Name: Adarsh Kushwaha

Roll no:

Subject: Soft Computing(Practicals)

Practical 1a

Aim: Design a simple linear neural network model.

Code:

```
x = float(input("enter the vlaue of x :"))
w = float(input("enter the value of w :"))
b = float(input("enter the vlaue of bias :"))
net = (x*w+b)
if(net<0):
   out = 0
elif((net>=0) & (net <=1)):
   out = net
else:
   out = 1
print("net:",net)
print("output:",out)
```

Output:

```
================= RESTART: D:\SOFT COMPUTING\neuralnetmodels.py ==
enter the vlaue of x :2
enter the value of w :1
enter the vlaue of bias :1
net: 3.0
output: 1
```

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

Code:

```
n = int(input("enter the number of inputs:"))
inputs=[]
print("enter the inputs")
```

```
for i in range(0,n):
    elements = float(input())
    inputs.append(elements)
print(inputs)
print("enter the weights :")
weights =[]
for i in range(0,n):
    weight = float(input())
    weights.append(weight)
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:

```
==================== RESTART: D:\SOFT COMPUTING\netbinary.py =
enter the number of inputs: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
```

Code:

```
n = int(input("enter the number of inputs:"))
inputs=[]
print("enter the inputs")
```

```
for i in range(0,n):
    elements = float(input())
    inputs.append(elements)
print(inputs)
print("enter the weights :")
weights =[]
for i in range(0,n):
    weight = float(input())
    weights.append(weight)
print(weights)
#b = float(input("enter the bias value :"))
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))
#print(round((sum(Yin)+b),1))
```

Output:

```
==================== RESTART: D:\SOFT COMPUTING\netbinary.py =
enter the number of inputs:2
enter the inputs
0.2
0.36
[0.2, 0.36]
enter the weights :
0.3
0.7
[0.3, 0.7]
enter the bias value :0.45
The net input can be calculated as Yin = x1w1+x2w2+x3w3
0.8
```

Practical 2a:

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

Code:

```
num_ip=int(input("Enter the number of inputs: "))
w1=2
w2=1
print("For the ",num_ip,"inputs calculat 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(m[i]-n[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:

```
=============== RESTART: E:\Soft computing\Soft computing\P2a.py ===============
Enter the number of inputs: 4
For the  4 inputs calculat 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]
```

Practical 2b

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

Code:

```
import numpy as np
print('enter weights')
w11= int(input('weight w11 ='))
w12= int(input('weight w12 ='))
w21= int(input('weight w21 ='))
w22= int(input('weight w22 ='))
v1= int(input('weight v1 ='))
v2= int(input('weight v2='))
print('enter threshold value')
theta = int(input("theta="))
```

```
x1= np.array([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 the another set of weights and Threshold values")
        w11= input("weights w11 = ")
        w12= input("weights w12 = ")
        w21= input("weights w21 = ")
        w22= input("weights w22 = ")
        v1= input("weights v1= ")
        v2= input("weights v2 = ")
        theta=input("theta = ")
print("McCulloch-Pitts Net for XOR function")
print("Weights of Neuron z1")
print(w11)
print(w21)
print("Weights of Neuron z2")
print(w12)
print(w22)
print("weights of Neuron Y")
print(v1)
print(v2)
print("Threshold Value")
print(theta)
```

Output:

```
===================== RESTART: D:\SOFT COMPUTING\prac2b.py ==
enter weights
weight w11 =1
weight w12 =-1
weight w21 =-1
weight w22 =1
weight v1 =1
weight v2=1
enter threshold value
theta=1
z1 [ 0 -1  1  0]
z2 [ 0  1 -1  0]
yin [0. 1. 1. 0.]
Output of Net
y [0 1 1 0]
z [0 1 1 0]
McCulloch-Pitts Net for XOR function
Weights of Neuron z1
1
-1
Weights of Neuron z2
-1
1
weights of Neuron Y
1
1
Threshold Value
1
```

Practical 3a

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

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,))
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 vlaue",b)
print("second input with target = -1")
for i in range(0,9):
    wtnew[i] = wtold[i]+x2[i]*y[1]
b= b+y[1]
print("new wt=",wtnew)
print('bias value',b)
```

Output:

