INDEX

	B	Adaptive Resonance theory.		
7.		Implement the following		
	A	Write a program for linear separation.		
	B	Write a program for Hopfield network model for associative memory		
8.		Implement the following		
	A.	Membership and Identity operators (in/not in)		
	B.	Membership and Identity operators (is/is not)		
9.		Implement the following		
	A	Find ratios using fuzzy logic		
	B	Solve Tipping problem using fuzzy logic		
10.		Implement the following		
	A	Implementation of simple genetic algorithm.		
	B	Create two classes: City and Fitness using a Genetic Algorithm.		

Practical No: 01

Implement the Following:

Theory:

Neural networks are artificial systems that were inspired by biological neural networks. These systems learn to perform tasks by being exposed to various datasets and examples without any task-specific rules. Neural networks are based on computational models for threshold logic. Threshold logic is a combination of algorithms and mathematics. Neural networks are based either on the study of the brain or on the application of neural networks to artificial intelligence. The work has led to improvements in finite automata theory. Components of a typical neural network involve neurons, connections which are known as synapses, weights, biases, propagation function, and a learning rule.

A. Design a simple neural network model.

Calculate the output of neural net where input X = 0.2, w = 0.3 and bias b 0.45.

Given neural net:

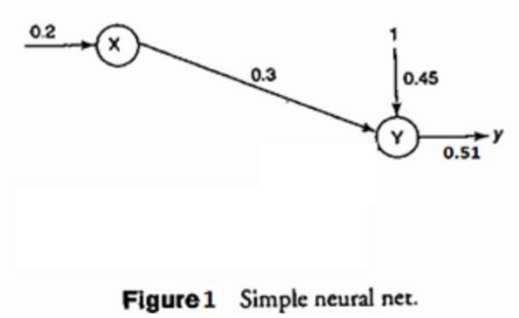


Figure 1 Simple neural net.

Yin = wx + b = 0.3*0.2 + 0.45 = 0.51.

if (yin < 0), then output=y=0

else if (yin >1) then output=y=1

else output=y=yin

Code:

```
inputs = float(input("Enter the input: "))
weights = float(input("Enter the weight: "))
bias = float(input("Enter bias: "))
print(' ')
yin = bias + (inputs * weights)
if yin < 0:
    out = 0
elif yin > 1:
    out= 1
else:
    out = yin
print("Output is: ",out)
print(' ')
print('Nishant-53004230004)
```

Output:

```
Enter the input: 1
Enter the weight: -1
Enter bias: 0

Output is:  0

Nishant -53004230004
```

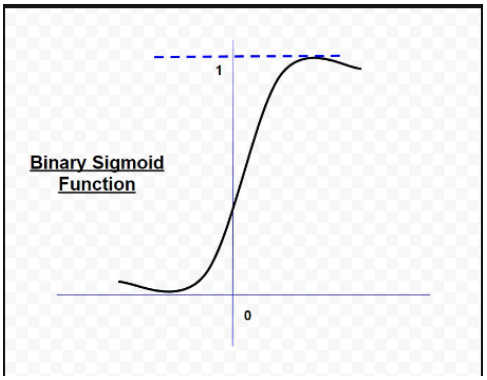
B. Calculate the output of neural net. using both binary and bipolar sigmoidal function.

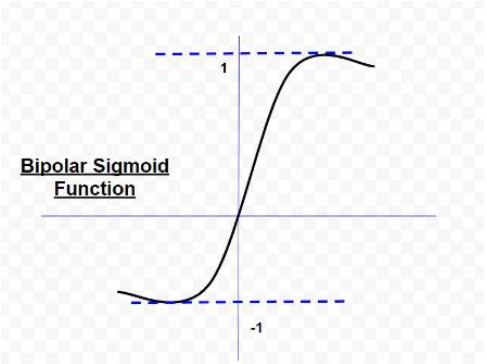
Theory:

- Binary Sigmoid function is by far the most commonly used activation function in neural networks. The need for sigmoid function stems from the fact that many learning algorithms require the activation function to be differentiable and hence continuous.

A binary sigmoid function is of the form: $y_{out} = f(x) = \frac{1}{1+e^{-kx}}$

where k = steepness or slope parameter, by varying the value of k, a sigmoid function with different slopes can be obtained. It has a range of (0,1). The slope of origin is k/4. As the value of k becomes very large, the sigmoid function becomes a threshold function.





- A **bipolar sigmoid function** is of the form: $y_{out} = f(x) = \frac{1-e^{-kx}}{1+e^{-kx}}$

The range of values of sigmoid functions can be varied depending on the application. However, the range of (-1,+1) is most commonly adopted.

Code:

```
import math
```

```
n = int(input("Enter no. of elements: "))
yin = 0
for i in range(0,n):
    x=float(input("x= "))
    w=float(input("w= "))
    yin = yin + (x*w)
b = float(input("b= "))
yin = yin + b
print(' ')
print("yin",yin)
print(' ')
binary_sigmoidal = (1/(1+(math.e**(-yin))))
print("Binary Sigmoidal= ",round(binary_sigmoidal,3))
print(' ')
bipolar_sigmoidal = (2/(1+(math.e**(-yin)))) - 1
print("Bipolar Sigmoidal= ",round(bipolar_sigmoidal,3))
print(' ')
print('Nishant-53004230004)
```

Output:

```
Enter no. of elements: 3
x= 1
w= -1
x= 0
w= 1
x= -1
w= 0
b= 2

yin 1.0

Binary Sigmoidal=  0.731

Bipolar Sigmoidal=  0.462

Nishant - 53004230004
```

Practical No: 02

Theory:

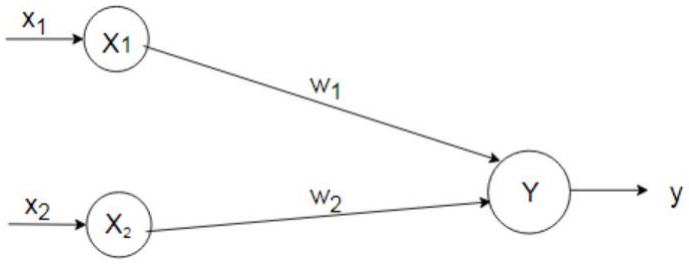
The McCulloch-Pitts neural model, which was the earliest ANN model, has only two types of inputs — Excitatory and Inhibitory. The excitatory inputs have weights of positive magnitude and the inhibitory weights have weights of negative magnitude. The inputs of the McCulloch-Pitts neuron could be either 0 or 1. It has a threshold function as an activation function. So, the output signal yout is 1 if the input ysum is greater than or equal to a given threshold value, else 0.

Implement the Following:

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

Theory:

In McCulloch-Pitts Neuron only analysis is performed. It is a logic gate that implements conjunction. Whenever both the inputs are high then only output will be high (1) otherwise low (0). Hence, assume weights be w1 = w2 =1. The network architecture is



ANDNOT Function:

Truth Table:

X1	X2	Y
1	1	0
1	0	1
0	1	0
0	0	0

Code:

```
print("\nANDNOT function using MP\n")
x1inputs = [1,1,0,0]
x2inputs = [1,0,1,0]
print("Considering all weights as excitatory");
w1 = [1,1,1,1]
w2 = [1,1,1,1]
yin = []
print("x1","x2","yin")
for i in range(0,4):
    yin.append(x1inputs[i]*w1[i] + x2inputs[i]*w2[i])
    print(x1inputs[i]," ",x2inputs[i]," ", yin[i])
```

```
print("\nConsidering all weights as excitatory");
w1 = [1,1,1,1]
w2 = [-1,-1,-1,-1]
yin = []
print("x1","x2","yin")
for i in range(0,4):
    yin.append(x1inputs[i]*w1[i] + x2inputs[i]*w2[i])
    print(x1inputs[i]," ",x2inputs[i]," ", yin[i])
theta = 2*1-1
print("Threshold -Theta =",theta)
print("\nApplying Threshold ")
y = []
for i in range(0,4):
    if(yin[i]>=theta):
        value = 1
        y.append(value)
    else:
        value = 0
        y.append(value)
print("x1","x2"," y")
for i in range(0,4):
  print(x1inputs[i]," ",x2inputs[i]," ",y[i])
print('\nNishant-53004230004)
```

Output:

