SOFT COMPUTING TECHNIQUES

PRACTICAL JOURNAL

Submitted in partial fulfillment of the Requirements for the award of the Degree of

MASTER OF SCIENCE (INFORMATION TECHNOLOGY)

By

Mr. Rishabh Anil Patel

SEAT NO.:

DEPARTMENT OF INFORMATION TECHNOLOGY & COMPUTER SCIENCE

D. G. RUPAREL COLLEGE OF ARTS, SCIENCE & COMMERCE

(Affiliated to University of Mumbai)

SENAPATI BAPAT MARG, MAHIM, MUMBAI, 400 016 MAHARASHTRA

2025-2026

M O D E R N E D U C A T I O N S O C I E T Y P U N E

The D. G. Ruparel College

OF ARTS, SCIENCE AND COMMERCE

senapati bapat marg, opp. matunga road station (w. r.), mahim, mumbai 400 016. tel.: 022-24303733, 022-24303081, 022-24361139 • fax : 022-24303042

Department of Information Technology and Computer Science

Certificate

This is to certify that Mr. Rishabh Anil Patel of M.Sc. Information Technology class, Seat No. __ Semester - I has successfully completed the required number of practical in the subject of Soft Computing Techniques for the Academic Year 2025-2026.

Teacher In-charge
Date:-

Examiner
Date:-

Course Coordinator

INDEX

PRACTICAL: 1 A

Aim: Design a simple Linear Neural Network Model.

Introduction:

A Linear Neural Network is the simplest form of neural network that builds a straight-line relationship between input and output. It has an input layer and an output layer without any non-linear activation function.

This model helps in understanding how weights and biases affect the final output and how learning happens through error minimization.

It is commonly used for linear regression, data fitting, and as a foundation for advanced neural network models.

Basic Concept:

The linear neuron produces output as:

$Y = W_1X_1 + W_2X_2 + ... + WnXn + B$

Where,

X = Inputs

W = Weights

B = Bias

Y = Output

Training aims to adjust weights and bias to minimize prediction error.

Keywords:

1. **Neuron –** The processing unit that receives inputs and produces an output.
2. **Weights & Bias –** Parameters controlling the influence of each input.
3. **Activation Function** – Function that transforms neuron output (not used in linear models).
4. **Error Function –** Measures difference between predicted and actual outputs.

Package Used:

NumPy – Used for numerical operations like matrix multiplication and array handling, essential for neural network computation.

Working Principle:

1. Multiply each input by its respective weight.

2. Add the bias term to the weighted sum.

3. The result gives the linear output.

4. Compare predicted and actual outputs to calculate error and adjust weights.

Conclusion:

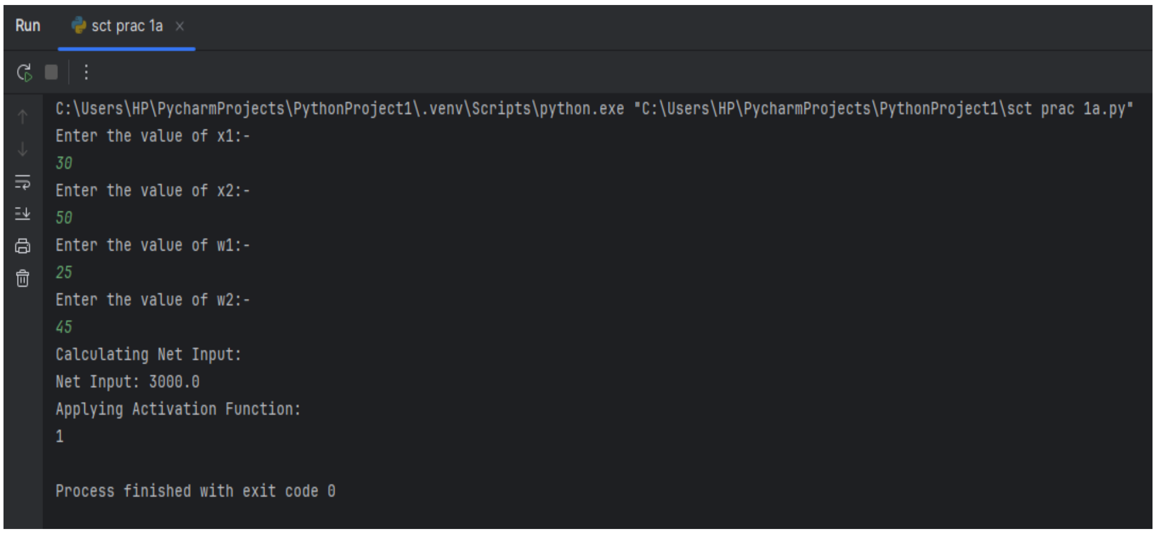
A Linear Neural Network provides a clear understanding of how neural networks learn from data. It explains the roles of weights, bias, and error correction.

Although it works only for linear relationships, it forms the foundation for complex neural network and deep learning architectures.

Code:

```
print ("Enter the value of x2:- ")
x2=float(input())
print("Enter the value of w1:-")
w1=float(input())
print("Enter the value of w2:-")
w2=float(input())
print("Calculating Net Input:")
net_input=x1*w1+x2*w2
print("Net Input:",net_input)
print("Applying Activation Function:")
threshold=0
if net_input>=threshold:
    print("1")
else:
    print("0")
```

Output:



```
C:\Users\HP\PycharmProjects\PythonProject1\.venv\Scripts\python.exe "C:\Users\HP\PycharmProjects\PythonProject1\sct prac 1a.py"
Enter the value of x1:-
30
Enter the value of x2:-
50
Enter the value of w1:-
25
Enter the value of w2:-
45
Calculating Net Input:
Net Input: 3000.0
Applying Activation Function:
1

Process finished with exit code 0
```

PRACTICAL: 1 B

Aim: CALCULATE THE OUTPUT OF NEURAL NET USING BOTH BINARY AND BIPOLAR SIGMOIDAL FUNCTION.

Introduction:

A **Neural Network** is a computational system inspired by the human brain. It processes data through interconnected layers of neurons**.**

Each neuron performs two operations:

1. Computes the **weighted sum** of inputs and adds a bias.

2. Passes the result through an **activation function** to produce output.

The **Sigmoidal function** introduces non-linearity, helping the model learn complex relationships between inputs and outputs**.**

Types of Sigmoidal Functions:

1. Binary Sigmoidal Function:

Output range**: 0 to 1**

• **Formula:** [

f(x) = \frac{1}{1 + e^{-x}}

]

• Commonly used in **probability-based** models.

2. Bipolar Sigmoidal Function:

• Output range: **–1 to +1**

• **Formula:**

[f(x) = \frac{1 - e^{-x}}{1 + e^{-x}}

]

• Mostly used in **classification and pattern recognition** tasks.

Keywords:

Neuron, Weights, Bias, Activation Function, Sigmoid Function, Neural Network

Package Used:

NumPy – a Python library used for **numerical and mathematical** operations, such as computing exponential functions and matrix operations in neural networks.

Example Calculation:

For **x = 1,**

• **Binary Sigmoid Output:** 0.731

• **Bipolar Sigmoid Output:** 0.462

Thus, for the same input, both give outputs in **different ranges**.

Conclusion:

Binary and Bipolar Sigmoidal functions are crucial for activating neurons and controlling output ranges.

