PRACTICAL NO – 01

Aim: Implement the following.

1A) Design a simple linear neural network model.

Code :

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

Output :

```
Enter value for weight:0.3
Enter value for bias:0.2
Enter value for input:0.2
The Net input is 0.26
```

1B) Calculate the output of a neural network using both binary and bipolar activation functions.

Code :

```
# w will take weight & x will take the input
w = [ ]
x = [ ]
n = int(input("Enter the number of input neurons: "))
bias=float(input("Enter bias: "))
# taking the value of input and their weight
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 input are: ",x)
print("The given weight are: ",w)
print("The Bias value is: ",bias)
yin=0.0
yin=yin+bias
for i in range(0,n):
   yin = yin + (w[i]*x[i])

print("The net input is ",yin)
#Binary
thres=float(input("enter threshold value for binary and bipolar :"))
```

```
if(yin>=thres):
   output_binary=1
else:
   output_binary=0
print("Final Output(binary)is",output_binary)
#Bipolar
if(yin>=thres):
   output_bipolar=1
else:
   output_bipolar=-1
print("Final Output(bipolar)is",output_bipolar)
```

Output:

```
Enter the number of input neurons: 3
Enter bias: 0.1
Enter the input: 1
Enter the weight: 0.1
Enter the input: 2
Enter the weight: 0.2
Enter the input: 3
Enter the weight: 0.3
The given input are:  [1.0, 2.0, 3.0]
The given weight are:  [0.1, 0.2, 0.3]
The Bias value is:  0.1
The net input is  1.5
enter threshold value for binary and bipolar :1
Final Output(binary)is 1
Final Output(bipolar)is 1
```

PRACTICAL NO – 02

Aim: Implement the following.

2A) Generate AND/NOT function using McCulloch-Pitts neural network.

Code :

```
num_ip=int(input("Enter the number of inputs:"))
print("For the",num_ip,"inputs calculate the net input using yin=x1w1+x2w2")

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("Case1: For calculating the net input we will take weights 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 its not possible to fire neuron from the given inputs so these weights are not suitable")
print("Case2: For calculating the net input we will take weights 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 neuron from the given inputs so these weights are not suitable")
```

Output:

```
======================== RESTART: E:/RIC/Sc-Prac2.py ========================
Enter the number of inputs:4
For the 4 inputs calculate the net input using yin=x1w1+x2w2
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
Case1: For calculating the net input we will take weights 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 its not possible to fire neuron from the given inputs so these weights are not suitable
Case2: For calculating the net input we will take weights 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 neuron from the given inputs so these weights are  suitable
>>> 
```

2B) Generate XOR function using McCulloch-Pitts neural network.

Code :

```
import pandas as pd
print("Enter Weights")
w11=int(input('Weights w11='))
w12=int(input('Weights w12='))
w21=int(input('Weights w21='))
w22=int(input('Weights w22='))
v1=int(input('Weights v1='))
v2=int(input('Weights v2='))
print('Enter threshold value')
theta=int(input('theta='))
import numpy as np
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,))
print(x1)
print(x2)
print(z)
print(y1)
print(y2)
print(y)
while con==1:
    zin1=np.zeros((4,))
    zin2=np.zeros((4,))
    zin1=x1*w11+x2*w12
    zin2=x1*w21+x2*w22
    print(zin1)
    print(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 able to learn use another set of weigths and thresholds")
        print("Enter Weights")
        w11=int(input('Weights w11='))
        w12=int(input('Weights w12='))
        w21=int(input('Weights w21='))
        w22=int(input('Weights w22='))
        v1=int(input('Weights v1='))
        v2=int(input('Weights v2='))
        theta=int(input('theta='))
print("MP Output:")
print("Weights of neuron z1")
print(w11)
print(w21)
print("Weights of neuron z2")
print(w21)
print(w22)
print("Weights of neuron Y")
print(v1)
print(v2)
```

```
print("Threshold")
print(theta)
```