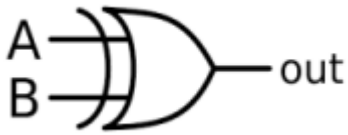
```
ANDNOT function using MP

Considering all weights as excitatory
x1 x2 yin
1   1   2
1   0   1
0   1   1
0   0   0

Considering all weights as excitatory
x1 x2 yin
1   1   0
1   0   1
0   1   -1
0   0   0
Threshold -Theta = 1

Applying Threshold
x1 x2  y
1   1   0
1   0   1
0   1   0
0   0   0

 Nishant - 53004230004
```

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

Theory:

The truth table for XOR function is computed as,

X_1	X_2	Y
0	0	0
0	1	1
1	0	1
1	1	0



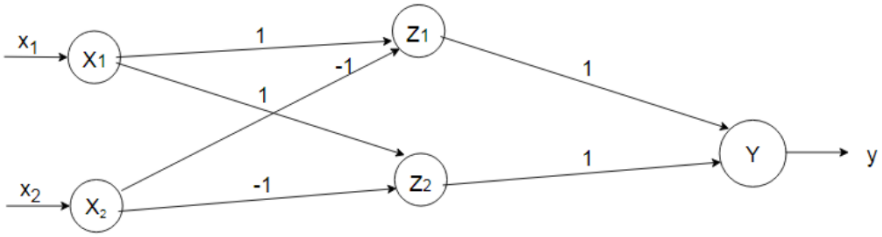
In this case, the output is "ON" only for odd number of 1's. For the rest it is "OFF". XOR function cannot be represented by simple and single logic function, it is represented as

$y = x_1\overline{x_2} + \overline{x_1}x_2$

$y = z_1 + z_2$

where $z_1 = x_1 . \overline{x_2}$ is the first functio,

and $z_2 = \overline{x_1} . x_2$ is the second function. $\Rightarrow y = z_1 + z_2$ is the third function



Code:

```
print("\nXOR function using McClloch-Pitts")
x1inputs = [1,1,0,0]
x2inputs = [1,0,1,0]
print("Calculating z1 = x1w11 + x2w12")
print("Considering one weight as exciatatory and other as inhibitory ")
w11 = [1,1,1,1]
w21 = [-1,-1,-1,-1]
print("x1","x2","z1")
z1 = []
for i in range(0,4):
    z1.append(x1inputs[i]*w11[i] + x2inputs[i]*w21[i])
    print(x1inputs[i]," ",x2inputs[i]," ",z1[i])
print("\nCalculating z1 = x1w21 + x2w22")
print("Considering one weight as exciatatory and other as inhibitory ")
w21 = [-1,-1,-1,-1]
w22 = [1,1,1,1]
```

```
print("x1","x2","z2")
z2 = []
for i in range(0,4):
    z2.append(x1inputs[i]*w21[i] + x2inputs[i]*w22[i])
    print(x1inputs[i]," ",x2inputs[i]," ",z2[i])
print("\nApplying Threshold = 1 for z1 and z2")
for i in range(0,4):
    if(z1[i]>=1):
        z1[i] = 1
    else:
        z1[i] = 0
    if(z2[i]>=1):
        z2[i]=1
    else:
        z2[i]=0
print("z1","z2")
for i in range(0,4):
    print(z1[i]," ",z2[i]," ")
print("x1" , "x2" , "yin")
yin = []
v1 = 1
v2 = 1
for i in range(0,4):
    yin.append(z1[i]*v1 + z2[i]*v2)
    print(x1inputs[i]," ",x2inputs[i]," ",yin[i])
y=[]
for i in range(0,4):
    if(yin[i]>=1):
        y.append(1)
    else:
        y.append(0)
print("x1","x2","y")
for i in range(0,4):
    print(x1inputs[i]," ",x2inputs[i]," ",y[i])
print('\nNishant-53004230004)
```

Output:

```
XOR function using McClloch-Pitts
Calculating z1 = x1w11 + x2w12
Considering one weight as exciatatory and other as inhibitory
x1 x2 z1
1   1   0
1   0   1
0   1   -1
0   0   0

Calculating z1 = x1w21 + x2w22
Considering one weight as exciatatory and other as inhibitory
x1 x2 z2
1   1   0
1   0   -1
0   1   1
0   0   0

Applying Threshold = 1 for z1 and z2
z1 z2
0   0
1   0
0   1
0   0
x1 x2 yin
1   1   0
1   0   1
0   1   1
0   0   0
x1 x2 y
1   1   0
1   0   1
0   1   1
0   0   0

Nishant - 53004230004
```

Practical No: 03

Implement the Following:

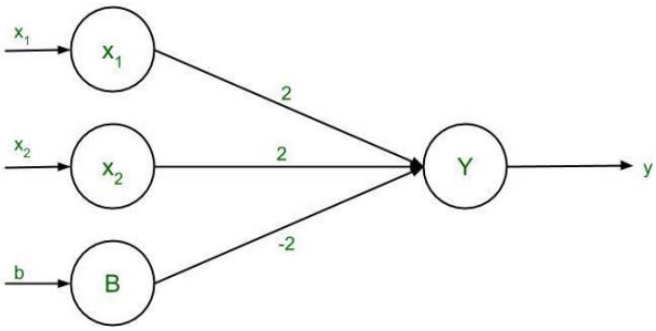
A. Write a program to implement Hebb's rule.

Theory:

Hebb's rule is a postulate proposed by Donald Hebb in 1949. It is a learning rule that describes how neuronal activities influence the connection between neurons, i.e., synaptic plasticity. It provides an algorithm to update the weight of neuronal connections within the neural networks. Hebb's rule provides a simplistic physiology-based model to mimic the activity-dependent features of synaptic plasticity and has been widely used in the area of artificial neural network.

Hebbian Learning Rule Algorithm:

1. Set all weights to zero, wi = 0 for i=1 to n, and bias to zero.
2. For each input vector, S(input vector) : t(target output pair), repeat steps 3-5.
3. Set activations for input units with the input vector Xi = Si for i = 1 to n.
4. Set the corresponding output value to the output neuron, i.e. y = t.
5. Update weight and bias by applying Hebb rule for all i = 1 to n:



Code:

```
import numpy as np
x1 = np.array([1,-1,-1,1,-1,-1,1,1,1,1])
x2 = np.array([1,-1,1,1,-1,1,1,1,1,1])
y = np.array([1,-1])
b = 0
wtold = np.zeros((9,)).astype(int)
wtnew = np.zeros((9,)).astype(int)
print("--",wtold)
print("First input with target 1")
for i in range(0,9):
    wtnew[i] = wtold[i] + x1[i]*y[0]
wtold = wtnew
b = b + y[0]
print("New Weights:",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 Weights:",wtnew)
print("Bias Value:",b)
print('\nNishant-53004230004)
```

Output:

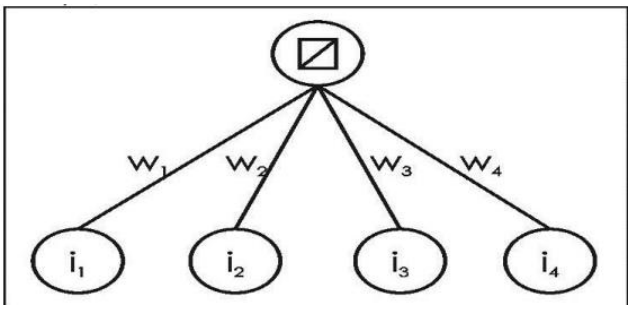
```
-- [0 0 0 0 0 0 0 0 0]
First input with target 1
New Weights: [ 1 -1 -1  1 -1 -1  1  1  1]
Bias Value: 1
Second input with target -1
New Weights: [ 0  0 -2  0  0 -2  0  0  0]
Bias Value: 0

Nishant - 53004230004
```

B. Write a program to implement delta rule.

Theory:

The Delta rule in machine learning and neural network environments is a specific type of backpropagation that helps to refine connectionist ML/AI networks, making connections between inputs and outputs with layers of artificial neurons.

The Delta rule is also known as the Delta learning rule. Backpropagation has to do with recalculating input weights for artificial neurons using a gradient method. Delta learning does this using the difference between a target activation and an actual obtained activation. Using a linear activation function, network connections are adjusted. Another way to explain the Delta rule is that it uses an error function to perform gradient descent learning.