• **Binary Sigmoid: Output between 0 and 1 – suitable for probability models.**

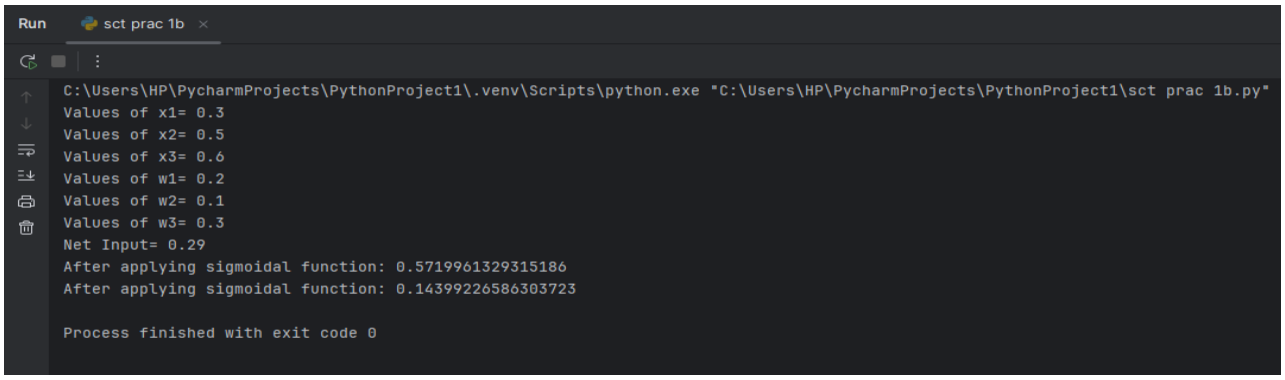
• **Bipolar Sigmoid: Output between –1 and +1 – suitable for classification problems.**

This experiment helps understand how weights, biases, and activation functions influence the neural network's output, forming the base for advanced AI models.

Code :

```
import math
x1= 0.3
x2=0.5
x3=0.6
print("Values of x1=",x1)
print("Values of x2=",x2)
print("Values of x3=",x3)
w1=0.2
w2=0.1
w3=0.3
print("Values of w1=",w1)
print("Values of w2=",w2)
print("Values of w3=",w3)
yin=x1*w1+x2*w2+x3*w3
print("Net Input=",yin)
funct_x1=1/(1+math.exp(-yin))     #Applying Binary sigmoidal function
print("After applying sigmoidal function:",funct_x1)
funct_x2=(1-math.exp(-yin))/(1+math.exp(-yin))     #Applying Bipolar sigmoidal function
print("After applying sigmoidal function:",funct_x2)
```

Output:



```
C:\Users\HP\PycharmProjects\PythonProject1\.venv\Scripts\python.exe "C:\Users\HP\PycharmProjects\PythonProject1\sct prac 1b.py"
Values of x1= 0.3
Values of x2= 0.5
Values of x3= 0.6
Values of w1= 0.2
Values of w2= 0.1
Values of w3= 0.3
Net Input= 0.29
After applying sigmoidal function: 0.5719961329315186
After applying sigmoidal function: 0.14399226586303723

Process finished with exit code 0
```

PRACTICAL: 2 A

Aim: GENERATE AND/NOT FUNCTION USING MCCULLOH-PITTS NEURAL NET.

Introduction:

The McCulloch-Pitts (M-P) neuron is a simple computational model that simulates basic logical functions using binary inputs (0 or 1), fixed weights, and a threshold value. In this experiment, the M-P neuron is used to generate AND and NOT functions by appropriately assigning weights and thresholds.

Methods:

1. **Input and Weight Setup:**

The user enters the number of input pairs (num_ip). Binary input values for x1 and x2 are taken from the user and stored in lists. Weights are assigned as w1 = 1 and w2 = -1, defining the effect of each input on the neuron output**.**

2. **Net Input Calculation:**

For each input pair, the net input is calculated using Yin=(x1×w1)+(x2×w2)Y_{in} = (x1 \times w1) + (x2 \times w2)Yin=(x1×w1)+(x2×w2) This gives the total weighted sum of all inputs.

3. **Threshold Application and Output Generation:**

A threshold value of 1 is applied. If Yin ≥ 1, the neuron output Y = 1; otherwise, Y = 0. This step decides whether the neuron “fires” (outputs 1) or not (outputs 0).

Packages Used:

Python Core: Conditional statements, loops, and printing results.

1 input() → to take user input for number of input pairs and values of x1 and x2

2 int() → to convert user input to integer (0 or 1)

3 for loop → used inside list comprehension to iterate over multiple inputs

4 List comprehension → to create lists for x1, x2, Yin, and Y efficiently

5 Arithmetic operators * and + → to calculate weighted sum (Yin)

6 Conditional expression 1 if yin >= 1 else 0 → to apply threshold logic

7 print() → to display net inputs and outputs

Advantages:

1. Simple to understand and implement using binary inputs and threshold logic.

2. Can simulate basic logical functions like AND, OR, and NOT.

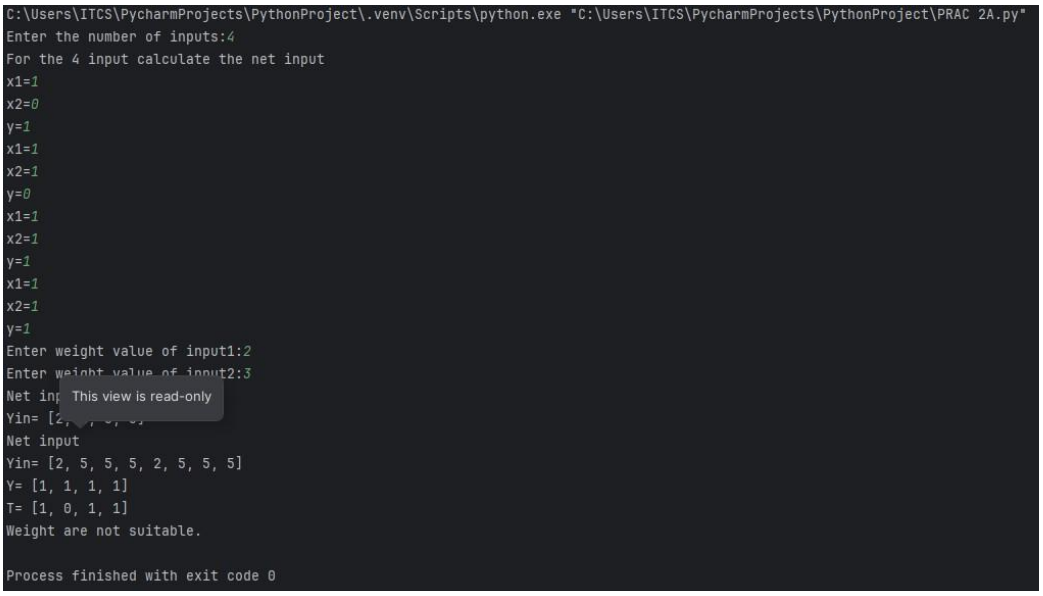
3. Forms the foundation for modern neural network concepts.

Conclusion: The McCulloch-Pitts model was implemented successfully, showing how basic logical operations can be simulated using simple neural computation principles.