Output:

```
Enter Weights
Weights w11=1
Weights w12=-1
Weights w21=-1
Weights w22=1

Weights v1=1
Weights v2=1

Enter threshold value
theta=1
 [0 0 1 1]
 [0 1 0 1]
 [0 1 1 0]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]

[ 0 -1  1  0]
[ 0  1 -1  0]
yin [0. 1. 1. 0.]
Output of Net:
y [0 1 1 0]
z [0 1 1 0]

MP Output:
Weights of neuron z1
1
-1
Weights of neuron z2
-1
1
Weights of neuron Y
1
1
Threshold
1
```

PRACTICAL NO – 03

Aim: Implement the following.

3A) Write a program to implement Hebb's rules.

Code :

```
import numpy as np
x1=np.array([1,1,1,-1,1,-1,1,1,1])
x2=np.array([1,1,1,1,-1,1,1,1,1])
b=0
y=np.array([1,-1])
wtold=np.zeros([9,])
wtnew=np.zeros([9,])
wtold=wtold.astype(int)
wtnew=wtnew.astype(int)
print("Expected output:",y)
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]
wtnew=wtold
b=b+y[1]
print("New weights are",wtnew)
print("Final bias is:",b)
```

Output :

```
Expected output: [ 1 -1]
First input with target =1
Second input with target =-1
New weights are [ 0  0  0 -2  2 -2  0  0  0]
Final bias is: 0
```

3B) Write a program to implement the delta rule.

Code :

```
import numpy as np
import time
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("Inital inputs:"))
for i in range(0,3):
   weights[i]=float(input("Inital 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
   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 :

```
Inital inputs:1
Inital inputs:1
Inital inputs:1
Inital weights:1
Inital weights:1
Inital 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 NO – 04

Aim: Implement the following.

4A) Write a program for Backpropagation algorithm.

Code:

```
import numpy as np
import decimal
import math
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 : ",zin1)
print('Calculate net input to z2 layer')
zin2 = round(b2+x1*v1[1]+x2*v2[1],4)
print("z2 : ",zin2)
print('Now applying activation function')
z1 = 1/(1+math.exp(-zin1))
z1 = round(z1,4)
z2 = 1/(1+math.exp(-zin2))
z2 = round(z2,4)
print('z1 : ',z1,'\n z2 : ',z2)
```

```
print('Calculate net input to output layer')
yin = w[0]+z1*w[1]+z2*w[2]
print('yin : ',yin)
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('Comnputing 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
```

```
dv21 = alpha*d1*x2
dv01 = alpha*d1
dv12 = alpha*d2*x1
dv22 = alpha*d2*x2
dv02 = alpha*d2
print('dv11 : ',dv11)
print('dv21 : ',dv21)
print('dv01 : ',dv01)
print('dv12 : ',dv12)
print('dv22 : ',dv22)
print('dv02 : ',dv02)
print('Final weight 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[0] = w[0]+dw0
w[1] = w[1]+dw1
w[2] = w[2]+dw2
b1 = b1 + dv01
b2 = b2 + dv01
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
Now applying activation function
z1 :  0.5498
 z2 :  0.7109
Calculate net input to output layer
yin :  0.09101
y =  0.5227368084248941
dk :  0.11906907074145694
Comnputing error portion in delta
din1 :  0.04762762829658278
din2 :  0.011906907074145694
Error in delta
fzin1 :  0.24751996
fzin2 :  0.20552119000000002
d1 :  0.011788788650865037
d2 :  0.002447121711097841 7
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 weight of network
v1 :  [0.6 0.3]
v2 :  [-0.0970528   0.40061178]
w :  [-0.17023273  0.41636604  0.12116155]
bias b1 =  0.30294719716271623  b2 =  0.5029471971627163
```

4B) Write a program for the Error Backpropagation algorithm.

