test all possible clusters
reset =False
winner_index=None
ifnot reset:
ifwinner_indexisnotNone:
weight_12[winner_index, :] = (step * output1) / (
step + np.dot(output1.T, output1) -1
)
weight_21[:, winner_index] = output1
else:
# Get result with the best `rho`
winner_index=max(reseted_values)[1]
classes[i] =winner_index
return classes
defpredict(self, X):
returnself.train(X)
```

Practical 7

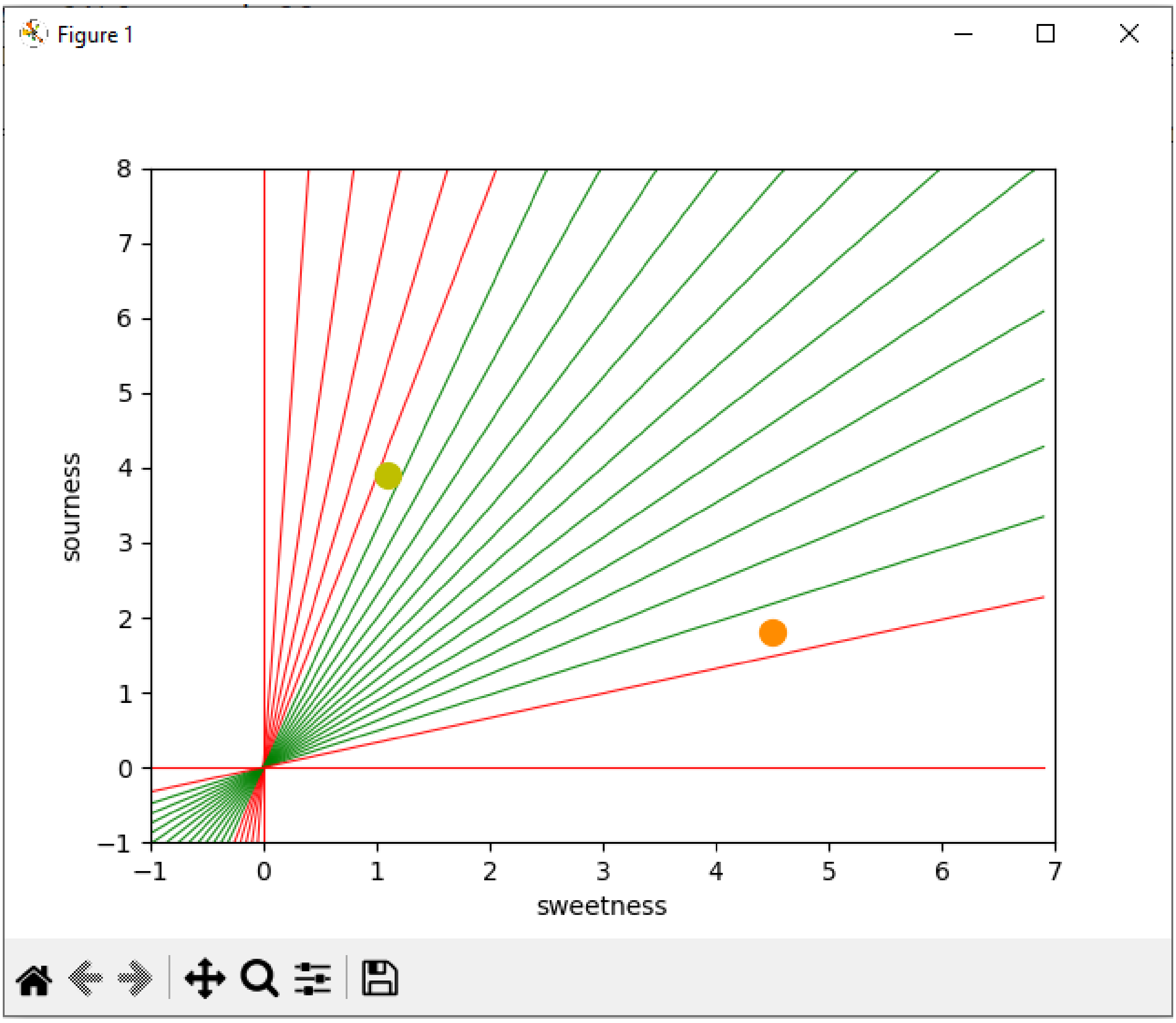
Practical 7 a: Line Separation

Coding:-

```
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

orange = (4.5, 1.8)
lemon = (1.1, 3.9)
fruits_coords = [orange, lemon]

fig, ax = plt.subplots()
ax.set_xlabel("sweetness")
ax.set_ylabel("sourness")
x_min, x_max = -1, 7
y_min, y_max = -1, 8
ax.set_xlim([x_min, x_max])
ax.set_ylim([y_min, y_max])
X = np.arange(x_min, x_max, 0.1)