Code:

```
num_ip=int(input("Enter the number of inputs:"))
print("For the", num_ip, "input calculate the net input")
x1=[]
x2=[]
t=[]
for i in range(0, num_ip):
    ele1=int (input("x1="))
    ele2=int (input("x2="))
    out=int (input ("y="))
    x1.append(ele1)
    x2.append(ele2)
    t.append(out)

w1=int(input("Enter weight value of input1:"))
w2=int(input("Enter weight value of input2:"))
n=[w1*i for i in x1]
m=[w2*i for i in x2]
Yin=[]

for i in range(0,num_ip):
    Yin.append(n[i]+m[i])
print("Net input")
print("Yin=",Yin)
Y=[]
for i in range(0,num_ip):
    Yin.append(n[i]+m[i])
print("Net input")
print("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)
print("T=",t)

if Y==t:
    print("Weight values accepted")
else:
    print("Weight are not suitable.")
```

Output :



```
C:\Users\ITCS\PycharmProjects\PythonProject\.venv\Scripts\python.exe "C:\Users\ITCS\PycharmProjects\PythonProject\PRAC 2A.py"
Enter the number of inputs:4
For the 4 input calculate the net input
x1=1
x2=0
y=1
x1=1
x2=1
y=0
x1=1
x2=1
y=1
x1=1
x2=1
y=1
Enter weight value of input1:2
Enter weight value of input2:3
Net inp
Yin= [2
Net input
Yin= [2, 5, 5, 5, 2, 5, 5, 5]
Y= [1, 1, 1, 1]
T= [1, 0, 1, 1]
Weight are not suitable.

Process finished with exit code 0
```

PRACTICAL: 2 B

Aim: GENERATE XOR USING MCCULLOH-PITS NEURAL NETWORK.

Introduction

The XOR (Exclusive OR) function cannot be implemented with a single M-P neuron because it is non linearly separable. A two-layer McCulloch-Pitts network is required, where intermediate neurons compute logical combinations (like AND, OR, NOT), and the final neuron produces the XOR output.

Methods:

1. Input and Weight Setup

1 The user enters the number of input pairs (num_ip)

2 Binary input values for x1 and x2 are taken from the user and stored in lists

3 Appropriate weights are assigned for intermediate neurons to compute AND, OR, and NOT operations

2 **.Net Input Calculation**

1 For each intermediate neuron, the net input is calculated using

Yin = (x1 * w1) + (x2 * w2)

2 This gives the total weighted sum of all inputs

3 **Threshold Application and Output Generation**

1 .A threshold is applied to the net input of each neuron. If Yin ≥ threshold, the neuron output Y = 1 otherwise Y = 0

2 .Outputs of intermediate neurons are combined in the final neuron to generate XOR

Keywords:

McCulloch–Pitts neuron, XOR logic, threshold function, multi-layer neural network, soft computing, non-linear separability.

Packages Used:

NumPy: Used for efficient array and matrix operations in neural computation.

· numpy.array() → to store input vectors for easier computation.

· numpy.dot() → to calculate the weighted sum (Yin) of inputs and weights.

· numpy.where() → optional, to apply threshold and generate binary outputs efficiently.

Python Core: Conditional statements, loops, and printing results.

input() → to take user input for number of pairs and values of x1 and x2. ·

int() → to convert input values to integers (0 or 1).

for loops → to iterate over multiple input pairs.

if-else statements → to apply threshold logic and generate neuron output.

print() → to display net inputs and final outputs.

Advantages:

1. Demonstrates the need for multi-layer neural networks for solving non-linear problems.
2. Provides insight into how complex logic can be built from simple neurons.
3. Reinforces the concept of activation thresholds and weighted summation.

Conclusion:

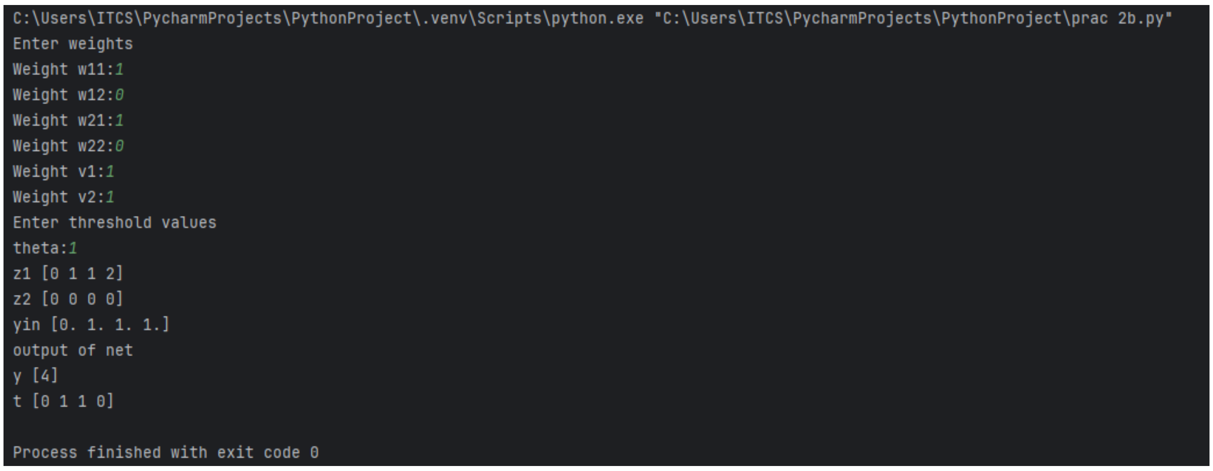
We implemented a two-layer McCulloch–Pitts network to generate the XOR function, showing that multi-layer models can solve problems a single neuron cannot — forming the basis of deep learning.

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 values")
theta=int(input("theta:"))

x1=np.array([0,0,1,1])
x2=np.array([0,1,0,1])
```

```
t=np.array([0,1,1,0])
z1=np.zeros((4,))
z2=np.zeros((4,))
y=np.array((4,))

zin1=np.zeros((4,))
zin2=np.zeros((4,))
zin1=(x1*w11)+(x2*w21)
zin2=(x1*w12)+(x2*w22)

print("z1", zin1)
print("z2", zin2)
for i in range(0,4):
    if zin1[i] >= theta:
        z1[i]=1
    else:
        z2[i]=0
    if zin2[i]>=theta:
        z2[i]=1
    else:
        z2[i]=0
yin=np.array([])
yin=(z1*v1)+(z2*v2)

for i in range(0,4):
    if yin[i]>=theta:
        yin[i]=1
    else:
        yin[i]=0
print("yin", yin)
print("output of net")
y=y.astype(int)
print("y",y)
print("t",t)
```

Output:



```
C:\Users\ITCS\PycharmProjects\PythonProject\.venv\Scripts\python.exe "C:\Users\ITCS\PycharmProjects\PythonProject\prac 2b.py"
Enter weights
Weight w11:1
Weight w12:0
Weight w21:1
Weight w22:0
Weight v1:1
Weight v2:1
Enter threshold values
theta:1
z1 [0 1 1 2]
z2 [0 0 0 0]
yin [0. 1. 1. 1.]
output of net
y [4]
t [0 1 1 0]

