# INDEX

| SR.NO | PRACTICAL AIM | DATE | SIGNATURE |
|---|---|---|---|
| **1** | Implement the following:<br>**a.** Design a simple linear neural network model.<br>**b.** Calculate the output of neural net using both binary andbipolar sigmoidal function. | | |
| **2** | Implement the following:<br>a. Generate AND/NOT function using McCulloch-Pittsneural net.<br>b. Generate XOR function using McCulloch-Pitts neuralnet. | | |
| **3** | Implement the Following<br>a. Write a program to implement Hebb's rule.<br>b. Write a program to implement of delta rule. | | |
| **4** | Implement the Following<br>a. Write a program for Back Propagation Algorithm<br>b. Write a program for error Backpropagation algorithm. | | |
| **5** | Implement the Following<br>a. Write a program for Hopfield Network.<br>b. Write a program for Radial Basis function | | |
| **6** | Implement the Following<br>a. Kohonen Self organizing map<br>b. Adaptive resonance theory | | |
| **7** | Implement the Following<br>a. Write a program for Linear separation. | | |
| **8** | Implement the Following<br>a. Membership and Identity Operators \| in, not in,<br>b. Membership and Identity Operatorsis, is not | | |
| **9** | Implement the Following<br>a. Find ratios using fuzzy logic<br>b. Solve Tipping problem using fuzzy logic | | |
| **10** | Implement the Following<br>a. Implementation of Simple genetic algorithm<br>b. Create two classes: City and Fitness using Geneticalgorithm | | |

# Practical 1

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

```
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)
```

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

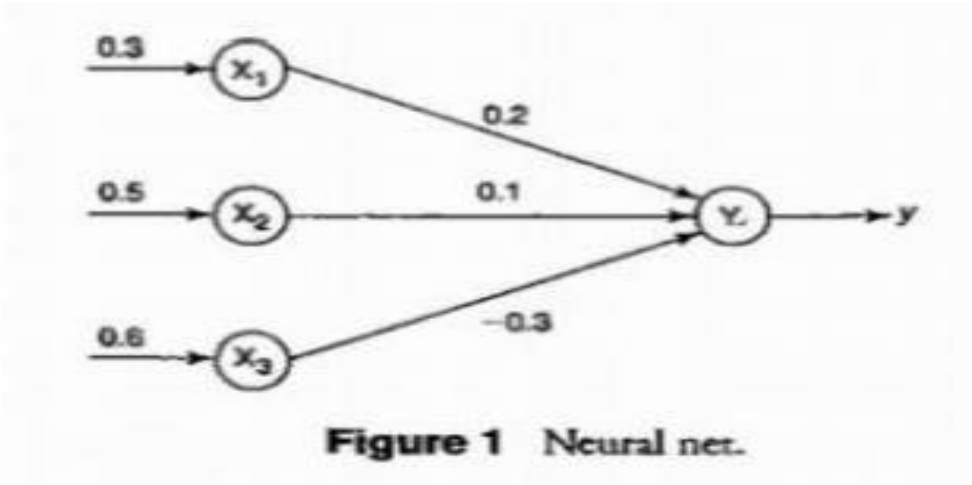



**Figure 1** Neural net.

Code :

```
# number of elements as input
n = int(input("Enter number of elements : "))
# In[2]:
print("Enter the inputs")
inputs = [] # creating an empty list for inputs
# iterating till the range
for i in range(0, n):
ele = float(input())
inputs.append(ele) # adding the element
print(inputs)
# In[3]:
print("Enter the weights")
# creating an empty list for weights
weights = []
# iterating till the range
for i in range(0, n):
ele = float(input())
weights.append(ele) # adding the element
print(weights)
# In[4]:
print("The net input can be calculated as Yin = x1w1 + x2w2 + x3w3")
# In[5]:
Yin = []
for i in range(0, n):
Yin.append(inputs[i]*weights[i])
print(round(sum(Yin),3))
```