Code:

```
import numpy as np
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("Initial Desired: "))
a = float(input("Enter learning rate: "))
print("\nActual",actual)
print("Desired\n",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('\nNishant-53004230004)
```

Output:

```
Initial Inputs: 1
Initial Inputs: 1
Initial Inputs: 1
Initial weights: 1
Initial weights: 1
Initial weights: 1
Initial Desired: 1
Initial Desired: 1
Initial Desired:.1
Enter learning rate: 1

Actual [0. 0. 0.]
Desired
 [1. 1. 1.]
Weights: [2. 1. 1.]
Weights: [2. 1. 1.]
Weights: [2. 1. 1.]
Weights: [1. 1. 1.]
Weights: [1. 1. 1.]
Weights: [1. 1. 1.]

Nishant - 53004230004
```

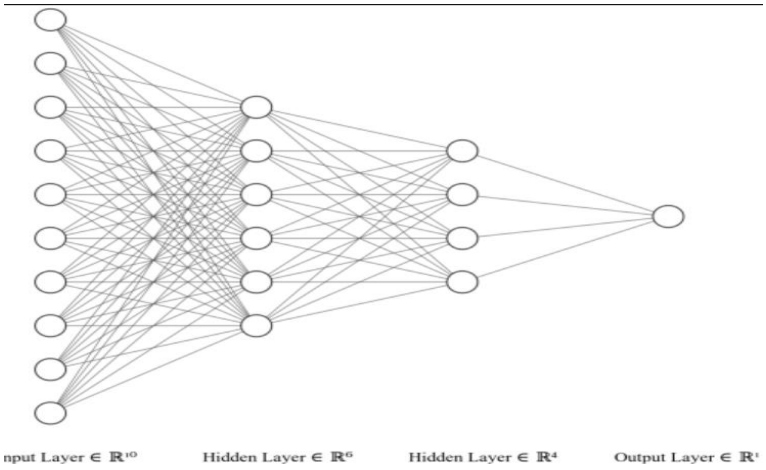
Practical No: 04

Implement the Following:

A. Write a program for Back Propagation Algorithm.

Theory:

Backpropagation is the essence of neural network training. It is the method of fine-tuning the weights of a neural network based on the error rate obtained in the previous epoch (i.e., iteration). Proper tuning of the weights allows you to reduce error rates and make the model reliable by increasing its generalization. Backpropagation in neural network is a short form for “backward propagation of errors.” It is a standard method of training artificial neural networks. This method helps calculate the gradient of a loss function with respect to all the weights in the network.



Code:

```
import numpy as np
X=np.array(([2,9],[1,5],[3,6]),dtype=float)
Y=np.array(([92],[86],[89]),dtype=float)
#scale units
X=X/np.amax(X,axis=0)
Y=Y/100;
class NN(object):
    def __init__(self):
        self.inputsize=2
        self.outputsize=1
        self.hiddensize=3
        self.W1=np.random.randn(self.inputsize,self.hiddensize)
        self.W2=np.random.randn(self.hiddensize,self.outputsize)
    def forward(self,X):
        self.z=np.dot(X,self.W1)
        self.z2=self.sigmoidal(self.z)
```

```
        self.z3=np.dot(self.z2,self.W2)
        op=self.sigmoidal(self.z3)
        return op;
    def sigmoidal(self,s):
        return 1/(1+np.exp(-s))
obj=NN()
op=obj.forward(X)
print("actual output\n"+str(op))
print("expected output\n"+str(Y))
print('\nNishant-53004230004)
```

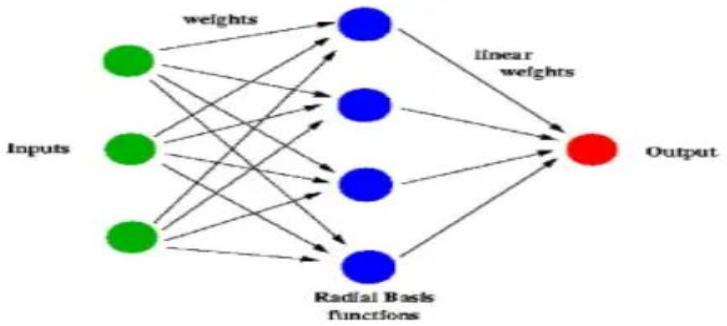
Output:

```
/SC Pracs/4A.py"
actual output
[[0.08680804]
 [0.13339794]
 [0.1063096 ]]
expected output
[[0.92]
 [0.86]
 [0.89]]
Nishant - 53004230004
```

B. Write a program for Error Back Propagation Algorithm.

Theory:

Error backpropagation. For hidden units, we must propagate the error back from the output nodes (hence the name of the algorithm). Again, using the chain rule, we can expand the error of a hidden unit in terms of its posterior nodes:

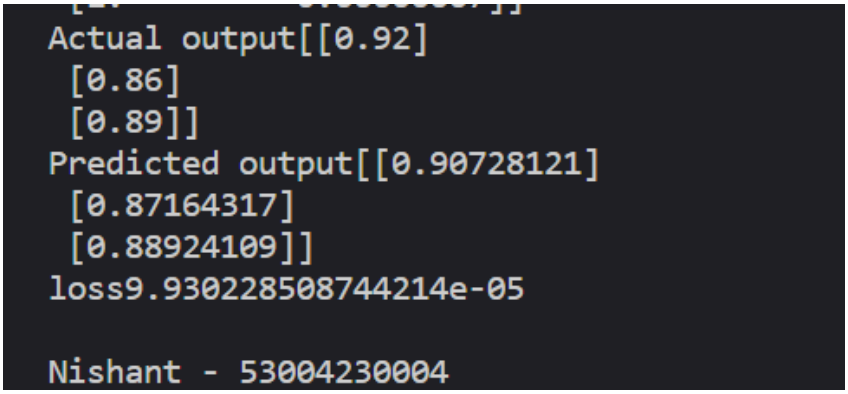


Code:

```
import numpy as np
```

```
X=np.array(([2,9],[1,5],[3,6]),dtype=float)
Y=np.array(([92],[86],[89]),dtype=float)
X=X/np.amax(X,axis=0)
Y=Y/100;
class NN(object):
    def __init__(self):
        self.inputsize=2
        self.outputsize=1
        self.hiddensize=3
        self.W1=np.random.randn(self.inputsize,self.hiddensize)
        self.W2=np.random.randn(self.hiddensize,self.outputsize)
        return s* (1-s)
    def backward(self,X,Y,o):
        self.o_error=Y-o
        self.o_delta=self.o_error * self.sigmoidalprime(o)
        self.z2_error=self.o_delta.dot(self.W2.T)
        self.z2_delta=self.z2_error * self.sigmoidalprime(self.z2)
        self.W1 = self.W1 + X.T.dot(self.z2_delta)
        self.W2= self.W2+ self.z2.T.dot(self.o_delta)
    def train(self,X,Y):
        o=self.forward(X)
        self.backward(X,Y,o)
obj=NN()
for i in range(2000):
    print("input"+str(X))
    print("Actual output"+str(Y))
    print("Predicted output"+str(obj.forward(X)))
    print("loss"+str(np.mean(np.square(Y-obj.forward(X)))))
    obj.train(X,Y)
print('\nNishant-53004230004)
```

Output:



Practical No: 05

Implement the Following:

A. Write a program for Hopfield Network.

Theory:

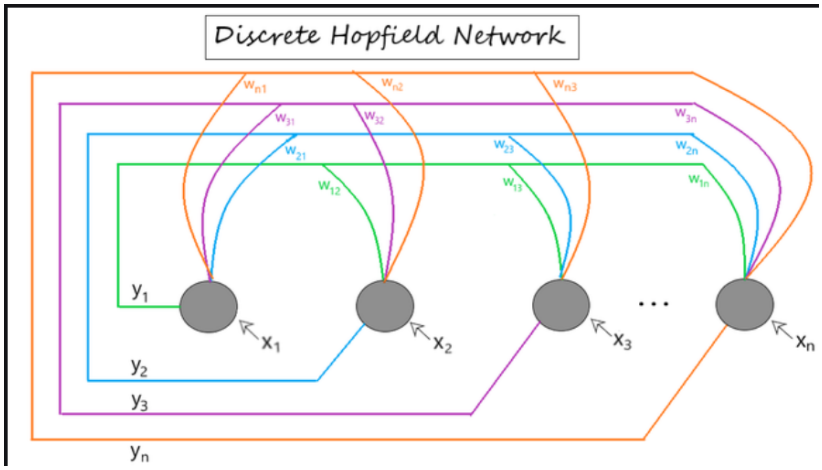
• A Hopfield network is a single-layered and recurrent network in which the neurons are entirely connected, i.e., each neuron is associated with other neurons. If there are two neurons i and j, then there is a connectivity weight wij lies between them which is symmetric wij = wji .

• A Hopfield network is at first prepared to store various patterns or memories. Afterward, it is ready to recognize any of the learned patterns by uncovering partial or even some corrupted data about that pattern, i.e., it eventually settles down and restores the closest pattern. Thus, similar to the human brain, the Hopfield model has stability in pattern recognition.