Process finished with exit code 0
```

PRACTICAL: 3 A

Aim: Program to implement Hebb's Rule.

Introduction:

Hebb's Learning Rule, proposed by psychologist Donald Hebb (1949), is one of the earliest and simplest neural learning algorithms.

It is based on the biological principle that "neurons that fire together, wire together." In this rule, the connection (weight) between two neurons strengthens when both neurons are active simultaneously.

Hebbian learning is unsupervised, meaning it does not require target outputs or error feedback. It is primarily used to learn associations and patterns from data.

Keywords

- **Hebb's Rule / Hebbian Learning: Strengthens connections between simultaneously active neurons.**
- **Weights: Represent the strength of neuron connections.**
- **Bias: Shifts the neuron activation function.**
- **Neural Network: Computational system inspired by biological neurons.**
- **Unsupervised Learning: Learning without labeled data.**
- **NumPy: Library for numerical computations.**

Packages

The only external package used is:

- **numpy (as np**): Essential for efficient array and matrix operations, particularly for handling the input patterns and weight vector.

Methodology

Initialize weights and bias to zero.

Provide input patterns and their corresponding output values

For each input pattern:

- Update weights using:

wi=wi+xi×yw_i = w_i + x_i \times ywi=wi+xi×y

- Update bias using:

b=b+yb = b + yb=b+y

Repeat for all training patterns.

Display the final updated weights and bias.

Hebb's Rule Formula

$\Delta wi = \eta \times xi \times y$

Δwi = xi ×y

wi new = wi old + Δwi

bnew = $bold$ +y

where:

- $\Delta wi \rightarrow$ change in weight
- $\eta \rightarrow$ learning rate (often taken as 1)
- $xi \rightarrow$ input value
- $y \rightarrow$ output (target) value

Conclusion

Hebb's Learning Rule provides a foundational concept of how learning occurs through synaptic strengthening.

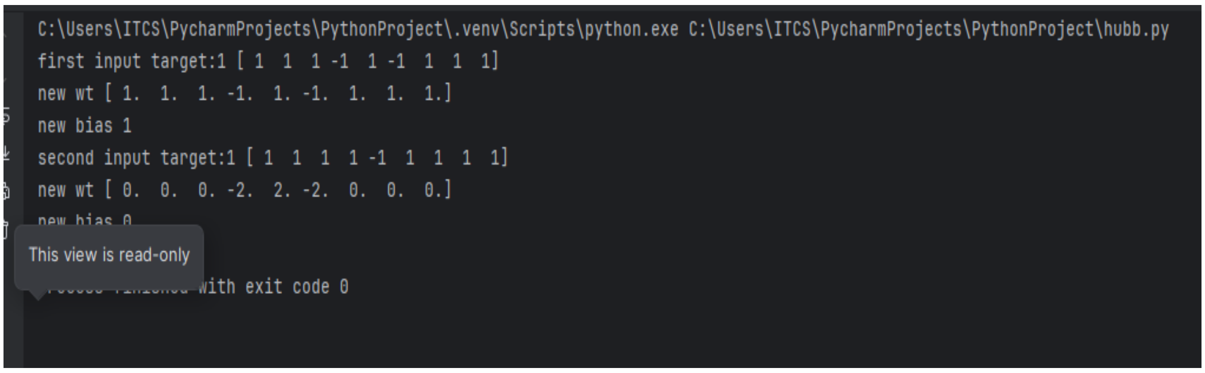
It updates weights and bias based on the co-activation of input and output neurons. Though simple, it laid the groundwork for modern neural learning algorithms and remains crucial in understanding unsupervised learning and pattern recognition

Code:

```
import numpy as np

#pip install numpy

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,))

bias=0
```

```
print('first input target:1',x1)
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('new bias',b)
print('second input target:1',x2)
for i in range(0,9):
    wtold[i]=wtold[i]+x2[i]*y[1]
b=b+y[1]
print('new wt',wtnew)
print('new bias',b)
```

Output:



```
C:\Users\ITCS\PycharmProjects\PythonProject\.venv\Scripts\python.exe C:\Users\ITCS\PycharmProjects\PythonProject\hubb.py
first input target:1 [ 1  1  1 -1  1 -1  1  1  1]
new wt [ 1.  1.  1. -1.  1. -1.  1.  1.  1.]
new bias 1
second input target:1 [ 1  1  1  1 -1  1  1  1  1]
new wt [ 0.  0.  0. -2.  2. -2.  0.  0.  0.]
new bias 0

Process finished with exit code 0
```

PRACTICAL: 3 B

Aim: Write a program to implement of delta rule.

Introduction

The **Delta Rule** (also called **Widrow–Hoff or LMS Rule**) is an error-correcting learning algorithm used to train single-layer neural networks.

It updates the connection weights based on the difference between the **desired output** and the **actual output**, aiming to **minimize the Mean Squared Error (MSE).**

This rule forms the basis of advanced methods like Gradient Descent and Backpropagation.

Methodology

The program follows **a supervised learning approach**, meaning the model learns from labeled input output pairs. The steps are:

1. Initialize input values, initial weights, desired outputs, and the learning rate.

2. Compute the actual output as the dot product of inputs and weights.

3. Calculate the error between the desired and actual outputs.

4. Update the weights according to the Delta Rule until the actual output matches the desired output.
5. Print the updated weights after learning is complete.

Formula

• **Error:** e=d−ye = d - ye=d−y

• **Weight Update**: winew=wiold+η(d−y)xiw_i^{new} = w_i^{old} + η (d - y) x_iwinew=wiold+η(d−y)xi

• **Mean Squared Error**: E=12(d−y)2E = \frac{1}{2}(d - y)^2E=21(d−y)2

Packages The only external package used is:

• **numpy (as np):** Essential for efficient array and matrix operations, particularly for handling the input patterns and weight vector.

Keywords

Delta Rule, Widrow–Hoff Learning, LMS Algorithm, Error Correction, Gradient Descent, Supervised Learning, ADALINE, Learning Rate, MSE, Weight Update.

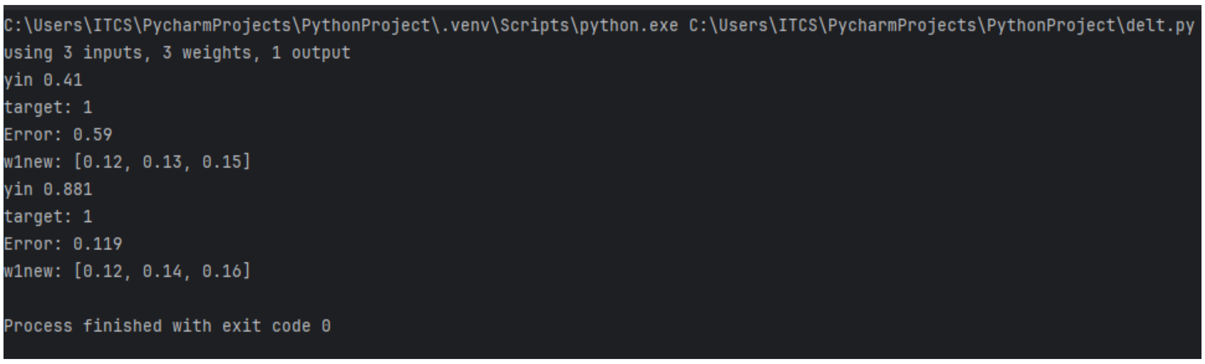
Conclusion

The Delta Rule effectively trains a neuron by minimizing the difference between actual and desired outputs. It demonstrates the principle of supervised learning through error correction, serving as the foundation for modern neural network training algorithms.

Code:

```
import math
print('using 3 inputs, 3 weights, 1 output')
x1=[0.3,0.5,0.8]
w1=[0.1,0.1,0.1]
t=1
a=0.1
diff=1
yin=0

while (diff > 0.4):
    for i in range(0,3):
        yin=yin+(x1[i]*w1[i])
    yin=yin+0.25
    yin=round(yin,3)
    print('yin',yin)
    print('target:',t)

    diff=t-yin
    diff=round(diff,3)
    diff=math.fabs(diff)
    print('Error:',diff)

    new1=[]
    for i in range(0,3):
        wnew1=w1[i]+a*diff*x1[i]
        wnew1=round(wnew1,2)
    new1.append(wnew1)
w1=new1
print('w1new:',w1)
```

Output:



```
C:\Users\ITCS\PycharmProjects\PythonProject\.venv\Scripts\python.exe C:\Users\ITCS\PycharmProjects\PythonProject\delt.py
using 3 inputs, 3 weights, 1 output
yin 0.41
target: 1
Error: 0.59
w1new: [0.12, 0.13, 0.15]
yin 0.881
target: 1
Error: 0.119
w1new: [0.12, 0.14, 0.16]