**Output :**

```
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).**

Code:

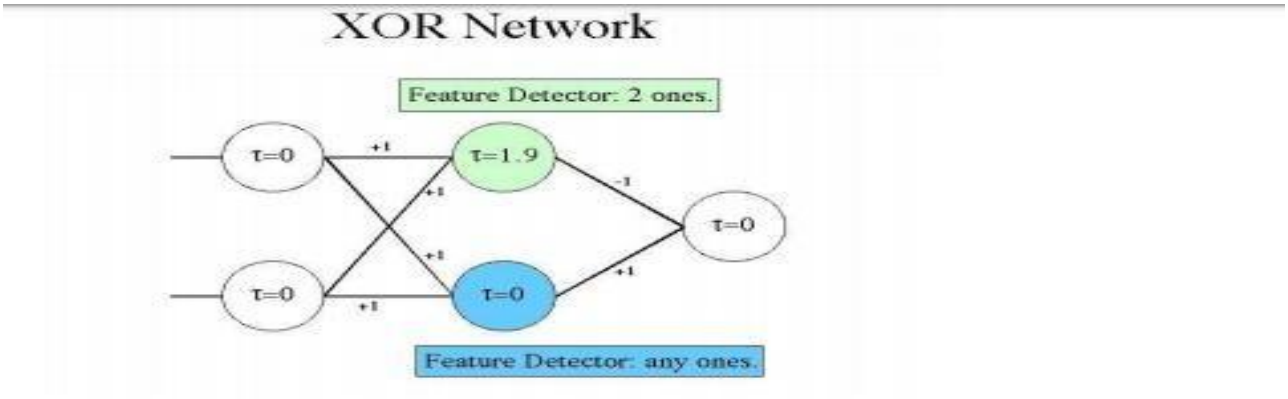
```
# enter the no of inputs
num_ip = int(input("Enter the number of inputs : "))
#Set the weights with value 1
w1 = 1
w2 = 1
print("For the ", num_ip , " inputs calculate the net input using yin = x1w1 + x2w2 ")
x1 = []
x2 = []
for j in range(0, num_ip):
 ele1 = int(input("x1 = "))
ele2 = int(input("x2 = "))
x1.append(ele1)
x2.append(ele2)
print("x1 = ",x1)
print("x2 = ",x2)
n = x1 * w1
m = x2 * w2
Yin = []
for i in range(0, num_ip):
Yin.append(n[i] + m[i])
print("Yin = ",Yin)
#Assume one weight as excitatory and the other as inhibitory, i.e.,
Yin = []
for i in range(0, num_ip):

Yin.append(n[i] - m[i])
print("After assuming one weight as excitatory and the other as inhibitory Yin = ",Yin)
#From the calculated net inputs, now it is possible to fire the neuron for input (1, 0)
#only by fixing a threshold of 1, i.e., θ ≥ 1 for Y unit.
#Thus, w1 = 1, w2 = -1; θ ≥ 1
Y=[]
for i in range(0, num_ip):
 if(Yin[i]>=1):
ele= 1
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 = 1

x1 = 1

x2 = 0

x1 = 1

x2 = 1
x1 =  [0, 0, 1, 1]
x2 =  [0, 1, 0, 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]

In [14]:
```

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




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

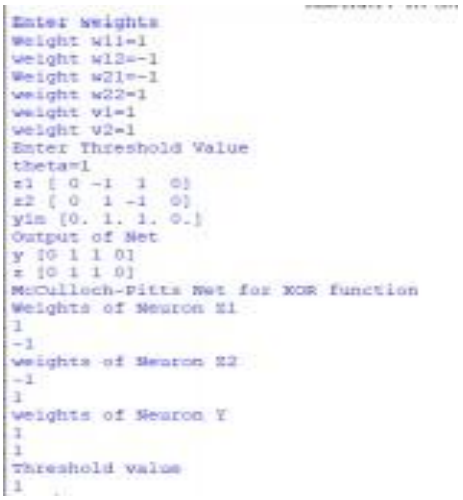
| Input1 | Input2 | XOR Output |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

#Getting weights and threshold value