```
===================== RESTART: D:\SOFT COMPUTING\prac3a.py ====
First input with target = 1
new wt = [ 1  1  1 -1  1 -1  1  1  1]
bias vlaue 1
second input with target = -1
new wt= [ 0  0  0 -2  2 -2  0  0  0]
bias value 0
```

Practical 3b

Aim: Write a program to implement of delta rule.

Code:

```
import numpy as np
import time
np.set_printoptions(precision=2)
x=np.zeros((3,))
weights = np.zeros((3,))
desired = np.zeros((3,))
actual= np.zeros((3,))
for i in range(0,3):
  x[i]=float(input("initial inputs:"))
for i in range(0,3):
  weights[i]=float(input("Initial Weights :"))
for i in range(0,3):
  desired[i]=float(input("Desired Output:"))
a= float(input("Enter the 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:

```
===================== RESTART: D:\SOFT COMPUTING\prac3b.py =====
initial inputs:1
initial inputs:1
initial inputs:1
Initial Weights :1
Initial Weights :1
Initial Weights :1
Desired Output:2
Desired Output:3
Desired Output:4
Enter the learning rate:1
actual [1. 1. 1.]
desired [2. 3. 4.]
weights [2. 3. 4.]
actual [2. 3. 4.]
desired [2. 3. 4.]
******************************
Final output
Corrected weights  [2. 3. 4.]
actual [2. 3. 4.]
desired [2. 3. 4.]
```

Practical 4a

Aim: Write a program for Back Propagation Algorithm

Code:

```
import numpy as np
import decimal
import math
np.set_printoptions(precision=2)
v1=np.array([0.6,0.3])
v2=np.array([-0.1,0.4])
w= np.array([-0.2,0.4,0.1])
b1=0.3
b2=0.5
x1=0
x2=1
alpha=0.25
print("calculate net input to z1 layer")
zin1=round(b1+x1*v1[0]+x2*v2[0],4)
print("z1=",round(zin1,3))
print("calculate net input to z2 layer")
zin2 = round(b2+x1*v1[1]+x2*v2[1],4)
print("z2=",round(zin2,4))
print("Apply activation function to calculate output")
z1=1/(1+math.exp(-zin1))
z1=round(z1,4)
z2=1/(1+math.exp(-zin2))
z2=round(z2,4)
print("z1=",z1)
print("z2=",z2)
print("calculate net input to output layer")
```

```
yin= w[0]+z1*w[1]+z2*w[2]
print("yin=",yin)
print("calculate net output")
y= 1/(1+math.exp(-yin))
print("y=",y)
fyin= y*(1-y)
dk=(1-y)*fyin
print("dk=",dk)
dw1=alpha*dk*z1
dw2=alpha*dk*z2
dw0= alpha*dk
print("compute error portion in delta")
din1=dk*w[1]
din2=dk*w[2]
print("din1 =",din1)
print("din2 =",din2)
print("error in delta")
fzin1=z1*(1-z1)
print("fzin1 =",fzin1)
d1=din1*fzin1
fzin2= z2*(1-z2)
print("fzin2 =",fzin2)
d2= din2*fzin2
print("d1=",d1)
print("d2=",d2)
print("Changes in weights between input and hidden layer")
dv11 = alpha*d1*x1
print("dv11=",dv11)
dv21=alpha*d1*x2
```

```
print("dv21=",dv21)
dv01=alpha*d1
print("dv01=",dv01)
dv12 = alpha*d2*x1
print("dv12=",dv12)
dv22=alpha*d2*x2
print("dv22=",dv22)
dv02=alpha*d2
print("dv02=",dv02)
print("Final weights of network")
v1[0]=v1[0]+dv11
v1[1]=v1[1]+dv12
print("v=",v1)
v2[0]=v2[0]+dv21
v2[1]=v2[1]+dv22
print("v2=",v2)
w[1]=w[1]+dw1
w[2]=w[2]+dw2
b1=b1+dv01
b2=b2+dv02
w[0]=w[0]+dw0
print("w=",w)
print("bias b1=", b1, "b2=",b2)
```