Process finished with exit code 0
```

PRACTICAL: 4 A

Aim: Perform Back Propogation Algorithm.

Introduction: Backpropagation calculates how much each weight in a neural network contributes to the total error. It uses gradient descent and derivatives to adjust these weights, reducing the error and improving the network's accuracy over time.

Packages Used:

• NumPy: it is used for fast numerical calculations, including matrix multiplications and vector operations, which are essential for neural network training and propagation.

• Math: It provides precise mathematical functions, like exponentials and activation calculations, to perform non-linear transformations in backpropagation.

• Matplotlib : used to visualize training results, such as loss curves or weight changes, helping to track and understand model performance over time.

Components

1. Input Layer: Receives raw data for processing.
2. Hidden Layers: Perform weighted computations and apply activation functions to learn patterns.
3. Output Layer: Produces the final prediction or classification result.
4. Weights & Biases: Adjustable parameters that determine the strength and direction of connections between neurons.

Method

Backpropagation is a supervised learning method that updates network parameters by minimizing the difference between predicted and actual outputs.

• Forward Pass: Inputs are processed through layers to generate outputs.

• Backward Pass: Errors are propagated backward using derivatives, and weights are updated using gradient descent to reduce error.

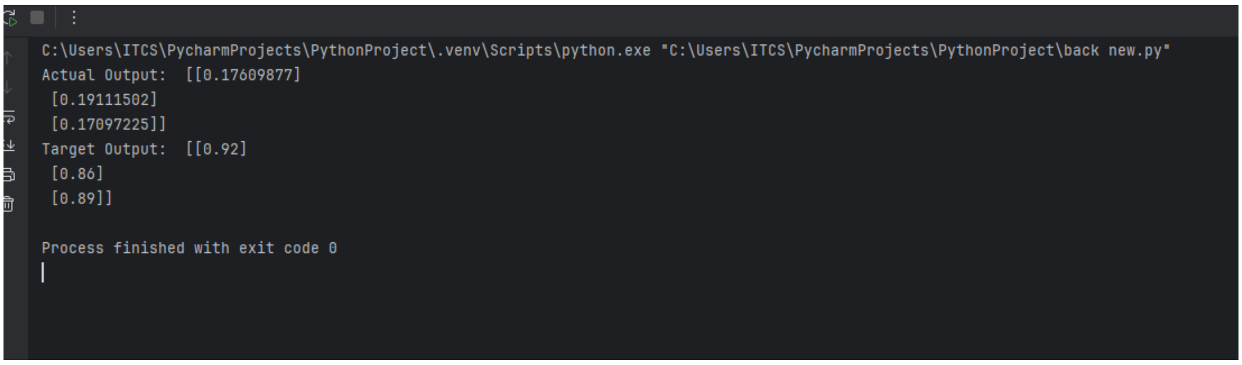
Conclusion

Backpropagation is the backbone of neural network learning. It enables models to automatically improve by minimizing error over time, contributing to breakthroughs in computer vision, natural language processing, and intelligent automation

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)

    def forward(self, x):
        self.z1 = np.dot(x, self.w1)
        self.z2 = self.sigmoidal(self.z1)
        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: ', str(op))
print('Target Output: ', str(y))
```

Output:



```
C:\Users\ITCS\PycharmProjects\PythonProject\.venv\Scripts\python.exe "C:\Users\ITCS\PycharmProjects\PythonProject\back new.py"
Actual Output:  [[0.17609877]
 [0.19111502]
 [0.17097225]]
Target Output:  [[0.92]
 [0.86]
 [0.89]]

Process finished with exit code 0
```

PRACTICAL: 4 B

Aim: Write a program Error Backpropagation Algorithm (EBPA) Algorithm.

Introduction

Error Backpropagation is a supervised learning algorithm used in neural networks to minimize prediction errors. It determines how much each weight contributes to the overall error and uses gradient descent and derivatives to adjust the weights, improving the model's accuracy over multiple iterations.

Packages Used

• NumPy: Used for efficient numerical operations such as matrix multiplications and vectorized calculations required during forward and backward passes.

• Math: Provides essential mathematical functions like exponentials and activation computations for non-linear transformations.

• Matplotlib (optional): Helps visualize training progress by plotting loss curves, accuracy metrics, and weight updates.

Components

1. Input Layer: Accepts the raw data for processing.
2. Hidden Layers: Processes inputs using weights and activation functions to extract meaningful patterns.
3. Output Layer: Produces the network's predicted output.
4. Weights & Biases: Adjustable parameters fine-tuned to reduce overall prediction error.

Method

Error Backpropagation works in two main stages:

• Forward Pass: Inputs are passed through the layers to produce an output prediction.

• Backward Pass: The algorithm calculates the error (difference between predicted and target outputs) and propagates it backward to update the weights using gradient descent.

This iterative process continues until the network's predictions are as close as possible to the desired outputs.

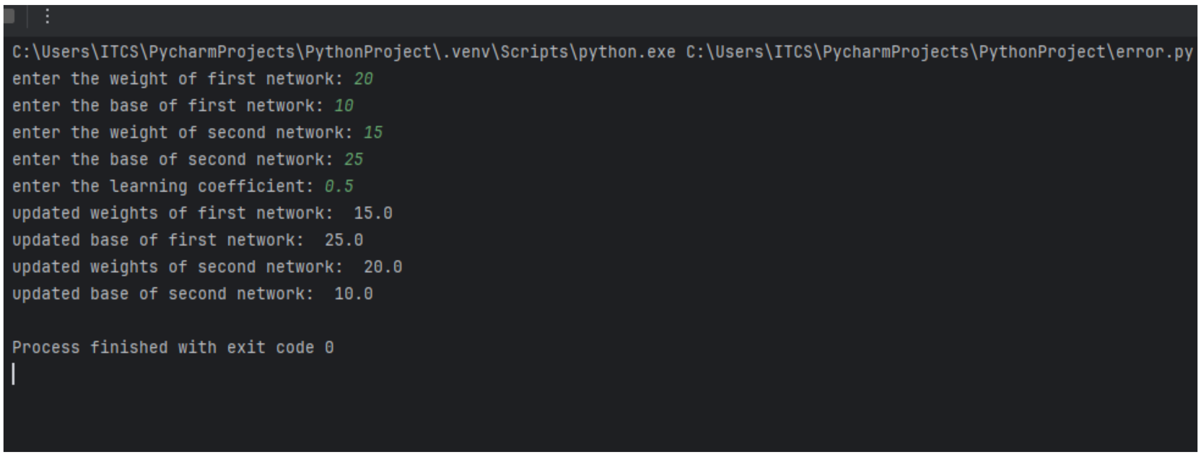
Conclusion

Error Backpropagation is the foundation of how neural networks learn. By continuously adjusting weights to minimize output error, it enables models to detect patterns, make predictions, and adapt intelligently—forming the core of modern deep learning applications.

Code:

```
import math
a0=-1
t=-1
w10=float(input('enter the weight of first network: '))
b10=float(input('enter the base of first network: '))
w20=float(input('enter the weight of second network: '))
b20=float(input('enter the base of second network: '))
c=float(input('enter the 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-a2*a2)*w20*s2
w21=w20-(c*s2*a1)
w11=w10-(c*s2*a0)
b21=b20-(c*s2)
b11=b10-(c*s1)
print('updated weights of first network: ', w21)
print('updated base of first network: ', b21)
print('updated weights of second network: ', w11)
print('updated base of second network: ', b11)
```

Output:



```
C:\Users\ITCS\PycharmProjects\PythonProject\.venv\Scripts\python.exe C:\Users\ITCS\PycharmProjects\PythonProject\error.py
enter the weight of first network: 20
enter the base of first network: 10
enter the weight of second network: 15
enter the base of second network: 25
enter the learning coefficient: 0.5
updated weights of first network:  15.0
updated base of first network:  25.0
updated weights of second network:  20.0
updated base of second network:  10.0

Process finished with exit code 0
```

PRACTICAL: 5 A

Aim: Perform Hopfield Network model for Associative memory.