```
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)
```

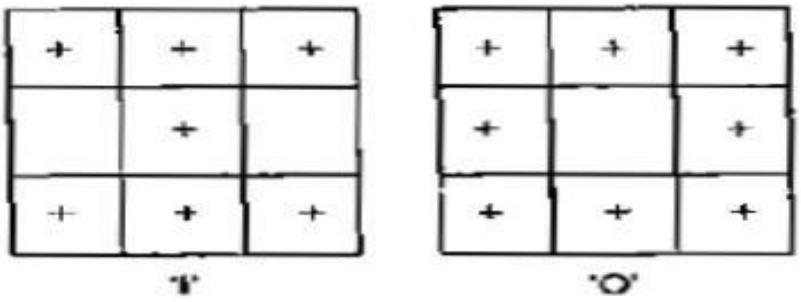
## Output :



# Practical 3

**Practical 3 a: 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).




```
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 1
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

```
#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])
```

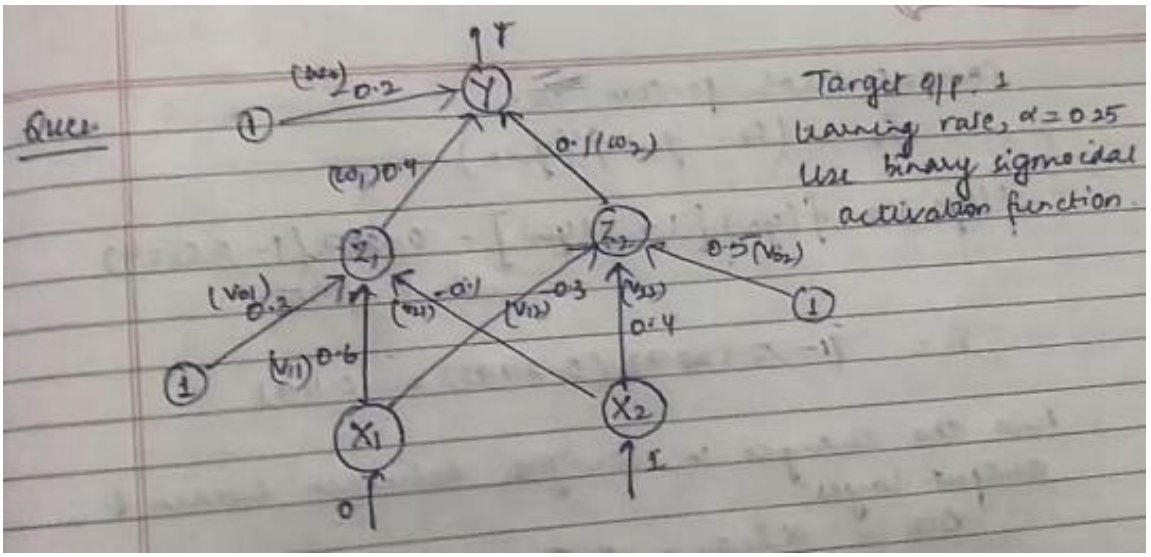
```
actual=x*weights
print("weights",weights)

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
Desired output:3
Desired output:4
Enter learning rate:1
actual [1. 1. 1.]
desired [2. 3. 4.]
weights [2. 3. 4.]
actual [2. 3. 4.]
desired [2. 3. 4.]
******************************
Final output
corrected weights [2. 3. 4.]
actual [2. 3. 4.]
desired [2. 3. 4.]
```

# Practical 4

## Practical 4 a: Write a program for Back Propagation Algorithm




```
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 :