output:

```
===================== RESTART: D:\SOFT COMPUTING\prac4a.py =
calculate net input to z1 layer
z1= 0.2
calculate net input to z2 layer
z2= 0.9
Apply activation function to calculate output
z1= 0.5498
z2= 0.7109
calculate net input to output layer
yin= 0.09101
calculate net output
y= 0.5227368084248941
dk= 0.11906907074145694
compute error portion in delta
din1 = 0.04762762829658278
din2 = 0.011906907074145694
error in delta
fzin1 = 0.24751996
fzin2 = 0.20552119000000002
d1= 0.011788788650865037
d2= 0.002447121711 0978417
Changes in weights between input and hidden layer
dv11= 0.0
dv21= 0.002947197162 7162592
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 4b

Aim: Write a Program For Error Back Propagation Algorithm (Ebpa) Learning

Code:

import math

a0=-1

t=-1

w10=float(input("Enter weight first network:"))

b10=float(input("Enter base first network:"))

```
w20=float(input("Enter weight second network:"))
b20=float(input("Enter base second network:"))
c= float(input("Enter learning coefficient:"))
n1=float(w10*c+b10)
a1=math.tanh(n1)
n2=float(w20*c+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 weight of first n/w b10 = ",b10)
print("The updated base of second n/w b20 =", b20)
```

Output:

```
===================== RESTART: D:\SOFT COMPUTING\prac4b.py
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 weight of first n/w b10 =  35.0
The updated base of second n/w b20 = 45.0
```

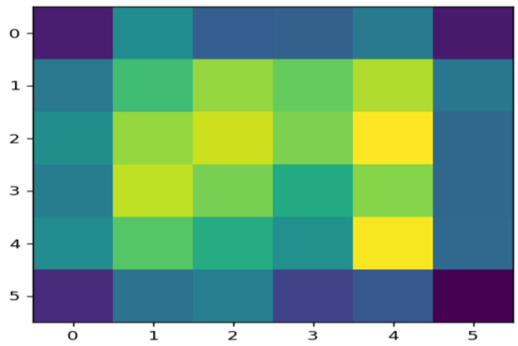
Practical 6a

Aim:Self-Organizing Maps

Code:

```
from minisom import MiniSom
import matplotlib.pyplot as plt
data = [[0.80,0.55,0.22,0.03],
        [0.82,0.50,0.23,0.03],
        [0.80,0.54,0.22,0.03],
        [0.80,0.53,0.26,0.03],
        [0.79,0.56,0.22,0.03],
        [0.75,0.60,0.25,0.03],
        [0.77,0.59,0.22,0.03]]
som =MiniSom(6,6,4,sigma=0.3,learning_rate=0.5)
som.train_random(data,100)
plt.imshow(som.distance_map())
plt.show()
```

Output:



Practical 7a

Aim: Line Separation

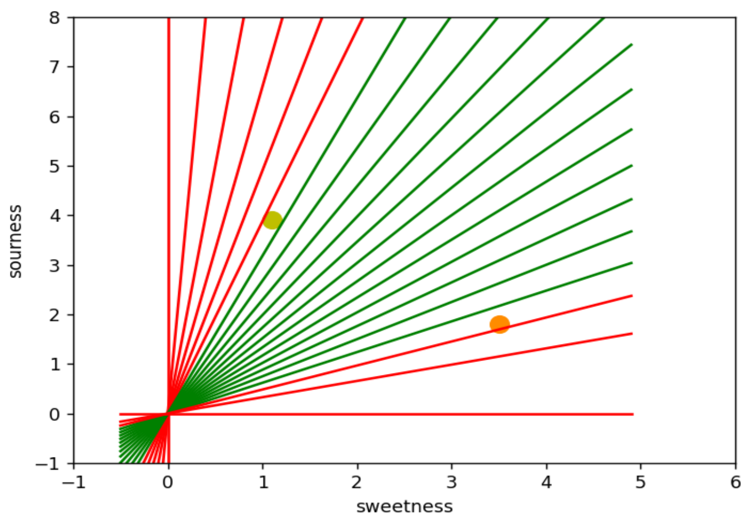
Code:

```
import numpy as np
import matplotlib.pyplot as plt
```

```
def create_distance_function(a,b,c):
    """0=ax+by+c"""
    def distance(x,y):
        """returns tuple(d,pos) d is the distance
        If pos==-1 point is below the line,
        0 on the line and +1 if above the line"""
        nom=a*x+b*y+c
        if nom==0:
            pos=0
        elif (nom<0 and b<0)or(nom>0 and b>0):
            pos=-1
        else:
            pos=1
        return (np.absolute(nom)/np.sqrt(a**2+b**2),pos)
    return distance
points=[(3.5,1.8),(1.1,3.9)]
fig,ax=plt.subplots()
ax.set_xlabel("sweetness")
ax.set_ylabel("sourness")
ax.set_xlim([-1,6])
ax.set_ylim([-1,8])
X=np.arange(-0.5,5,0.1)
colors=["r",""]
size=10
for (index,(x,y)) in enumerate(points):
    if index==0:
        ax.plot(x,y,"o",color="darkorange",markersize=size)
    else:
        ax.plot(x,y,"oy",markersize=size)
```

```
    step=0.05
for x in np.arange(0,1+step,step):
  slope=np.tan(np.arccos(x))
  dist4line1=create_distance_function(slope,-1,0)
  Y=slope*X
  results=[]
  for point in points:
    results.append(dist4line1(*point))
  if(results[0][1]!=results[1][1]):
    ax.plot(X,Y,"g-")
  else:
    ax.plot(X,Y,"r-")
plt.show()
```

Output:



Practical 7b

Aim: Hopfield Network model of associative memory

Code:

```
import matplotlib.pyplot as plt
from neurodynex.hopfield_network import network, pattern_tools, plot_tools
```

```
# Parameters
pattern_size = 10  # Define the size of the patterns
nr_neurons = pattern_size ** 2
# Create an instance of the Hopfield network
hopfield_net = network.HopfieldNetwork(nr_neurons=nr_neurons)
# Instantiate a pattern factory
factory = pattern_tools.PatternFactory(pattern_size, pattern_size)
# Create a checkerboard pattern and a random pattern list
checkerboard = factory.create_checkerboard()
pattern_list = [checkerboard]
pattern_list.extend(factory.create_random_pattern_list(nr_patterns=3,
on_probability=0.5))
# Plot the patterns
plot_tools.plot_pattern_list(pattern_list)
# Compute and plot the overlap matrix
overlap_matrix = pattern_tools.compute_overlap_matrix(pattern_list)
plot_tools.plot_overlap_matrix(overlap_matrix)
# Store patterns in the Hopfield network
hopfield_net.store_patterns(pattern_list)
# Create a noisy version of the checkerboard pattern
noisy_init_state = pattern_tools.flip_n(checkerboard, nr_of_flips=4)
hopfield_net.set_state_from_pattern(noisy_init_state)
# Run the Hopfield network with monitoring
states = hopfield_net.run_with_monitoring(nr_steps=4)
# Reshape the states into patterns
states_as_patterns = factory.reshape_patterns(states)
# Plot the state sequence and overlap
plot_tools.plot_state_sequence_and_overlap(
    states_as_patterns,
    pattern_list,
```

```
    reference_idx=0,
    suptitle="Network Dynamics"
)
# Show plots
plt.show()
```

Output:

ERROR

Practical 8a

Aim: Membership and Identity operators in, not in.

Code:

```
#Aim: Membership and Identity operators in, not in.

# Python program to illustrate
# Finding common member in list
# without using 'in' operator

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 8b: Membership and Identity Operators is, is not

Code:

```
# Python program to illustrate the use
# of 'is' identity operator

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

Output:

True

True

Practical 9a

Aim: Find the ratios using fuzzy logic

Code:

```
from fuzzywuzzy import fuzz

from fuzzywuzzy import process

s1="I love fuzzysforfuzzys"

s2="I am loving fuzzysforfuzzys"

print("FuzzyWuzzy Ratio:",fuzz.ratio(s1,s2))

print("FuzzyWuzzyPartial Ratio:",fuzz.partial_ratio(s1,s2))

print("FuzzyWuzzyTokenSort Ratio:",fuzz.token_sort_ratio(s1,s2))

print("FuzzyWuzzyTokenSet Ratio:",fuzz.token_set_ratio(s1,s2))

print ("FuzzyWuzzyWRatio: ", fuzz.WRatio(s1, s2),'\n\n')