• There are two different approaches to update the nodes:

1] Synchronously: In this approach, the update of all the nodes taking place simultaneously at each time.

2] Asynchronously: In this approach, at each point of time, update one node chosen randomly or according to some rule. Asynchronous updating is more biologically realistic.

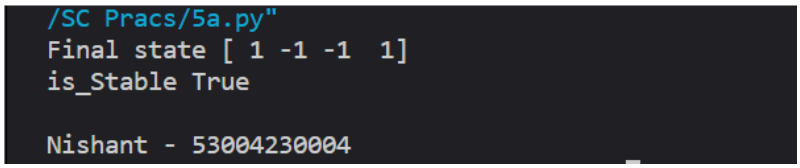


Code:

```
import numpy as np
def compute_next_state(state,weight):
    next_state = np.where(weight @ state>= 0, +1, -1)
    return next_state
def compute_final_state(initial_state,weight,max_iter=1000):
    previous_state = initial_state
    next_state =compute_next_state(previous_state,weight)
    is_stable = np.all(previous_state == next_state)
    n_iter = 0
    while(not is_stable) and (n_iter <= max_iter):
```

```
        previous_state = next_state;
        next_state = compute_next_state(previous_state,weight)
        is_stable = np.all(previous_state==next_state)
        n_iter +=1
    return previous_state, is_stable,n_iter
initial_state = np.array([+1,-1,-1,-1])
weight = np.array([
        [0, -1, -1, +1],
        [-1, 0, +1, -1],
        [-1,+1,  0, -1],
        [+1,-1, -1,  0]])
final_state, is_stable, n_iter = compute_final_state(initial_state,weight)
print("Final state",final_state)
print("is_Stable",is_stable)
print('\nNishant-53004230004)
```

Output:

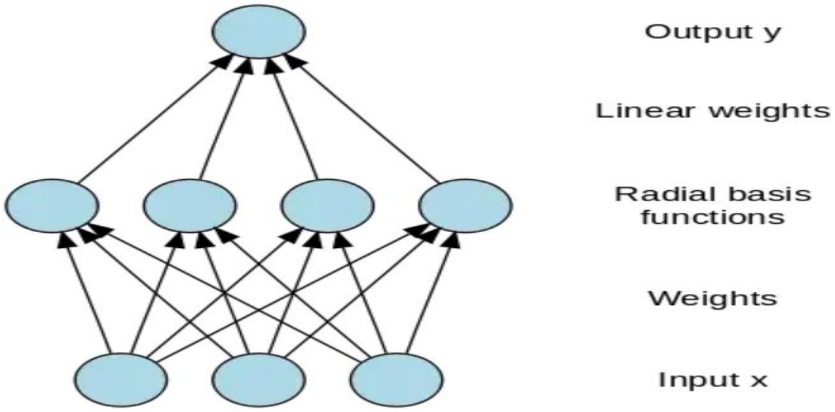


B. Write a program for Radial Basis function.

Theory:

• Radial Basis Function (RBF) Networks are a particular type of Artificial Neural Network used for function approximation problems. RBF Networks differ from other neural networks in their three-layer architecture, universal approximation, and faster learning speed.

•Radial Basis Functions are a special class of feed-forward neural networks consisting of three layers: an input layer, a hidden layer, and the output layer. The input layer receives input data and passes it into the hidden layer, where the computation occurs. The hidden layer of Radial Basis Functions Neural Network is the most powerful and very different from most Neural networks. The output layer is designated for prediction tasks like classification.

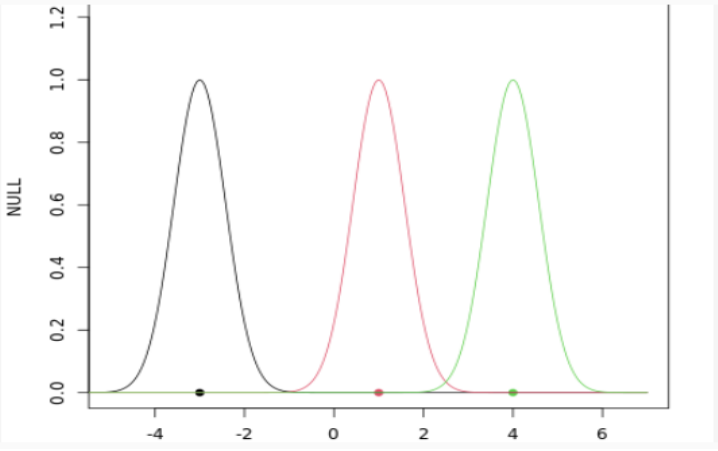


Code: R Compiler

```
D <- matrix(c(-3,1,4), ncol=1)
N <- length(D)
rbf.gauss <- function(gamma=1.0) {
   function(x){
      exp(-gamma * norm(as.matrix(x),"F")^2)
   }
}
xlim <- c(-5,7)
print(N)
print(xlim)
plot(NULL,xlim=xlim,ylim=c(0,1.25), type = "n")
points(D,rep(0,length(D)), col= 1:N,pch=19)
x.coord = seq(-7,7,length=250)
gamma <- 1.5
for (i in 1:N){
   points(x.coord, lapply(x.coord - D[i,],rbf.gauss(gamma)),type="l",col=i)
}
```

Output:

```
[1] 3
[1] -5  7

[Execution complete with exit code 0]
```



Practical No: 06

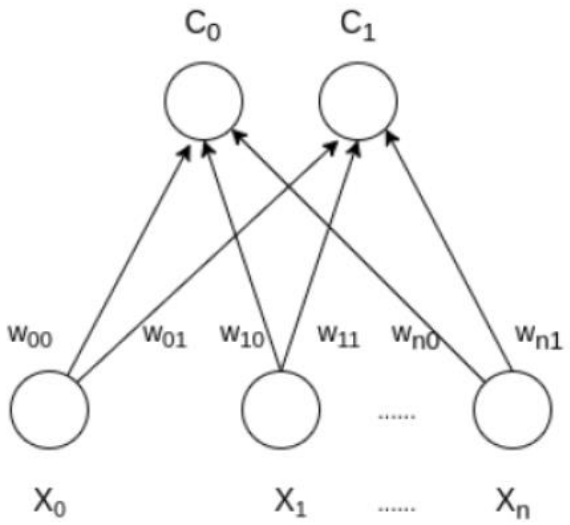
Implement the Following:

A. Kohonen Self organizing map.

Theory:

Self-Organizing Map (or Kohonen Map or SOM) is a type of Artificial Neural Network which is also inspired by biological models of neural systems from the 1970s. It follows an unsupervised learning approach and trained its network through a competitive learning algorithm. SOM is used for clustering and mapping (or dimensionality reduction) techniques to map multidimensional data onto lower-dimensional which allows people to reduce complex problems for easy interpretation. SOM has two layers; one is the Input layer and the other one is the Output layer.

The architecture of the Self Organizing Map with two clusters and n input features of any sample is given below:



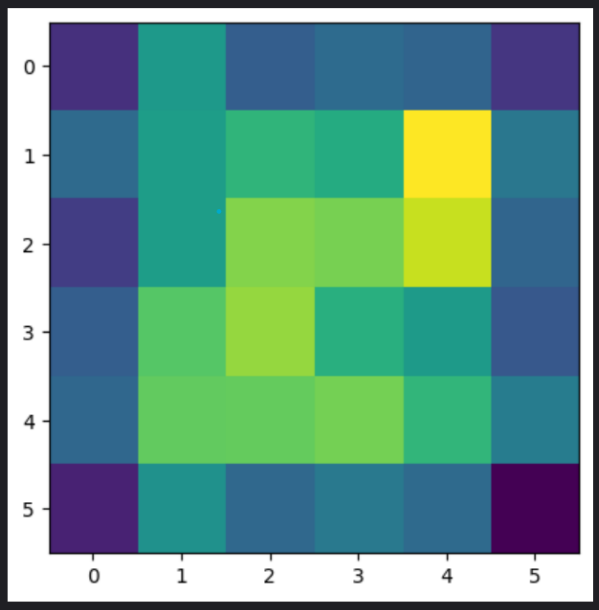
Code:

```
!pip install minisom
import minisom
from minisom import MiniSom
import matplotlib.pyplot as plt
data = [[ 0.80, 0.55, 0.22, 0.03],
 [ 0.82, 0.50, 0.23, 0.03],
 [ 0.80, 0.54, 0.22, 0.03],
 [ 0.80, 0.53, 0.26, 0.03],
 [ 0.79, 0.56, 0.22, 0.03],
 [ 0.75, 0.60, 0.25, 0.03],
 [ 0.77, 0.59, 0.22, 0.03]]
som = MiniSom(6, 6, 4, sigma=0.3, learning_rate=0.5) # initialization of 6x6 SOM
som.train_random(data, 100) # trains the SOM with 100 iterations
plt.imshow(som.distance_map())
```

```
print('\nNishant-53004230004)
```