Introduction

A Hopfield Network is a recurrent neural network used for associative memory.

It stores binary patterns and can recall the original pattern even when the input is noisy or incomplete.

The network updates neuron states iteratively until a stable memory pattern is reached.

Methodology

1. **Initialize Inputs & Weights:**

Define the size of the network and initialize the weight matrix as zeros.

2. **Train the Network:**

Update weights using the outer product of each training pattern. Set diagonal elements to zero to avoid self-connections**.**

3. **Pattern Recall:**

Provide a noisy input, compute weighted sums, and apply the sign function until the pattern stabilizes.

4. **Output:**

Display the recalled pattern to confirm successful memory retrieval

Libraries Used

NumPy (Numerical Python) : is a powerful Python library used for **scientific and numerical computing**.

It provides support for **multidimensional arrays**, **matrices**, and a wide range of **mathematical operations**, making it ideal for neural network computations

Keywords

- **Hopfield Network:** Recurrent model used for associative memory.
- **Weights:** Store learned associations between neurons.
- **Outer Product:** Operation for updating weights.
- **Recall:** Retrieving stored pattern from noisy input.
- **Sign Function:** Converts numeric values to binary states.

Conclusion

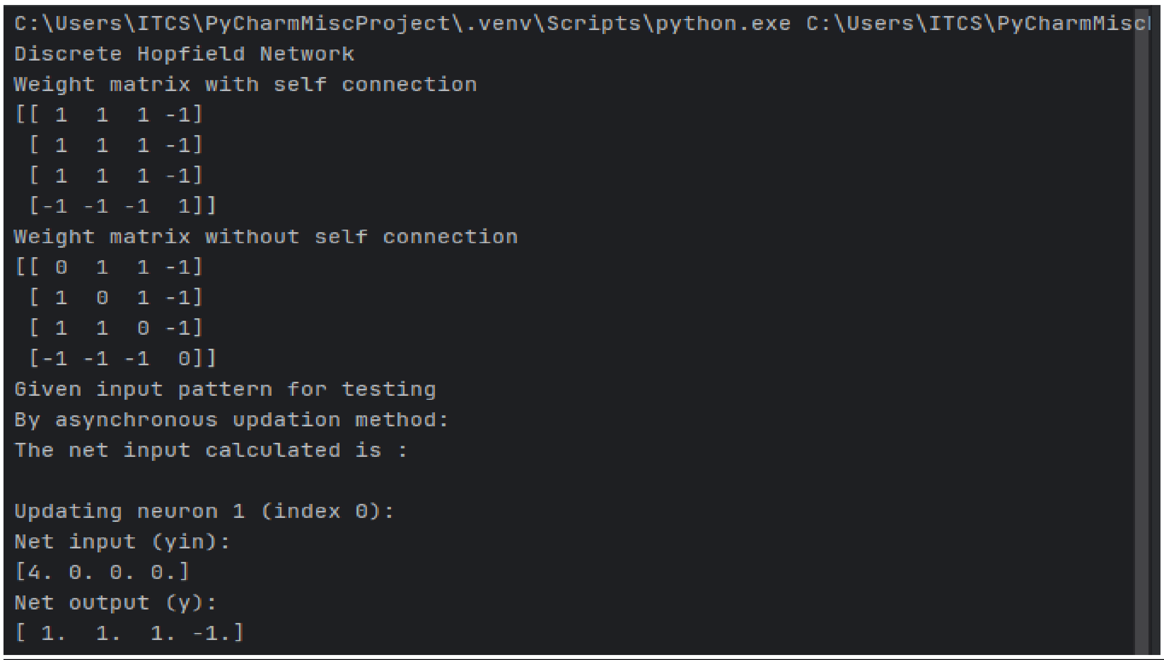
The **Hopfield Neural Network** effectively stores and recalls binary patterns, even from noisy inputs, demonstrating its power in **pattern recognition** and **associative memory** systems.

Code:

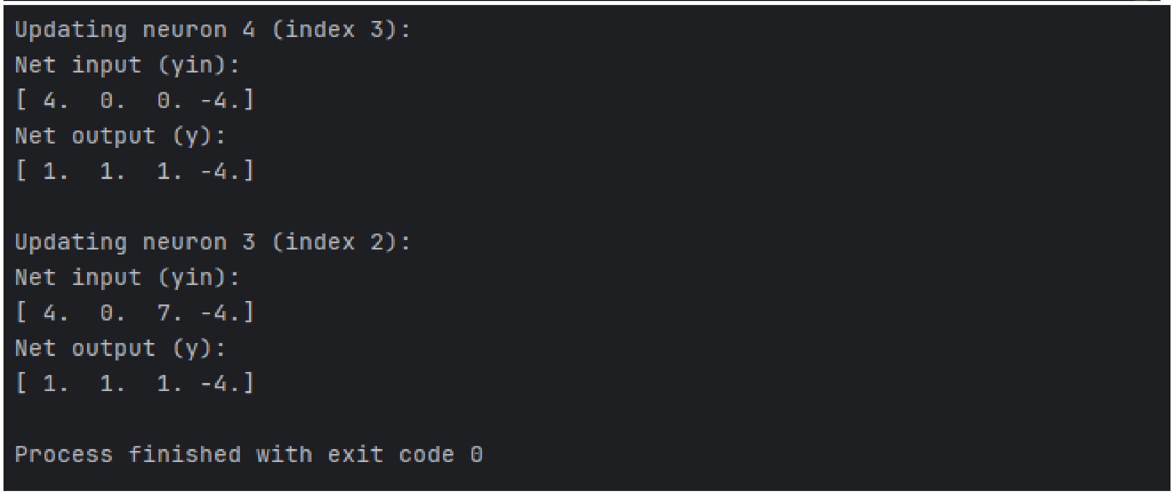
```
import numpy as np

def net(j_idx, w_matrix, input_pattern, threshold_val, yin_vector, y_vector):
    temp_sum = 0

    for i in range(4):
        temp_sum += (y_vector[i] * w_matrix[i, j_idx])

    yin_vector[j_idx] = input_pattern[j_idx] + temp_sum

    if yin_vector[j_idx] > threshold_val:
        y_vector[j_idx] = 1
    elif yin_vector[j_idx] < -threshold_val:

        y_vector[j_idx] = yin_vector[j_idx]
    else:
        y_vector[j_idx] = -1

    print("Net input (yin):")
    print(yin_vector)
    print("Net output (y):")
    print(y_vector)
    return yin_vector, y_vector

print("Discrete Hopfield Network")
theta = 0
x = np.array([1, 1, 1, -1])

w = np.outer(x, x)
print("Weight matrix with self connection")
print(w)

for i in range(4):
    w[i,i] = 0

print("Weight matrix without self connection")
print(w)

print("Given input pattern for testing")
```

```
x1 = np.array([1, 1, 1, -1])
y = np.array([1, 1, 1, -1], dtype=float)

print("By asynchronous updation method:")
print("The net input calculated is :")
yin = np.array([0, 0, 0, 0], dtype=float)

print("\nUpdating neuron 1 (index 0):")
yin, y = net(0, w, x1, theta, yin, y)

print("\nUpdating neuron 4 (index 3):")
yin, y = net(3, w, x1, theta, yin, y)

print("\nUpdating neuron 3 (index 2):")
yin, y = net(2, w, x1, theta, yin, y)
```

Output:



```
C:\Users\ITCS\PyCharmMiscProject\.venv\Scripts\python.exe C:\Users\ITCS\PyCharmMisc
Discrete Hopfield Network
Weight matrix with self connection
[[ 1  1  1 -1]
 [ 1  1  1 -1]
 [ 1  1  1 -1]
 [-1 -1 -1  1]]
Weight matrix without self connection
[[ 0  1  1 -1]
 [ 1  0  1 -1]
 [ 1  1  0 -1]
 [-1 -1 -1  0]]
Given input pattern for testing
By asynchronous updation method:
The net input calculated is :

Updating neuron 1 (index 0):
Net input (yin):
[4. 0. 0. 0.]
Net output (y):
[ 1.  1.  1. -1.]
```



```
Updating neuron 4 (index 3):
Net input (yin):
[ 4.  0.  0. -4.]
Net output (y):
[ 1.  1.  1. -4.]

Updating neuron 3 (index 2):
Net input (yin):
[ 4.  0.  7. -4.]
Net output (y):
[ 1.  1.  1. -4.]

Process finished with exit code 0
```

PRACTICAL: 5 B

Aim: To implement a Radial Basis Function (RBF) Neural Network in Python.