```
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= 0.0119069070741456 94
error in delta
fzin1 0.24751996
fzin2 0.20552119000000002
d1= 0.011788788650865037
d2= 0.0024471217110978417
Changes in weights between input and hidden layer
dv11= 0.0
dv21= 0.0029471971627162592
dv01= 0.0029471971627162592
dv12= 0.0
dv22= 0.0006117804277744604
dv02= 0.0006117804277744604
Final weights of network
v= [0.6 0.3]
v2 [-0.1  0.4]
w= [-0.17  0.42  0.12]
bias b1= 0.30294719716271623  b2= 0.5006117804277744
```

## Practical 4 b: Write a Program For Error Back Propagation Algorithm (Ebpa) Learning

```
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.

```
#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*);
};
```

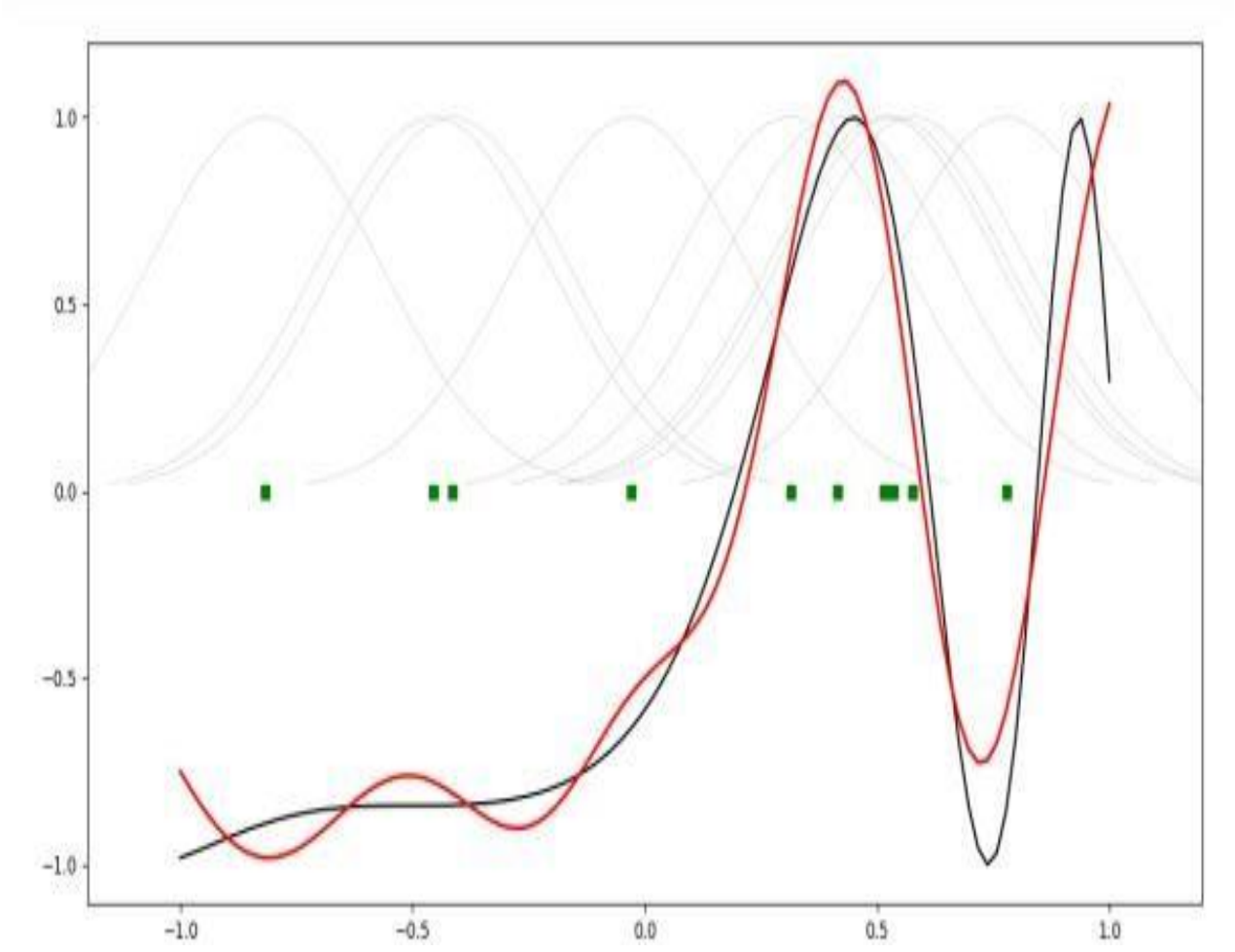
## Practical 5 b: Write a program for Radial Basis function

```
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()
```