Output:

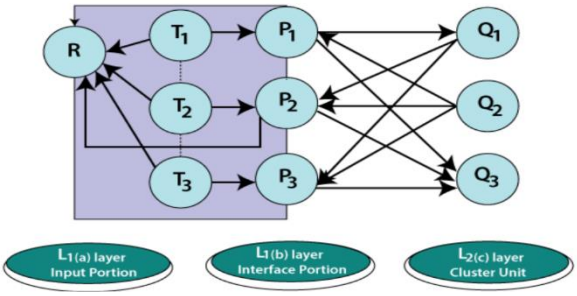
```
Nishant - 53004230004
...
```

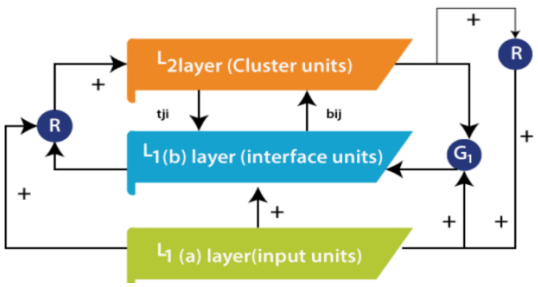


B. Adaptive Resonanace Theory.

Theory:

Adaptive resonance theory is a type of neural network technique. The basic ART uses unsupervised learning technique. The term “adaptive” and “resonance” used in this suggests that they are open to new learning (i.e. adaptive) without discarding the previous or the old information(i.e. resonance). The ART networks are known to solve the stability-plasticity dilemma i.e., stability refers to their nature of memorizing the learning and plasticity refers to the fact that they are flexible to gain new information. Due to this the nature of ART they are always able to learn new input patterns without forgetting the past. ART networks implement a clustering algorithm.



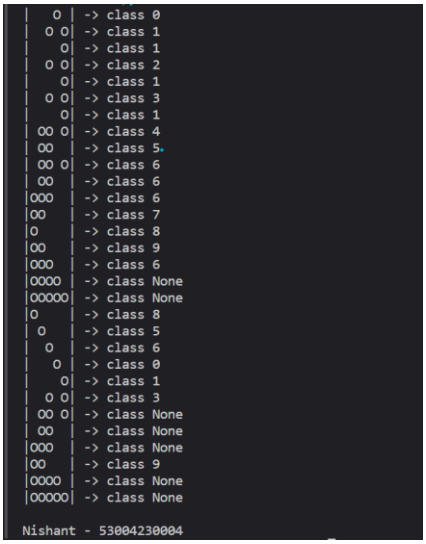


Code:

```
from __future__ import print_function
from __future__ import division
import numpy as np
class ART:
    def __init__(self, n=5, m=10, rho=.5):
        self.F1 = np.ones(n)
        self.F2 = np.ones(m)
        self.Wf = np.random.random((m,n))
        self.Wb = np.random.random((n,m))
        self.rho = rho
        self.active = 0
    def learn(self, X):
        self.F2[...] = np.dot(self.Wf, X)
        I = np.argsort(self.F2[:self.active].ravel())[::-1]
        for i in I:
            d = (self.Wb[:,i]*X).sum()/X.sum()
            if d >= self.rho:
                self.Wb[:,i] *= X
                self.Wf[i,:] = self.Wb[:,i]/(0.5+self.Wb[:,i].sum())
                return self.Wb[:,i], i
        if self.active < self.F2.size:
            i = self.active
            self.Wb[:,i] *= X
            self.Wf[i,:] = self.Wb[:,i]/(0.5+self.Wb[:,i].sum())
            self.active += 1
            return self.Wb[:,i], i
        return None,None
if __name__ == '__main__':
    np.random.seed(1)
    network = ART( 5, 10, rho=0.5)
    data = ["   O ",
            "  O O",
            "    O",
            "  O O",
            "    O",
            "  O O",
            "    O",
            " OO O",
            " OO  ",
            " OO O",
            " OO  ",
            "OOO  ",
            "OO   ",
```

```
            "O    ",
            "OO   ",
            "OOO  ",
            "OOOO ",
            "OOOOO",
            "O    ",
            " O   ",
            "  O  ",
            "   O ",
            "    O",
            "  O O",
            " OO O",
            " OO  ",
            "OOO  ",
            "OO   ",
            "OOOO ",
            "OOOOO"]
    X = np.zeros(len(data[0]))
    for i in range(len(data)):
        for j in range(len(data[i])):
            X[j] = (data[i][j] == 'O')
        Z, k = network.learn(X)
        print("|%s|"%data[i],"-> class", k)
print('\nNishant-53004230004)
```

Output:



Practical No: 07

Implement the Following:

A. Write a program for Linear Separation.

Theory:

Linear separability is the concept wherein the separation of input space into regions is based on whether the network response is positive or negative.

A decision line is drawn to separate positive and negative responses. The decision line may also be called as the decision-making Line or decision-support Line or linear-separable line. The necessity of the linear separability concept was felt to clarify classify the patterns based upon their output responses.

Generally, the net input calculated to the output unit is given as -

$$y_{in} = b + \sum_{i=1}^{n}(x_i w_i)$$

⇒ The linear separability of the network is based on the decision-boundary line. If there exist weight for which the training input vectors having a positive (correct) response, or lie on one side of the decision boundary and all the other vectors having negative, −1, response lies on the other side of the decision boundary then we can conclude the problem is "Linearly Separable".

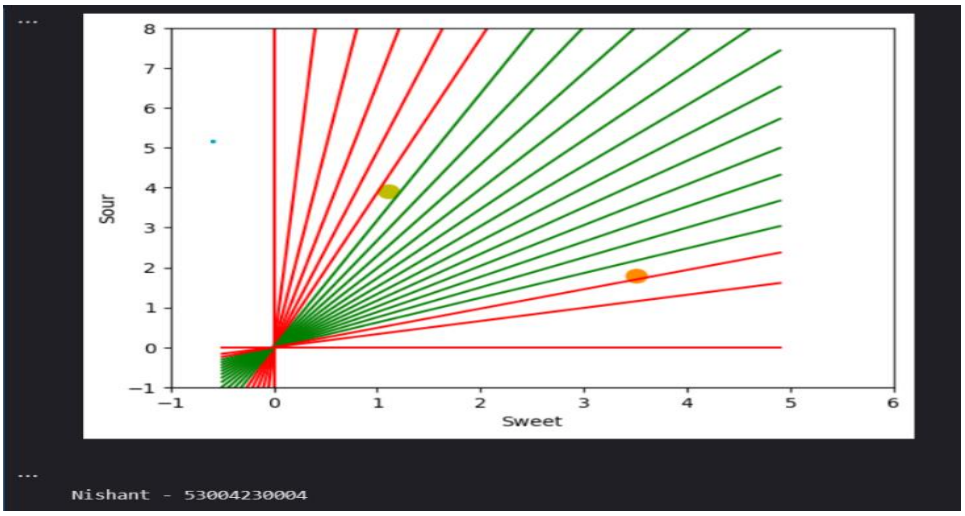
Code:

```
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("Sweet")
ax.set_ylabel("Sour")
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()
```

```
print('\nNishant-53004230004)
```

Output:



B. Write a program for Hopfield Network model for associative memory.

Theory:

Hopfield networks are a special kind of recurrent neural networks that can be used as associative memory. Associative memory is memory that is addressed through its contents. That is, if a pattern is presented to an associative memory, it returns whether this pattern coincides with a stored pattern. An associative memory may also return a stored pattern that is similar to the presented one, so that noisy input can also be recognized.

Hopfield networks are used as associative memory by exploiting the property that they possess stable states, one of which is reached by carrying out the normal computations of a Hopfield network. If the connection weights of the network are determined in such a way that the patterns to be stored become the stable states of the network, a Hopfield network produces for any input pattern a similar stored pattern. Thus, noisy patterns can be corrected or distorted patterns can still be recognized.