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 fruits_coords:
        results.append(dist4line1(*point))
    if (results[0][1] != results[1][1]):
        ax.plot(X, Y, "g-", linewidth=0.8, alpha=0.9)
    else:
        ax.plot(X, Y, "r-", linewidth=0.8, alpha=0.9)

size = 10
```

```
for (index, (x, y)) in enumerate(fruits_coords):
    if index== 0:
        ax.plot(x, y, "o",
                color="darkorange",
                markersize=size)
    else:
        ax.plot(x, y, "oy",
                markersize=size)


plt.show()
```

Output:-



Practical 8

Practical 8 a: Membership and Identity operators in, not in

Coding:-

```
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")
```

Output:-

```
not overlapping
```

Practical 8 b: Membership and Identity Operators is, is not

Coding:-

```
x = 5
if (type(x) is int):
   print ("true")
else:
   print ("false")
x = 5.2
if (type(x) is not int):
   print ("true")
else:
   print ("false")
```

Output:-

```
true
true
```

Practical 9

Practical 9 a: Find the ratios using fuzzy logic

Coding:-

```
# Python code showing all the ratios together,
# make sure you have installed fuzzywuzzy module
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:-

```
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: Solve Tipping Problem using fuzzy logic

Coding:-

```
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])
"""To help understand what the membership looks like, use the ``view`` methods."""
quality['average'].view()
""".. image:: PLOT2RST.current_figure"""
service.view()
""".. image:: PLOT2RST.current_figure"""
tip.view()
""".. image:: PLOT2RST.current_figure"""
```

Output:-

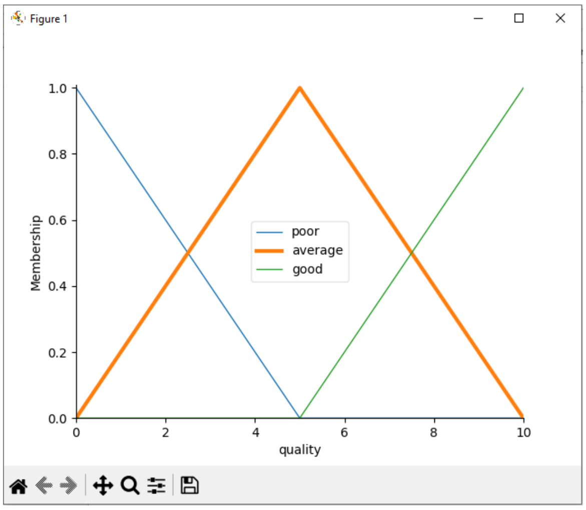


Figure 1
1.0
0.8
0.6
0.4
0.2
0.0
Membership
0
2
4
6
8
10
quality
poor
average
good

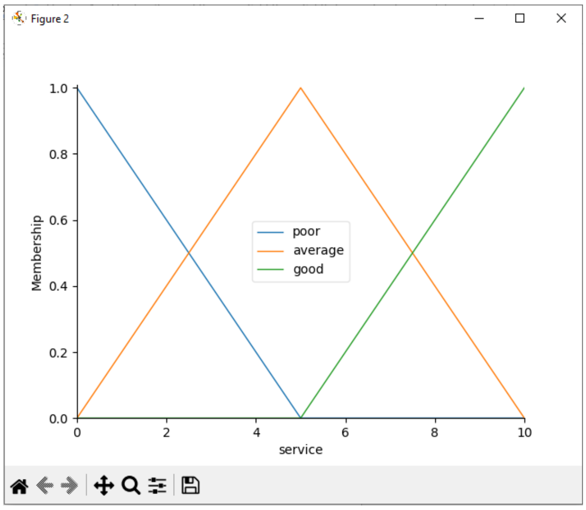


Figure 2
1.0
0.8
0.6
0.4
0.2
0.0
Membership
0
2
4
6
8
10
service
poor
average
good

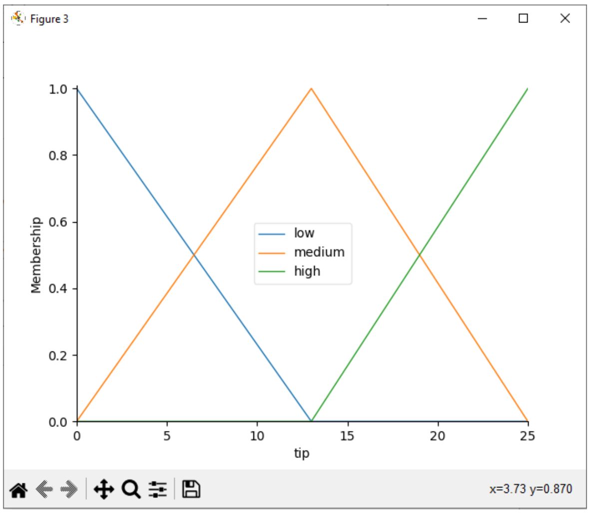


Figure 3
1.0
0.8
0.6
0.4
0.2
0.0
Membership
0
5
10
15
20
25
tip
low
medium
high
x=3.73 y=0.870

Practical 10

Practical 10 a: Implementation of simple genetic algorithm

Coding:-