#for process library

query="fuzzy 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
FuzzyWuzzyPartial Ratio: 86
FuzzyWuzzyTokenSort Ratio: 86
FuzzyWuzzyTokenSet Ratio: 87
FuzzyWuzzyWRatio:  86


List of ratios:
[('fuzzy for fuzzy', 97), ('fuzzy fuzzy', 95), ('g.for fuzzys', 86)]

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

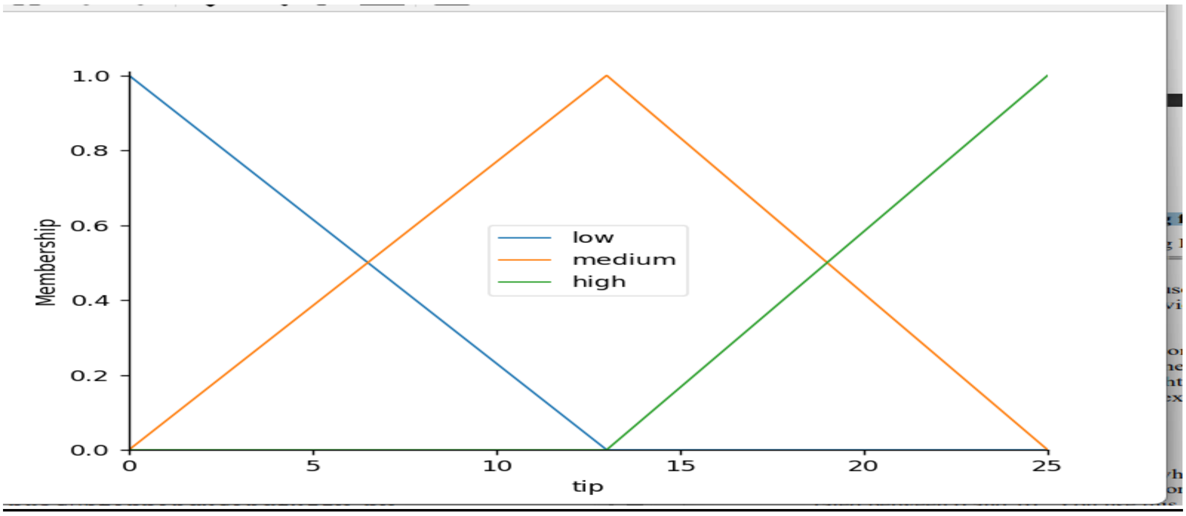
Practical 9b: Solve Tipping Problem using fuzzy logic

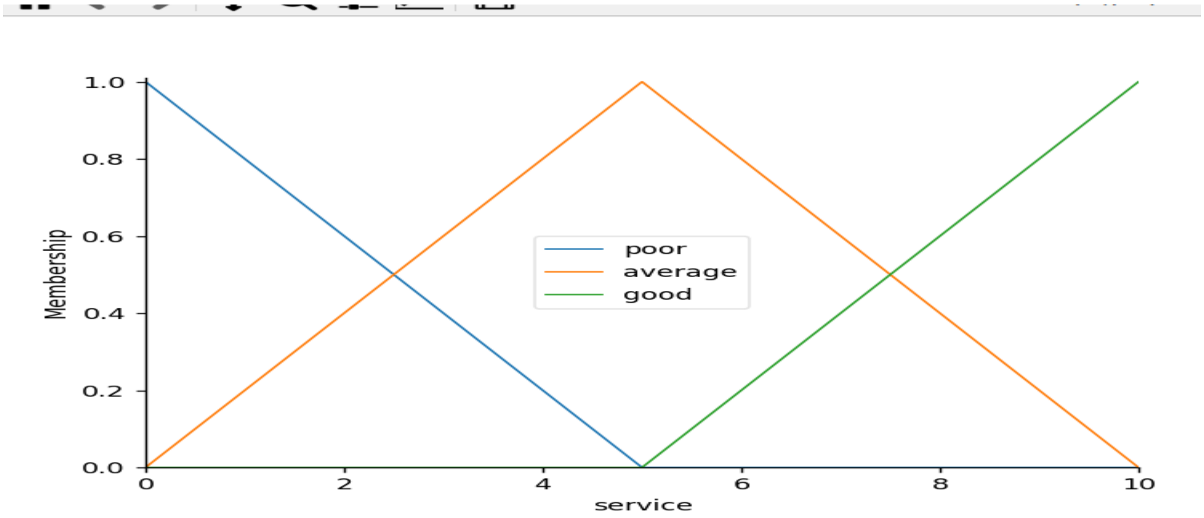
Code:

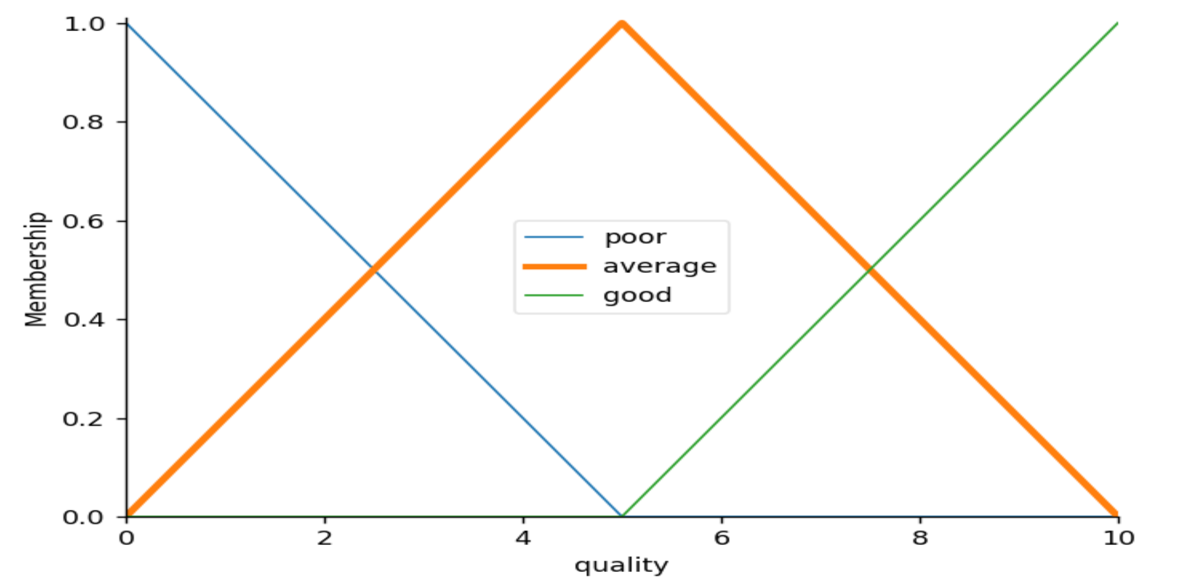
```
import numpy as np

import skfuzzy as fuzz
```

```
from skfuzzy import control as ctrl
quality=ctrl.Antecedent(np.arange(0,11,1),'quality')
service=ctrl.Antecedent(np.arange(0,11,1),'service')
tip=ctrl.Consequent(np.arange(0,26,1),'tip')
quality.automf(3)
service.automf(3)
tip['low']=fuzz.trimf(tip.universe,[0,0,13])
tip['medium']=fuzz.trimf(tip.universe,[0,13,25])
tip['high']=fuzz.trimf(tip.universe,[13,25,25])
quality['average'].view()
service.view()
tip.view()
rule1=ctrl.Rule(qulity['poor']|service['poor'],tip['low'])
rule2=ctrl.Rule(service['average'],tip['medium'])
rule3=ctrl.Rule(service['good']|quality['good'],tip['high'])
rule1.view()
tipping_ctrl=ctrl.ControlSystem([rule1,rule2,rule3])
tipping=ctrl.ControlSystemSimulation(tipping_ctrl)
tipping.input['quality']=6.5
tipping.input['service']=9.8
tipping.compute()
print(tipping.output['tip'])
tip.views(sim=tipping)
```

Output:


1.0
0.8
0.6
0.4
0.2
0.0
Membership
0
5
10
15
20
25
tip
low
medium
high


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


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