The dynamics is that of Equation:

$$S_i(t+1) = sgn\left(\sum_j w_{ij} S_j(t)\right)$$

In the Hopfield model each neuron is connected to every other neuron. The connection matrix is:

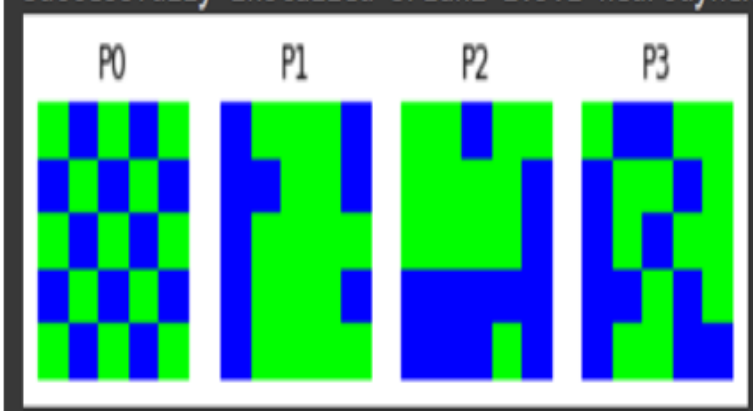
$$w_{ij} = \frac{1}{N} \sum_{\mu} p_i^{\mu} p_j^{\mu}$$

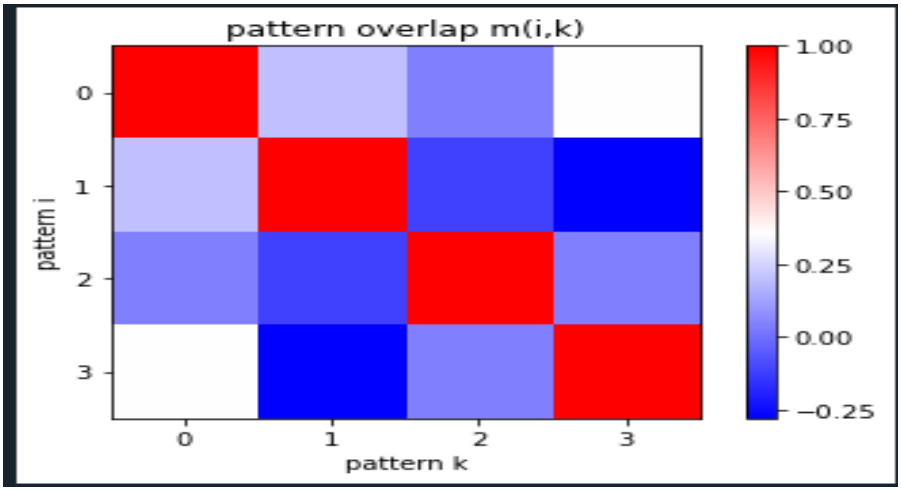
Code:

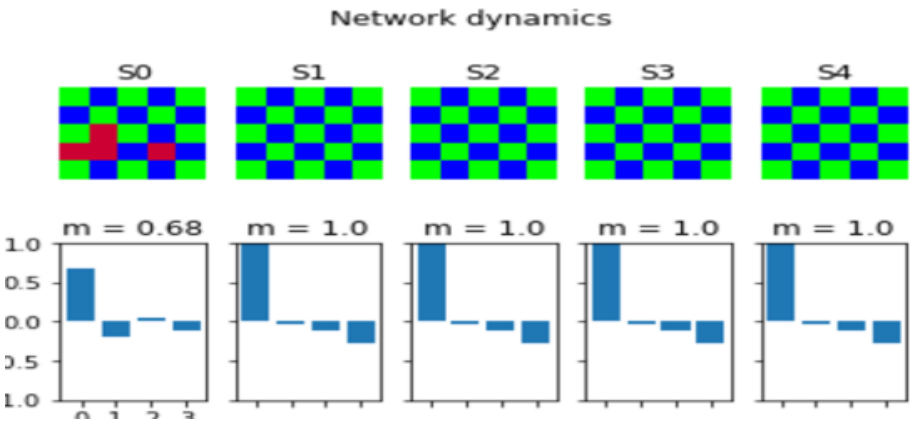
```
!pip install neurodynex
from neurodynex.hopfield_network import network, pattern_tools, plot_tools
import matplotlib.pyplot as plt
pattern_size = 5
# create an instance of the class HopfieldNetwork
hopfield_net = network.HopfieldNetwork(nr_neurons= pattern_size**2)
# instantiate a pattern factory
factory = pattern_tools.PatternFactory(pattern_size, pattern_size)
# create a checkerboard pattern and add it to the pattern list
checkerboard = factory.create_checkerboard()
pattern_list = [checkerboard]
# add random patterns to the list
pattern_list.extend(factory.create_random_pattern_list(nr_patterns=3, on_probability=0.5))
plot_tools.plot_pattern_list(pattern_list)
# how similar are the random patterns and the checkerboard? Check the overlaps
overlap_matrix = pattern_tools.compute_overlap_matrix(pattern_list)
plot_tools.plot_overlap_matrix(overlap_matrix)
# let the hopfield network "learn" the patterns. Note: they are not stored
# explicitly but only network weights are updated !
hopfield_net.store_patterns(pattern_list)
# create a noisy version of a pattern and use that to initialize the network
noisy_init_state = pattern_tools.flip_n(checkerboard, nr_of_flips=4)
hopfield_net.set_state_from_pattern(noisy_init_state)
# from this initial state, let the network dynamics evolve.
states = hopfield_net.run_with_monitoring(nr_steps=4)
# each network state is a vector. reshape it to the same shape used to create the patterns.
states_as_patterns = factory.reshape_patterns(states)
# plot the states of the network
plot_tools.plot_state_sequence_and_overlap(states_as_patterns, pattern_list,
reference_idx=0, suptitle="Network dynamics")
```

Output:

```
Installing collected packages: brian2, neurodynex
Successfully installed brian2-2.5.1 neurodynex-0.3.4
```







Practical No: 08

Implement the Following:

A. Membership and Identity Operators | in, not in.

Theory:

Python offers two membership operators to check or validate the membership of a value. It tests for membership in a sequence, such as strings, lists, or tuples.

in operator: The 'in' operator is used to check if a character/ substring/ element exists in a sequence or not. Evaluate to True if it finds the specified element in a sequence otherwise False.

*'not in' operato*r- Evaluates to true if it does not finds a variable in the specified sequence and false otherwise.

Identity operators:

Identity operators are used to compare the objects if both the objects are actually of the same data type and share the same memory location.

There are different identity operators such as:

'is' operator – Evaluates to True if the variables on either side of the operator point to the same object and false otherwise.

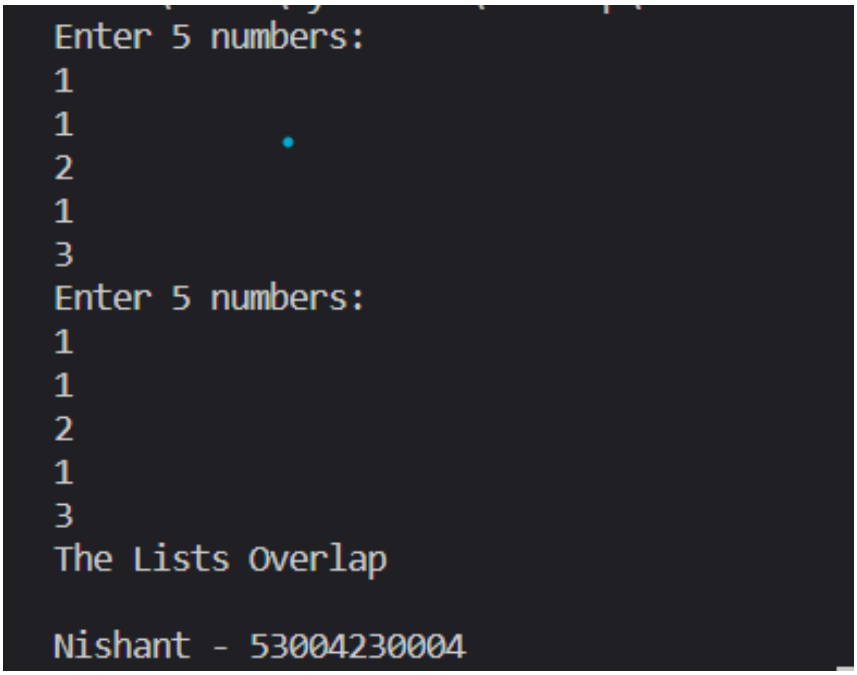
'is not' operator: Evaluates True if both variables are not the same object.