Code:

```
import math
a0 = -1
t = -1
w10 = float(input("Enter weight for first network : "))
b10 = float(input("Enter bias for first network : "))
w20 = float(input("Enter weight for second network : "))
b20 = float(input("Enter bias for second network : "))
c = float(input("Enter learning Coefficient : "))
n1 = float(w10*c+b10)
a1 = math.tanh(n1)
n2 = float(w20*c+b20)
a2 = math.tanh(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('Updated weight at first network w11 ',w11)
print('Updated weight at Second network w21 ',w21)
print('Updated bias at first network b10 ',b10)
print('Updated bias at first network b20 ',b20)
```

Output:

```
Enter weight for first network : 11
Enter bias for first network : 0.5
Enter weight for second network : 12
Enter bias for second network : 0.7
Enter learning Coefficient : 12
Updated weight at first network w11  11.0
Updated weight at Second network w21  12.0
Updated bias at first network b10  0.5
Updated bias at first network b20  0.7
```

PRACTICAL NO – 05

Aim: Implement the following.

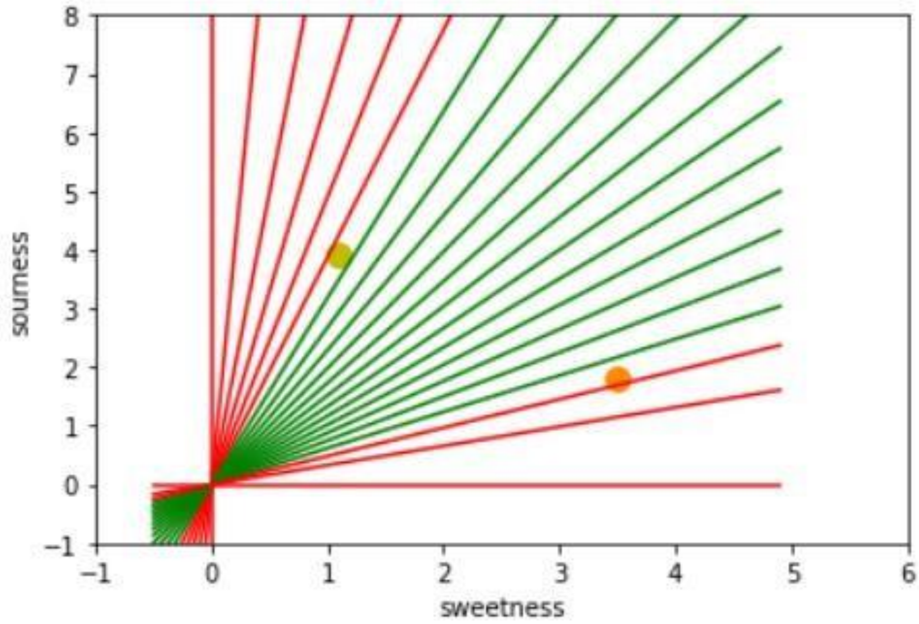
5A) Write a program for Linear separation.