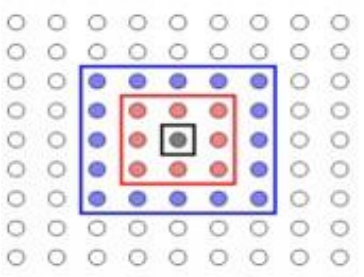
**Output :**



# Practical 6

## Practical 6 a: Self-Organizing Maps

The SOM algorithm is used to compress the information to produce a similarity graph while preserving the topologic relationship of the input data space. The basic SOM model construction algorithm can be interpreted as follows: 1) Create and initialize a matrix (weight vector) randomly to hold the neurons. If the matrix can be initialized with order and roughly compiles with the input density function, the map will converge quickly 2) Read the input data space. For each observation (instance), use the optimum fit approach, which is based on the Euclidean distance c = $argi$ min || x – $mi$ || to find the neuron which best matches this observation. Let x denote the training vector from the observation and $mi$ denote a single neuron in the matrix. Update that neuron to resemble that observation using the following equation: $mi\ (t + 1) = mi\ (t) + h(t)[x(t) - mi\ (t)]$ (4) $mi(t)$: the weight vector before the neuron is updated. $(t + 1)$: the weight vector after the neuron is updated. $(t)$: the training vector from the observation. $h(t)$: the neighborhood function (a smoothing kernel defined over the lattice points), defined though the following equation: $h(t) = \{\ \alpha(t), i \in Nc\ 0, i \in Nc$ (5) : the neighborhood set, which decreases with time. $(t)$: the learning-rate factor which can be linear, exponential or inversely proportional. It is a monotonically decreasing function of time (t)



In general, SOMs might be useful for visualizing high-dimensional data in terms of its similarity structure. Especially large SOMs (i.e. with large number of Kohonen units) are known to perform mappings that preserve the topology of the original data, i.e. neighboring data points in input space will also be represented in adjacent locations on the SOM. The following code shows the 'classic' color mapping example, i.e. the SOM will map a number of colors into a rectangular area.

**frommvpa2.suiteimport***

First, we define some colors as RGB values from the interval (0,1), i.e. with white being (1, 1, 1) and black being (0, 0, 0). Please note, that a substantial proportion of the defined colors represent variations of 'blue', which are supposed to be represented in more detail in the SOM.

```
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']
```

Now we can instantiate the mapper. It will internally use a so-called Kohonen layer to map the data onto. We tell the mapper to use a rectangular layer with 20 x 30 units. This will be the output space of the mapper. Additionally, we tell it to train the network using 400 iterations and to use custom learning rate.