1- In Operator

Code:

```
list1=[]
print("Enter 5 numbers: ")
for i in range(0,5):
    v=int(input())
    list1.append(v)
list2=[]
print("Enter 5 numbers: ")
for i in range(0,5):
    v=int(input())
    list2.append(v)
flag=0
for i in list1:
    if i in list2:
        flag=1
if(flag==1):
    print("The Lists Overlap")
else:
    print("The Lists do Not overlap")
print('\nNishant-53004230004)
```

Output:

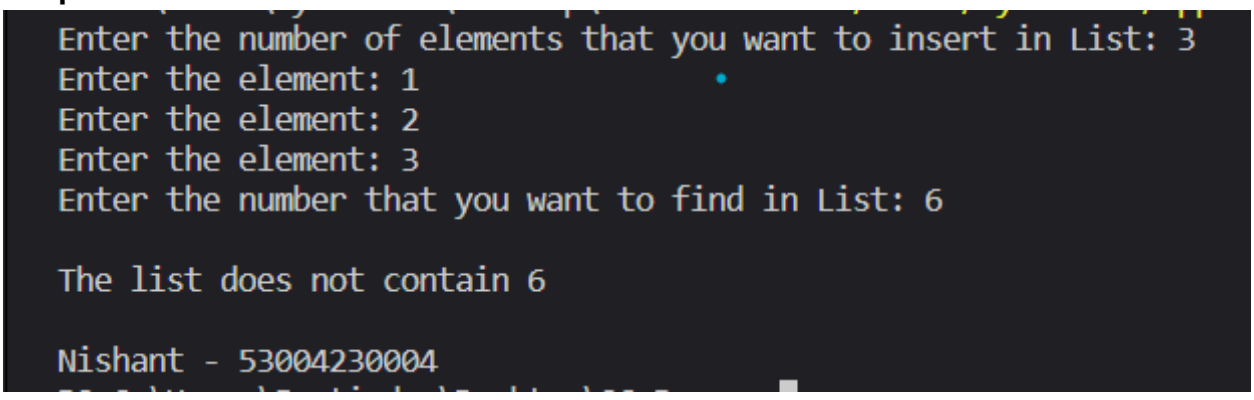


2- not In Operator

Code:

```
#Aim: Not in operator in python
list1=[]
c=int(input("Enter the number of elements that you want to insert in List: "))
for i  in range(0,c):
   ele = int(input("Enter the element: "))
   list1.append(ele)
a = int(input("Enter the number that you want to find in List: "))
if a not in list1:
    print("\nThe list does not contain", a )
else:
    print("\nThe list contains", a )
print('\nNishant-53004230004)
```

Output:



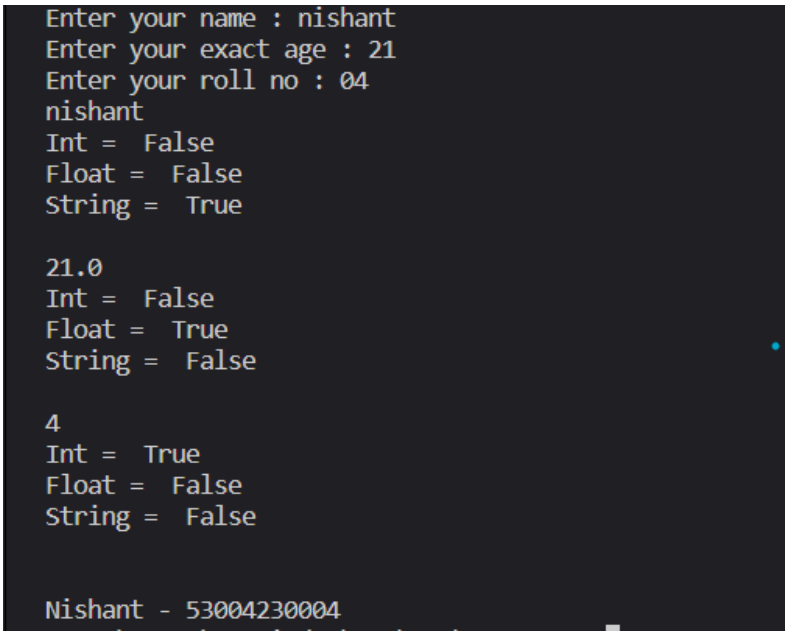
B. Membership and Identity Operators | is, is not.

1-Implement Membership and Identity Operators is.

Code:

```
details =[]
name=input("Enter your name : ")
details.append(name)
age=float(input("Enter your exact age : "))
details.append(age)
roll_no=int(input("Enter your roll no : "))
details.append(roll_no)
for i in details:
    print(i)
    print("Int = ",type(i) is int)
    print("Float = ",type(i) is float)
    print("String = ",type(i) is str)
    print()
print('\nNishant-53004230004)
```

Output:

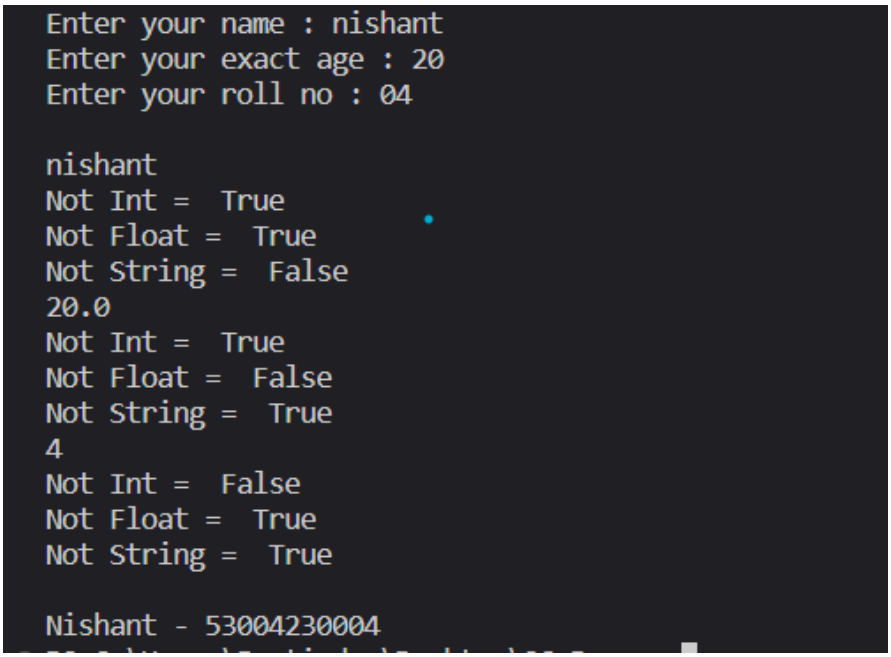


2-Implement Membership and Identity Operators is not.

Code:

```
details =[]
name=input("Enter your name : ")
details.append(name)
age=float(input("Enter your exact age : "))
details.append(age)
roll_no=int(input("Enter your roll no : "))
details.append(roll_no)
print()
for i in details:
    print(i)
    print("Not Int = ",type(i) is not int)
    print("Not Float = ",type(i) is not float)
    print("Not String = ",type(i) is not str)
print('\nNishant-53004230004)
```

Output:



Practical No: 09

Implement the Following:

A. Find ratios using Fuzzy logic.

Theory:

FuzzyWuzzy is a library of Python which is used for string matching. Fuzzy string matching is the process of finding strings that match a given pattern. Basically, it uses Levenshtein Distance to calculate the differences between sequences.

FuzzyWuzzy has been developed and open-sourced by SeatGeek.

The FuzzyWuzzy library is built on top of difflib library, python-Levenshtein is used for speed. So, it is one of the best way for string matching in python.