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):
      """return 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 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-")
```

Output:



5B) Write a program for Hopfield network model for associative memory.

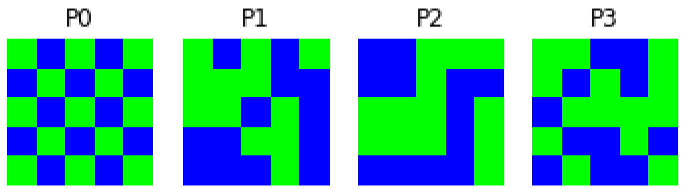
Code:

To install package

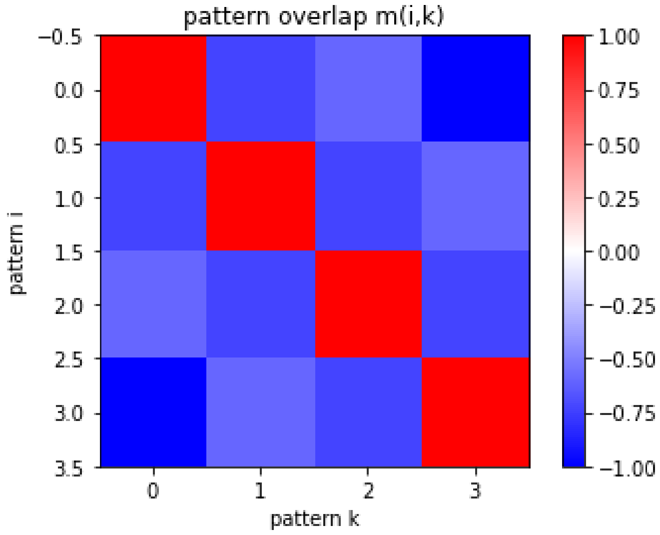
```
!pip install neurodynex3

from neurodynex3.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)

factory

checkboard=factory.create_checkerboard()

pattern_list=[checkboard]

pattern_list.extend(factory.create_random_pattern_list(nr_patterns=3,on_pro
bability=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(checkboard,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,r
eference_idx=0,suptitle="Network Dynamics")
```

Output:

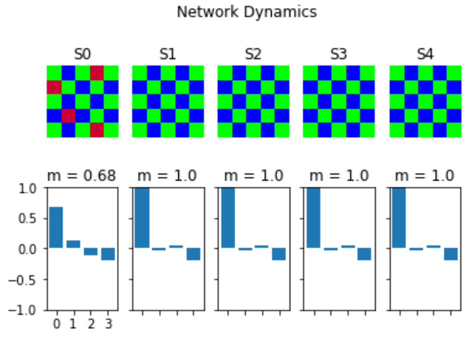
In [12]:
```
plot_tools.plot_pattern_list(pattern_list)
```



In [11]:
```
overlap_matrix=pattern_tools.compute_overlap_matrix(pattern_list)
plot_tools.plot_overlap_matrix(overlap_matrix)
```



In [19]:
```
plot_tools.plot_state_sequence_and_overlap(states_as_patterns,pattern_list,reference_idx=0,suptitle="Network Dynamics")
```



PRACTICAL NO – 06

Aim: Implement the following

6A) Membership and Identify operation IN, NOT IN.

Code:

```
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 :

```
========================= RESTART: E:/RIC/Sc-Prac.py =========================
not overlapping
>>> |
```

6B) Membership and Identify operation IS, IS NOT.

Code:

```
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 :

```
...
========================= RESTART: E:/RIC/prac8b.py =========================
true
true
>>> 
```

PRACTICAL NO – 07

Aim: Implement the following.

7A) Find ratio using fuzzy logic.

Code:

```
from fuzzywuzzy import fuzz
from fuzzywuzzy import process
s1="I Love ArtificaialIntelligenece"
s2="I am loving ArtificialIntelligence"
print("Fuzzywuzyy Ratio:",fuzz.ratio(s1,s2))
print("Fuzzywuzyy Ratio:",fuzz.partial_ratio(s1,s2))
print("Fuzzywuzyy Ratio:",fuzz.token_sort_ratio(s1,s2))
print("Fuzzywuzyy Ratio:",fuzz.token_set_ratio(s1,s2))
print("Fuzzywuzyy Ratio:",fuzz.WRatio(s1,s2))

query="artificial intelligence"
choices=["artificial intelligence","arts intelligence"," a intelligence"]
print("List of ratios")
print(process.extract(query,choices),'\n')
print("Best among the above list: ",process.extractOne(query,choices))
```

Output :

```
Fuzzywuzyy Ratio: 89
Fuzzywuzyy Ratio: 90
Fuzzywuzyy Ratio: 89
Fuzzywuzyy Ratio: 91
Fuzzywuzyy Ratio: 89
List of ratios
[('artificial intelligence', 100), (' a intelligence', 86), ('arts intelligence', 80)]