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

Finally, we train the mapper with the previously defined ‘color’ dataset.

```
som.train(colors)
```

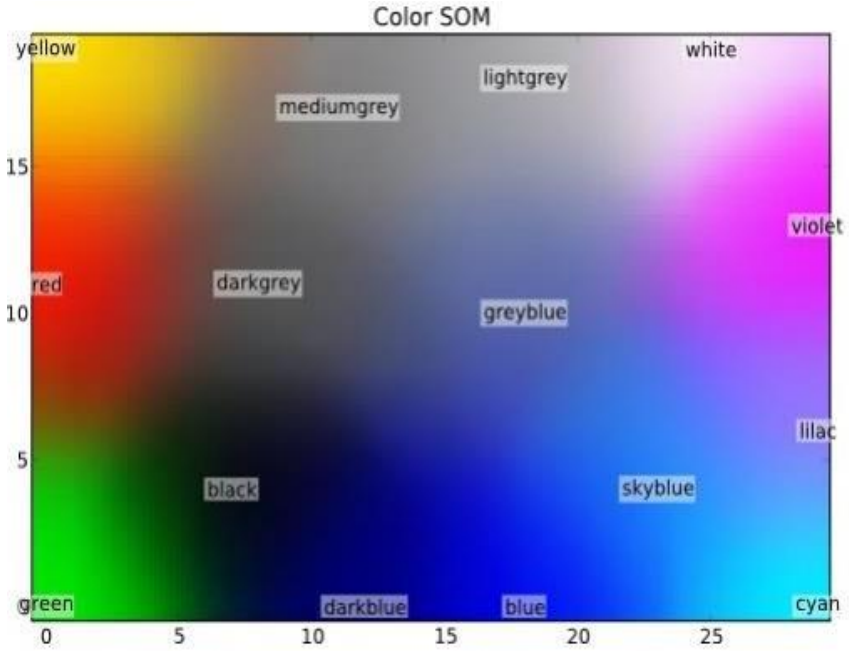
Each unit in the Kohonen layer can be treated as a pointer into the high-dimensional input space, that can be queried to inspect which input subspaces the SOM maps onto certain sections of its 2D output space. The color-mapping generated by this example’s SOM can be shown with a single matplotlib call:

```
pl.imshow(som.K,origin='lower')
```

And now, let’s take a look onto which coordinates the initial training prototypes were mapped to. The get those coordinates we can simply feed the training data to the mapper and plot the output

```
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 :




## Practical 6 b: ADAPTIVE RESONANCE THEORY

```
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)
```

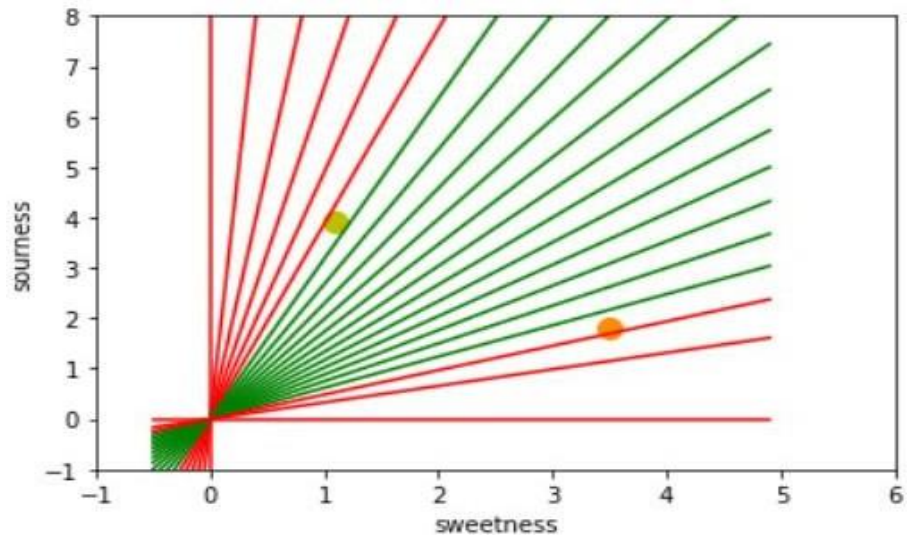
# Practical 7

## Practical 7 a: Line Separation

```
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)
 #print("x: ", x, "slope: ", slope)
 Y = slope * X
 results = []
```