Code:

```
!pip install fuzzywuzzy
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 ("FuzzyWuzzy PartialRatio: ", fuzz.partial_ratio(s1, s2))
print ("FuzzyWuzzy TokenSortRatio: ", fuzz.token_sort_ratio(s1, s2))
print ("FuzzyWuzzy TokenSetRatio: ", fuzz.token_set_ratio(s1, s2))
print ("FuzzyWuzzy WRatio: ", 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))
print('Nishant-53004230004)
```

Output:

```
FuzzyWuzzy Ratio: 86
FuzzyWuzzy PartialRatio:  86
FuzzyWuzzy TokenSortRatio:  86
FuzzyWuzzy TokenSetRatio:  87
FuzzyWuzzy WRatio:  86


List of ratios:
[('g. for fuzzys', 95), ('fuzzy for fuzzy', 94), ('fuzzy fuzzy', 86)]

Best among the above list:  ('g. for fuzzys', 95)
Nishant - 53004230004
```

B. Solve Tipping problem using Fuzzy logic.

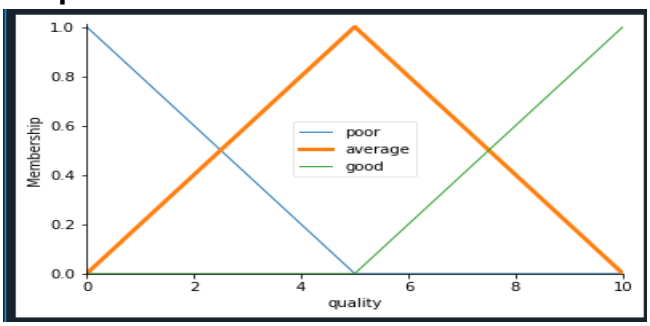
Theory:

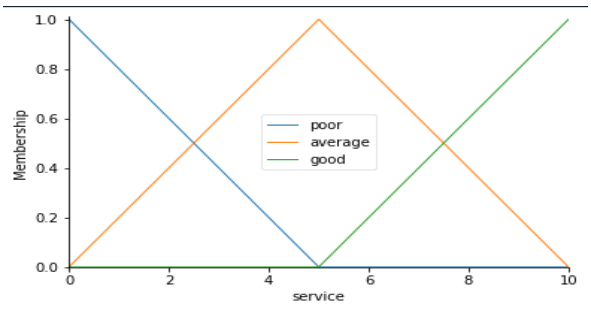
The ‘tipping problem’ is commonly used to illustrate the power of fuzzy logic principles to generate complex behavior from a compact, intuitive set of expert rules.

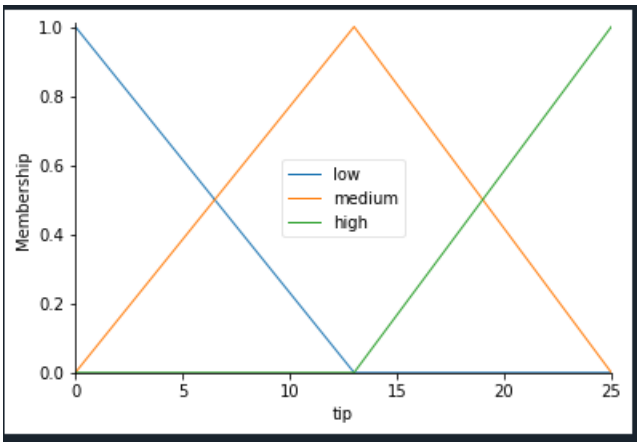
Code:

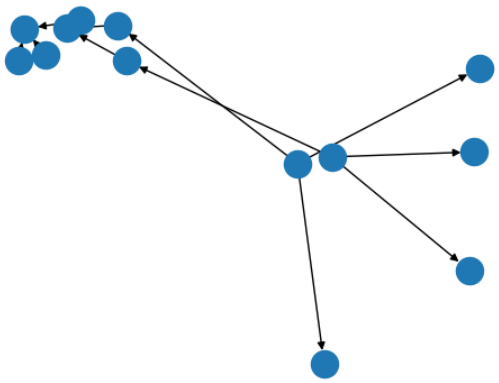
```
!pip install fuzzywuzzy
!pip install -U scikit-fuzzy
import skfuzzy as fuzz
from skfuzzy import control as ctrl
import numpy as np
quality = ctrl.Antecedent(np.arange(0, 11, 1), 'quality')
service = ctrl.Antecedent(np.arange(0, 11, 1), 'service')
tip = ctrl.Consequent(np.arange(0, 26, 1), 'tip')
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(quality['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.view(sim=tipping)
print('Nishant-53004230004)
```

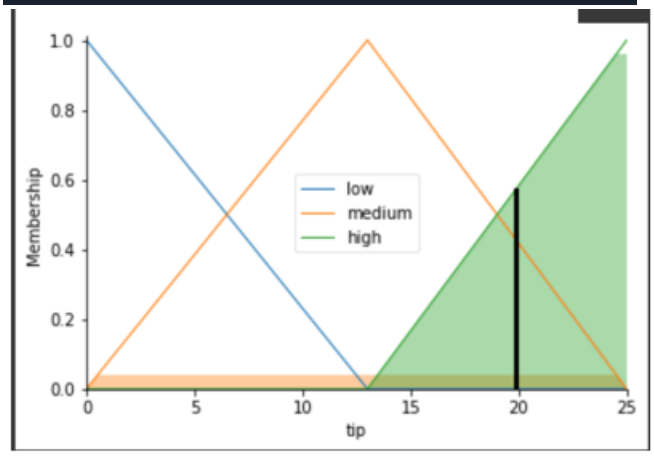
Output:






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



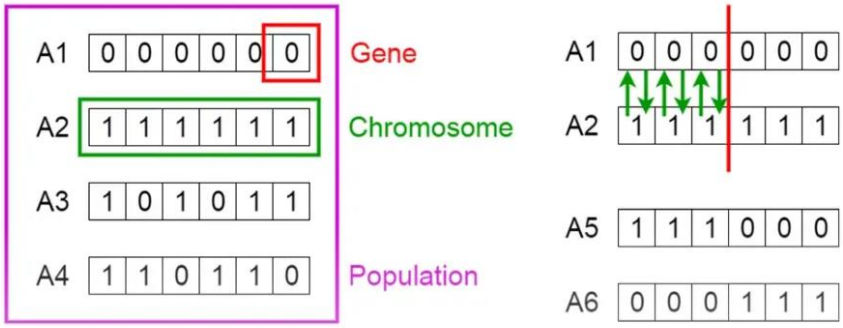

1.0
0.8
0.6
0.4
0.2
0.0
Membership
low
medium
high
0
5
10
15
20
25
tip

Practical No: 10

Theory:

The genetic algorithm is a method for solving both constrained and unconstrained optimization problems that is based on natural selection, the process that drives biological evolution. The genetic algorithm repeatedly modifies a population of individual solutions. At each step, the genetic algorithm selects individuals from the current population to be parents and uses them to produce the children for the next generation.

Over successive generations, the population "evolves" toward an optimal solution. You can apply the genetic algorithm to solve a variety of optimization problems that are not well suited for standard optimization algorithms, including problems in which the objective function is discontinuous, nondifferentiable, stochastic, or highly nonlinear.



Implement the Following:

A. Implementation of a Simple Genetic Algorithm.

Code:

```
import random
# Number of individuals in each generation
POPULATION_SIZE = 100
# Valid genes
GENES = '''abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOP
QRSTUVWXYZ 1234567890, .-;:_!"#%&/()=?@${[]}'''
# Target string to be generated
TARGET = "UPG College Student "
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()
print('Nishant-53004230004)
```

Output:

B. Create two classes: City and Fitness using Genetic Algorithm.

Code:

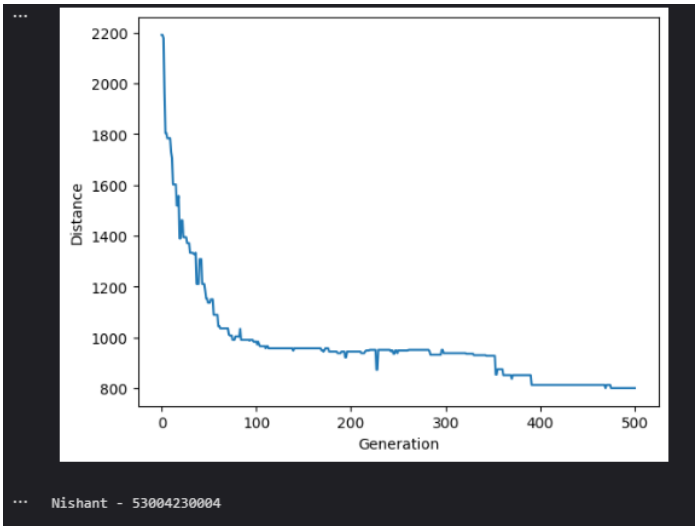
```
import numpy as np, random, operator, pandas as pd, matplotlib.pyplot as plt
class City:
    def __init__(self, x, y):
        self.x = x
        self.y = y
    def distance(self, city):
        xDis = abs(self.x - city.x)
        yDis = abs(self.y - city.y)
        distance = np.sqrt((xDis ** 2) + (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
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)
main()
print('Nishant-53004230004)
```

Output:



Nishant - 53004230004