Best among the above list:  ('artificial intelligence', 100)
>>> 
```

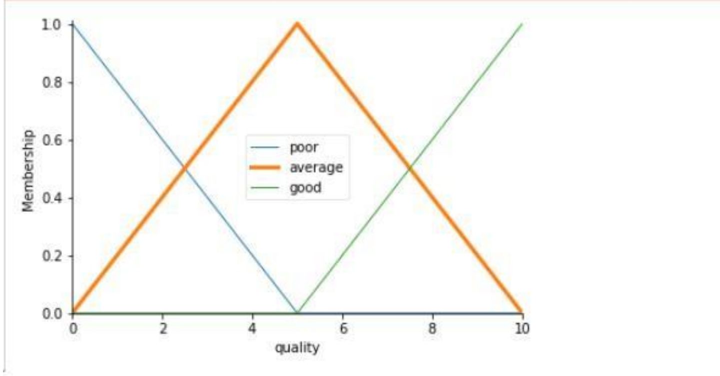
7B) Solve Tipping problem using fuzzy logic.

Code:

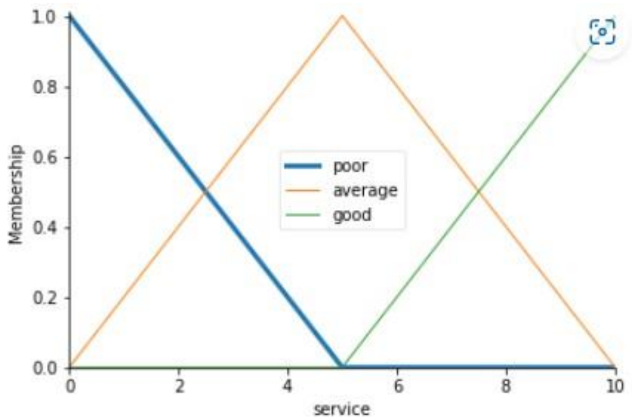
```
!pip install scikit-fuzzy

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) # automembership function
service.automf(3)
tip['low']=fuzz.trimf(tip.universe,[0,0,13]) #trigural membership function
tip['medium']=fuzz.trimf(tip.universe,[0,13,25])
tip['high']=fuzz.trimf(tip.universe,[13,25,25])
quality['average'].view()
service['poor'].view()
tip['high'].view()
rule1=ctrl.Rule(quality['poor']|service['poor'],tip['low'])
rule2=ctrl.Rule(quality['average']|service['average'],tip['medium'])
rule3=ctrl.Rule(quality['good']|service['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'])
```

Output:

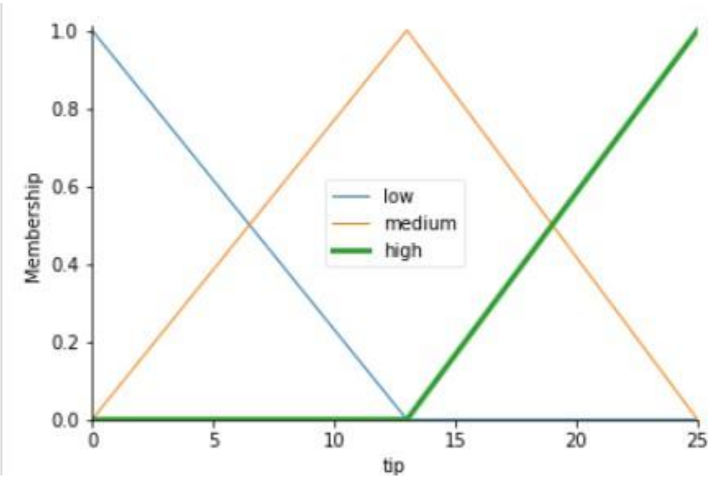
Quality(average):



Service(poor) :



tip(high) :



Output:

14.79822137450634

PRACTICAL NO – 08

Aim: Implementation of genetic algorithm.

Code:

To install package