Introduction:

A Radial Basis Function (RBF) Network is a feedforward neural network used for function approximation and pattern recognition.

It uses radial basis functions (usually Gaussian) that depend on the distance between input data and fixed points called centers.

The network learns to model nonlinear relationships and predict new values.

Methodology:

1. **Initialize:** Define input/output dimensions, randomly select centers, initialize weights and spread (β).
2. **Compute Basis Function:**

 Use Gaussian RBF → $\phi(x) = \exp(-\beta\|x - c\|^2)$

3. **Activation Matrix (G)**: Evaluate RBFs for all inputs and centers.
4. **Train**: Compute weights using pseudo-inverse → W = pinv(G) × Y
5. **Predict:** Compute output for test data → Y = G × W 6. Visualize: Plot actual function, RBF output, and centers.

Libraries :

 1. **NumPy:**

A Python library for numerical and array-based computations, essential for matrix operations in neural networks. Functions used:

- for np.random.uniform(a, b, size) → Initializes random RBF centers.
- np.random.randn(d1, d2) → Initializes random weight matrix.
- np.dot(A, B) → Performs matrix multiplication.
- np.exp(x) → Computes exponential Gaussian RBF.

2. **SciPy (scipy.linalg):**

Provides advanced linear algebra operations used in RBF training.

Functions used:

pinv(G) → Computes the pseudo-inverse for optimal weight calculation.

norm(vector) → Calculates the **Euclidean distance** between inputs and centers

3. Matplotlib:

Used for data visualization and plotting results. Functions used:

plt.figure(figsize=(w, h)) → Creates a plotting window.

plt.plot(x, y, ...) → Plots true and predicted functions.

plt.legend() → Adds legends to the plot.

plt.show() → Displays the final graph.

Keywords:

RBF, Centers, Beta (β), Weights, Pseudo-inverse, Gaussian Function.

Conclusion:

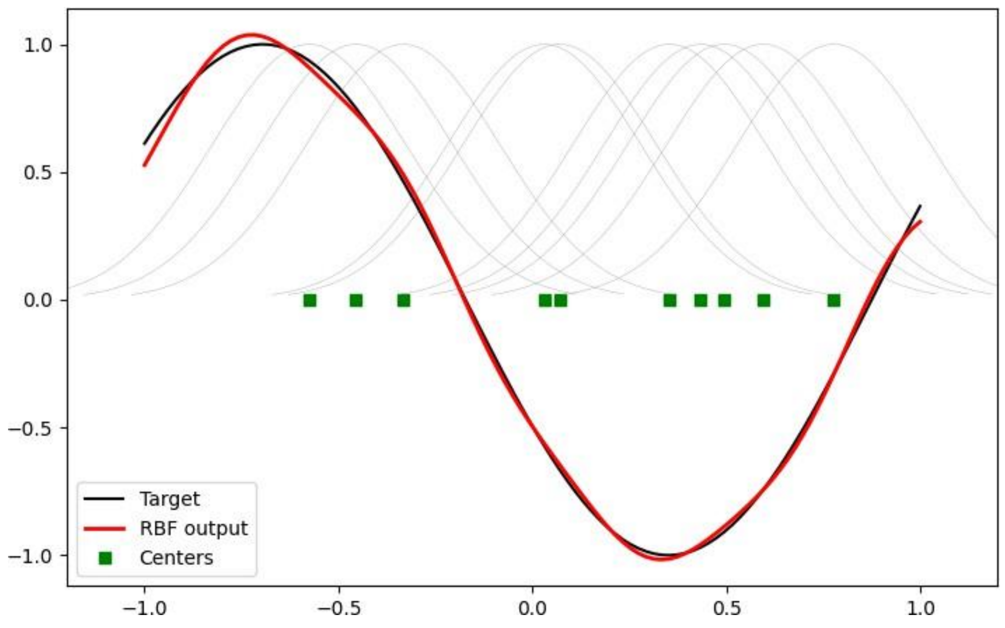
The **RBF Neural Network** accurately approximates nonlinear functions and predicts outputs from unseen data, proving its power in **regression** and **pattern recognition** applications.

Code:

```
import numpy as np

from scipy.linalg import norm, pinv

from matplotlib import pyplot as plt

from numpy import random, exp, zeros, dot, array, sin, arrange



class RBF:

    def _init_(self, indim, numCenters, outdim):

        self.indim = indim

        self.outdim = outdim

        self.numCenters = numCenters

        self.centers = [random.uniform(-1, 1, indim) for _ in range(numCenters)]

        self.beta = 8

        self.w = random.random((self.numCenters, self.outdim))
    def _basisfunc(self, c, d):
        assert len(d) == self.indim
        return exp(-self.beta * norm(c - d) ** 2)

    def _calcAct(self, X):
        G = zeros((X.shape[0], self.numCenters), float)
        for ci, c in enumerate(self.centers):
            for xi, xp in enumerate(X):
                G[xi, ci] = self._basisfunc(c, xp)
        return G

    def train(self, X, y):
        rnd_idx = random.permutation(X.shape[0])[:self.numCenters]
        self.centers = [X[i, :] for i in rnd_idx]
        G = self._calcAct(X)
        self.w = dot(pinv(G), y)

    def test(self, X):
        G = self._calcAct(X)
        y = dot(G, self.w)
        return y

if _name___== '_main_':
    n = 100
    x = np.mgrid[-1:1:complex(0, n)].reshape(n, -1)
    y = np.sin(3 * (x + 0.5**3 - 1))  # or adjust as you like

    rbf = RBF(1, 10, 1)
    rbf.train(x, y)
```

```
z = rbf.test(x)


plt.figure(figsize=(12, 8))
plt.plot(x, y, 'k-', label='Target')
plt.plot(x, z, 'r-', linewidth=2, label='RBF output')
plt.plot(rbf.centers, zeros(rbf.numCenters), 'gs', label='Centers')


for c in rbf.centers:
    cx = arange(c[0] - 0.7, c[0] + 0.7, 0.01)
    cy = [rbf._basisfunc(array([cx_]), array(c)) for cx_ in cx]
    plt.plot(cx, cy, '-', color='gray', linewidth=0.2)


plt.xlim(-1.2, 1.2)
plt.legend()
plt.show()
```

Output:



PRACTICAL: 6 A

Aim: To implement and understand the working of a Kohonen Self-Organizing Map (SOM) using Python for unsupervised learning and data clustering

A Self-Organizing Map (SOM) or Kohonen Map is an unsupervised neural network that maps highdimensional data into a 2D grid.

It helps visualize, cluster, and interpret complex data by organizing similar patterns close to each other.

SOM uses competitive learning, where neurons compete to respond to input data.