```
import random
POPULATION_SIZE = 100
GENES = '''abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ 1234567890, .
-;:_!"#%&/()=?@${[]}'''
TARGET = "I love GeeksforGeeks"
class Individual( object ):
    '''Class representing individual in population'''
    def __init__(self, chromosome):
        self.chromosome = chromosome
        self.fitness = self.cal_fitness()
    @classmethod
    def mutated_genes(self):
        '''create random genes for mutation'''
        global GENES
        gene = random.choice( GENES )
        return gene
    @classmethod
    def create_gnome(self):
        '''create chromosome or string of genes'''
        global TARGET
        gnome_len = len( TARGET )
        return [self.mutated_genes() for _ in range( gnome_len )]
    def mate(self, par2):
        '''Perform mating and produce new offspring'''
        child_chromosome = []
        for gp1, gp2 in zip( self.chromosome, par2.chromosome ):
            prob = random.random()
            if prob < 0.45:
                child_chromosome.append( gp1 )
            elif prob < 0.90:
                child_chromosome.append( gp2 )
            else:
                child_chromosome.append( self.mutated_genes() )
        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
def main():
    global POPULATION_SIZE
```

```
    generation = 1
    found = False
    population = []
    for _ in range( POPULATION_SIZE ):
        gnome = Individual.create_gnome()
        population.append( Individual( gnome ) )
    while not found:
        population = sorted( population, key=lambda x: x.fitness )
        if population[0].fitness <= 0:
            found = True
            break
        new_generation = []
        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] )
            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:-

```
Generation: 1   String: /VoiV! zqe#He&YL{dVw    Fitness: 18
Generation: 2   String: /VoiV! zqe#He&YL{dVw    Fitness: 18
Generation: 3   String: /MoRS! Bee]!j&[G8j@g    Fitness: 16
Generation: 4   String: /MoRS! Bee]!j&[G8j@g    Fitness: 16
Generation: 5   String: IVoibz Gew]iDkYhTeh6    Fitness: 15
Generation: 6   String: IMSJSS Gee]DDkYhTe@g    Fitness: 14
Generation: 7   String: IMSJSS Gee]DDkYhTe@g    Fitness: 14
Generation: 8   String: I lIv$ GWe2KdM/Lfrkg    Fitness: 12
Generation: 9   String: I lIv$ GWe2KdM/Lfrkg    Fitness: 12
Generation: 10  String: I &o6! BeeC"dy?PTeks    Fitness: 11
Generation: 11  String: I l7v! Gee2KjM/LTekS    Fitness: 10
Generation: 12  String: I l7v! Gee2{jMYLTeks    Fitness: 9
Generation: 13  String: I l7v! Gee2{jMYLTeks    Fitness: 9
Generation: 14  String: I l7v! Gee2{jMYLTeks    Fitness: 9
Generation: 15  String: I l7v! Gee2{jMYLTeks    Fitness: 9
Generation: 16  String: I lJv! Geej"jl/GTeks    Fitness: 8
Generation: 17  String: I lJv! Geej"jl/GTeks    Fitness: 8
Generation: 18  String: I lJv! Geej"jl/GTeks    Fitness: 8
Generation: 19  String: I lJv! Geej"jl/GTeks    Fitness: 8
Generation: 20  String: I lJv! Geej"jl/GTeks    Fitness: 8
Generation: 21  String: I lJv! Geej"jl/GTeks    Fitness: 8
Generation: 22  String: I lpv5 GeekZj:FGTeks    Fitness: 7
Generation: 23  String: I lpv5 GeekZj:FGTeks    Fitness: 7
Generation: 24  String: I lpv5 GeekZj:FGTeks    Fitness: 7
Generation: 25  String: I lov" GeekhjM[GTeks    Fitness: 6
Generation: 26  String: I lov" GeekhjM[GTeks    Fitness: 6
Generation: 27  String: I lov" GeekhjM[GTeks    Fitness: 6
Generation: 28  String: I lov" GeekhjM[GTeks    Fitness: 6
Generation: 29  String: I lov" GeekhjM[GTeks    Fitness: 6
Generation: 30  String: I lov" GeekhjM[GTeks    Fitness: 6
```

Practical 10 b: Create two classes: City and Fitness using Genetic algorithm

Coding:-

```
import numpy as np, random, operator, pandas as pd, matplotlib.pyplot as plt
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
cityList = []
for i in range(0,25):
    cityList.append(City(x=int(random.random() * 200), y=int(random.random() * 200)))
geneticAlgorithm(population=cityList, popSize=100, eliteSize=20, mutationRate=0.01,
generations=500)
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()
geneticAlgorithmPlot(population=cityList, popSize=100, eliteSize=20, mutationRate=0.01,
generations=500)
```

Output:-

```
Initial distance: 2139.8284418721596
Final distance: 916.3186808759739
```