```
 for point in points:
 results.append(dist4line1(*point))
 #print(slope, results)
 if (results[0][1] != results[1][1]):

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

## Output :



# Practical 8

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

```
# Python program to illustrate

# Finding common member in

list # using 'in' operator

list1=[1,2,3,4,5]

list2=[6,7,8,9]

for item in list1:

 if item in list2:

  print("overlapping")

 else:
print("not overlapping")
```

## Output:

```
not overlapping
not overlapping
not overlapping
not overlapping
not overlapping
```

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

```
# 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 :

```
true
```

# Practical 9

## Practical 9 a: Find the ratios using fuzzy logic

```
pip install fuzzywuzzy
pip install python-Levenshtein
# 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: 84
FuzzyWuzzy PartialRatio: 85
FuzzyWuzzy TokenSortRatio: 84
FuzzyWuzzy TokenSetRatio: 86
FuzzyWuzzy WRatio: 84
List of ratios:
[('g. for geeks', 95), ('geek for geek', 93), ('geek geek', 86)]
Best among the above list: ('g. for geeks', 95)
```

## Practical 9 b: Solve Tipping Problem using fuzzy logic

```
import numpy as np
import skfuzzy as fuzz
from skfuzzy import control as ctrl
# New Antecedent/Consequent objects hold universe variables and membership
# functions
quality = ctrl.Antecedent(np.arange(0, 11, 1), 'quality')
service = ctrl.Antecedent(np.arange(0, 11, 1), 'service')
tip = ctrl.Consequent(np.arange(0, 26, 1), 'tip')
# Auto-membership function population is possible with .automf(3, 5, or 7)
quality.automf(3)
```

```
service.automf(3)
# Custom membership functions can be built interactively with a familiar,
# Pythonic API
tip['low'] = fuzz.trimf(tip.universe, [0, 0, 13])
tip['medium'] = fuzz.trimf(tip.universe, [0, 13, 25])
tip['high'] = fuzz.trimf(tip.universe, [13, 25, 25])
"""
To help understand what the membership looks like, use the ``view`` methods.
"""
```

```
# You can see how these look with .view()
quality['average'].view()
"""

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

## Output :

# Practical 10

## Practical 10 a: Implementation of simple genetic algorithm

```
import random
# Number of individuals in each generation
POPULATION_SIZE =100
# Valid genes
GENES ='"abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOP
QRSTUVWXYZ 1234567890, .-;:_!"#%&/()=?@${[]}'"
# Target string to be generated
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
 '''
 # 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 :

```
Generation: 1 String: tO{"-?=jH[k8=B4]Oe@} Fitness: 18
Generation: 2 String: tO{"-?=jH[k8=B4]Oe@} Fitness: 18
Generation: 3 String: .#lRWf9k_Ifslw #O$k_ Fitness: 17
Generation: 4 String: .-1Rq?9mHqk3Wo]3rek_ Fitness: 16
Generation: 5 String: .-1Rq?9mHqk3Wo]3rek_ Fitness: 16
Generation: 6 String: A#ldW) #lIkslwcVek) Fitness: 14
Generation: 7 String: A#ldW) #lIkslwcVek) Fitness: 14
Generation: 8 String: (, o x _x%Rs=, 6Peek3 Fitness: 13
Generation: 29 String: I lope Geeks#o, Geeks Fitness: 3
Generation: 30 String: I loMeGeeksfoBGeeks Fitness: 2
Generation: 31 String: I love Geeksfo0Geeks Fitness: 1
Generation: 32 String: I love Geeksfo0Geeks Fitness: 1
Generation: 33 String: I love Geeksfo0Geeks Fitness: 1
Generation: 34 String: I love GeeksforGeeks Fitness: 0
```

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

first create a City class that will allow us to create and handle our cities.
Create Population

```
import numpy as np, random, operator, pandas as pd, 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 :