```
!pip install pygad

import pygad

X=[4,-2,3.5,5,-11,-4.7]

desired_y=44

print(X)

def fitness_function(solution,solution_idx):

   output=numpy.sum(solution*X)

   fitness=1.0/numpy.abs(output-desired_y)

   return fitness

fitness_function=fitness_function

num_generations=50

num_parents_mating=4

sol_per_pop=8

num_genes=len(X)

init_range_low=-2

init_range_high=5

parent_selection_type="sss"

keep_parents=1

crossover_type="single_point"

mutation_type="random"

mutation_percent_genes=10
```

```
ga_instance=pygad.GA(num_generations=num_generations,
            num_parents_mating=num_parents_mating,
            fitness_func=fitness_function,
            sol_per_pop=sol_per_pop,
            num_genes=num_genes,
            init_range_low=init_range_low,
            init_range_high=init_range_high,
            parent_selection_type=parent_selection_type,
            keep_parents=keep_parents,
            crossover_type=crossover_type,
            mutation_type=mutation_type,
            mutation_percent_genes=mutation_percent_genes)
import numpy
ga_instance.run()
solution,solution_fitness,solution_idx=ga_instance.best_solution()
print(solution)
print(solution_fitness)
print(solution_idx)
```

Output:

Solution:

```
print(solution)
```

```
[ 5.19278599  2.97604883  1.39789488  1.62158125 -0.88196618 -1.37562192]
```

Solution_fitness:

```
print(solution_fitness)
```

```
74.82549530318296
```

Solution_idx:

```
print(solution_idx)
```

```
0
```

: Adaptive Resonance Theory

CODE:

```
import datetime as dt
import math
import sys
now = dt.datetime.now()
print("Executed by Anu\nRoll No. : 7")
print("Current Date and Time : "+ now.strftime("%d-%m-%Y %H:%M:%S"))
print()
N = 4
M = 5
VIGILANCE = 0.4
PATTERN_ARRAY = [[1, 1, 0, 0], [0, 0, 0, 1], [1, 0, 0, 0], [0, 0, 1, 1],
 [0, 1, 0, 0], [0, 0, 1, 0], [1, 0, 1, 0]]
class ART1:
 def __init__(self, inputSize, numClusters, vigilance):
   self.mInputSize = inputSize
   self.mNumClusters = numClusters
   self.mvigilance = vigilance
   self.bottom_up_weights = []
   self.top_down_weights = []
   self.inputLayer = []
   self.interface_layer = []
   self.f2 = []
 #Initialize bottom-up weight matrix
   for i in range(self.mNumClusters):
     self.bottom_up_weights.append([0.0] * self.mInputSize)
     for j in range(self.mInputSize):
       self.bottom_up_weights[i][j] = 1.0 / (1.0 + self.mInputSize)
 #Initialize top-down weight matrix
   for i in range(self.mNumClusters):
     self.top_down_weights.append([0.0] * self.mInputSize)
     for j in range(self.mInputSize):
       self.top_down_weights[i][j] = 1.0
     self.inputLayer = [0.0] * self.mInputSize
     self.interface_layer = [0.0] * self.mInputSize
```

```
    self.f2 = [0.0] * self.mNumClusters
  def get_vector_sum(self, nodeArray):
    total = 0
    for i in range(len(nodeArray)):
      total += nodeArray[i]
    return total
  def get_maximum(self, nodeArray):
    maximum = -1
    maxValue = -0
    unique = True
    for i in range(len(nodeArray)):
      if nodeArray[i] > maxValue:
        maximum = i
        maxValue = nodeArray[i]
    return maximum

  def test_for_reset(self, activationSum, inputSum, f2max):
    if(inputSum == 0): return False
    elif(float(activationSum) / float(inputSum) >= self.mvigilance):
      return False
    else:
      self.f2[f2max] = -1.0
      return True
  def update_weights(self, activationSum, f2max):
    for i in range(self.mInputSize):
      self.bottom_up_weights[f2max][i] = (2.0 * float(self.interface_layer[i])) / (1.0 + float(activationSum))
    for i in range(self.mInputSize):
      self.top_down_weights[f2max][i] = self.interface_layer[i]
  def ART1(self, trainingPattern, isTraining):
    inputSum = 0
    activationSum = 0
    f2max = 0
    reset = True
    for i in range(self.mNumClusters):
      self.f2[i] = 0.0
    for i in range(self.mInputSize):
      self.inputLayer[i] = float(trainingPattern[i])
```

```
    inputSum = self.get_vector_sum(self.inputLayer)
    for i in range(self.mInputSize):
      self.interface_layer[i] = self.inputLayer[i]

    for i in range(self.mNumClusters):
      for j in range(self.mInputSize):
        self.f2[i] += self.bottom_up_weights[i][j] * float(self.inputLayer[j])
    reset = True
    while reset:
      f2Max = self.get_maximum(self.f2)
      if f2Max == -1:
        f2Max = self.mNumClusters
        self.f2.append(0.0)
        self.top_down_weights.append([1.0] * self.mInputSize)
        self.bottom_up_weights.append([1.0 / (1.0 + self.mInputSize)] * self.mInputSize)
        self.mNumClusters += 1

      for i in range(self.mInputSize):
        self.interface_layer[i] = self.inputLayer[i] * math.floor(self.top_down_weights[f2Max][i])
      activationSum = self.get_vector_sum(self.interface_layer)
      reset = self.test_for_reset(activationSum, inputSum, f2Max)
      if isTraining:
        self.update_weights(activationSum, f2Max)
      return f2Max
  def print_results(self):
    sys.stdout.write("Clusters found : "+str(self.mNumClusters)+"\n")
    sys.stdout.write("Rho : "+str(self.mvigilance)+"\n")
    sys.stdout.write("Final weight values : \n")
    for i in range(self.mNumClusters):
      for j in range(self.mInputSize):
        sys.stdout.write(str(self.bottom_up_weights[i][j])+",")
      sys.stdout.write("\n")
    sys.stdout.write("\n")
    for i in range(self.mNumClusters):
      for j in range(self.mInputSize):
        sys.stdout.write(str(self.top_down_weights[i][j])+",")
      sys.stdout.write("\n")
```

```
  sys.stdout.write("\n")
  return
if __name__ == '__main__':
  net = ART1(N,M,VIGILANCE)
  for line in PATTERN_ARRAY:
    net.ART1(line, True)
  for line in PATTERN_ARRAY:
    print(str(line)+','+str(net.ART1(line, True)))
```

AIM A: Kohonen Self organizing map

CODE:

```
import datetime as dt
from minisom import MiniSom
from matplotlib import pyplot as plt
now = dt.datetime.now()
print("Executed by Anu\nRoll No. : 7")
print("Current Date and Time : "+ now.strftime("%d-%m-%Y %H:%M:%S"))
print()
#find out the input features
data = [[0.80, 0.55, 0.22, 0.03, 0.21], [0.82, 0.50, 0.23, 0.03, 0.21],
 [0.80, 0.54, 0.22, 0.03, 0.21], [0.80, 0.53, 0.26, 0.03, 0.21],
 [0.79, 0.56, 0.22, 0.03, 0.21], [0.75, 0.60, 0.25, 0.03, 0.21],
 [0.77, 0.59, 0.22, 0.03, 0.21]]
som = MiniSom(6, 6, 5, sigma=0.3, learning_rate=0.2)
som.train_random(data, 100)
plt.imshow(som.distance_map()
```