The neuron with the smallest distance from the input becomes the Best Matching Unit (BMU), and nearby neurons are adjusted to become more like the input — gradually forming an organized topological map**.**

Packages Used:

1. **NumPy:**

Used for numerical operations, array handling, and efficient matrix computations.

np.random.rand() → Initializes random weights np.linalg.norm() → Calculates Euclidean distance

2. **Matplotlib:**

Used for visualizing the trained SOM and displaying clusters.

plt.figure(), plt.imshow(), plt.colorbar() → For plotting SOM grids and data clusters

3. **MiniSom:**

A lightweight Python library specifically designed for implementing SOMs. MiniSom(x, y, input_len, sigma, learning_rate) → Initializes the SOM grid som.train_random(data, num_iteration) → Trains SOM on input data som.winner(x) → Finds the Best Matching Unit (BMU) for an input vector

Keywords:

SOM, Kohonen Map, BMU, Competitive Learning, Unsupervised Learning, Clustering, Topological Ordering

Conclusion:

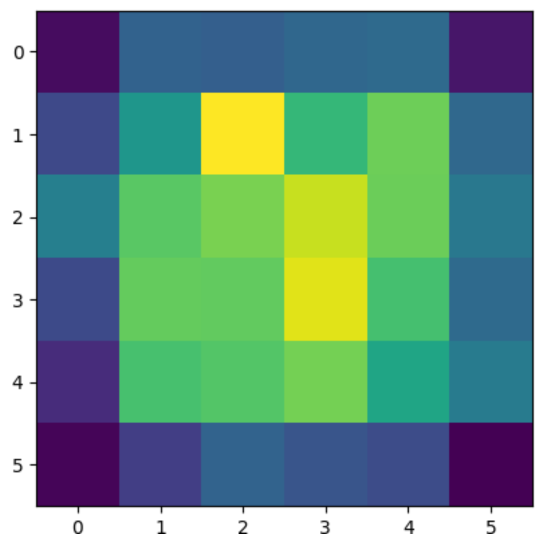
The Kohonen Self-Organizing Map effectively clustered similar data points, showing how unsupervised learning can organize and visualize complex data.

It serves as a powerful tool for data clustering, dimensionality reduction, and pattern recognition without requiring labeled datasets.

Code:

```
import numpy as np

from minisom import MiniSom

import matplotlib.pyplot as plt

data = [[0.8, 0.55, 0.22, 0.03], [0.82, 0.5, 0.23, 0.03], [0.8, 0.54, 0.22, 0.03], [0.8, 0.53, 0.26, 0.03],
[0.79, 0.56,0.22, 0.03], [0.75, 0.6, 0.25, 0.03], [0.77, 0.59, 0.22, 0.03]]

som = MiniSom(6, 6, 4, sigma = 0.4, learning_rate = 0.5)

som.train_random(data, 100)

plt.imshow(som.distance_map())

plt.show()
```

Output:



PRACTICAL: 6 B

Aim: Perform Adaptive Response Theory (ART).

Definition

Adaptive resonance theory (ART) is a neural network model that use unsupervised barning to category and recognize pattern. It can aduct to new information without forgetting old knowledge addressing the stability pascaly dilemmo

Methods

Competitor Learning Neurons in the recognition layer compete to respond to the import pattern. The winning neuron with the height activation is selected.

Vigilance Parameter. A thrushold parameter the deforming the level of similarity required for match the between the input pattern and a stored pattern

Function

ART is used for unsupervised learning pattern recognition and clustring. It is specially useful tasks for involving real time data streams and dynamic environments.

Library & Packages

• Munisom: A simple and efficient implementation of SOM, suitable for quick prototyping and Documentation

sabt learn' while not dedicated some library Saket learn provides the munisom as pevil of its sunsupervised learning module

Real World Example.

Financial Analysis - Analyzing stock Market data to identify pattern by trends.

Medical Imaging Classifying medical images Such X rays of MRIS

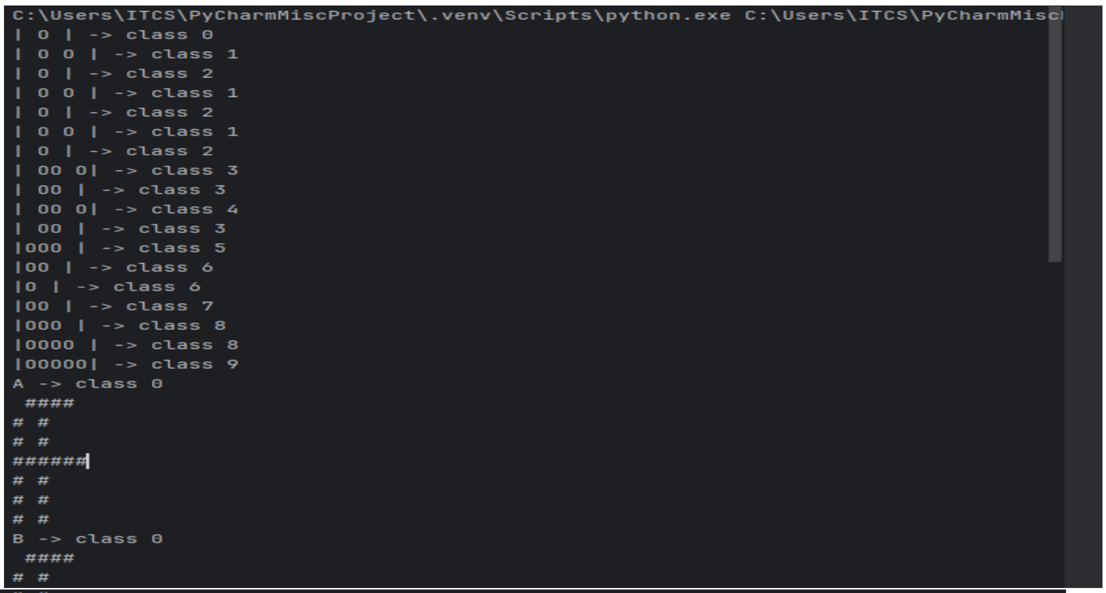
Code:

```
from _future___import print_function, division

import numpy as np


# Moved utility functions outside the class

def letter_to_array(letter):

    ''' Convert a letter to a numpy array '''

    # Determine the maximum width of the letters for consistent array shape

    max_width = max(len(row_str) for row_str in letter)
    shape = len(letter), max_width
    Z = np.zeros(shape, dtype=int)
    for row_idx, row_str in enumerate(letter):
        for column_idx in range(len(row_str)):  # Iterate only up to the length of the current row_str
            if row_str[column_idx] == '#':
                Z[row_idx][column_idx] = 1
    return Z
def print_letter(Z):
    ''' Print an array as if it was a letter '''
    for row in range(Z.shape[0]):
        for col in range(Z.shape[1]):
            if Z[row, col]:
                print('#', end="")
            else:
                print(' ', end="")
        print()

class ART:
    def _init_(self, n=5, m=10, rho=0.5):
        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):
        ''' Learn input X '''
        # Ensure X is of the correct input size (self.F1.size)
        # Pad or truncate X if necessary
        processed_X = np.zeros(self.F1.size)
        processed_X[:min(len(X), self.F1.size)] = X[:min(len(X), self.F1.size)]
        X = processed_X

        self.F2[...] = np.dot(self.Wf, X)
        I = np.argsort(self.F2[:self.active].ravel())[::-1]
        for i in I:
            # Handle division by zero if X.sum() is 0 (e.g., all zeros input)
            if X.sum() == 0:
                d = 0
            else:
                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)

    # First ART network for general patterns
    network = ART(5, 10, rho=0.5)  # Input size n=5
```

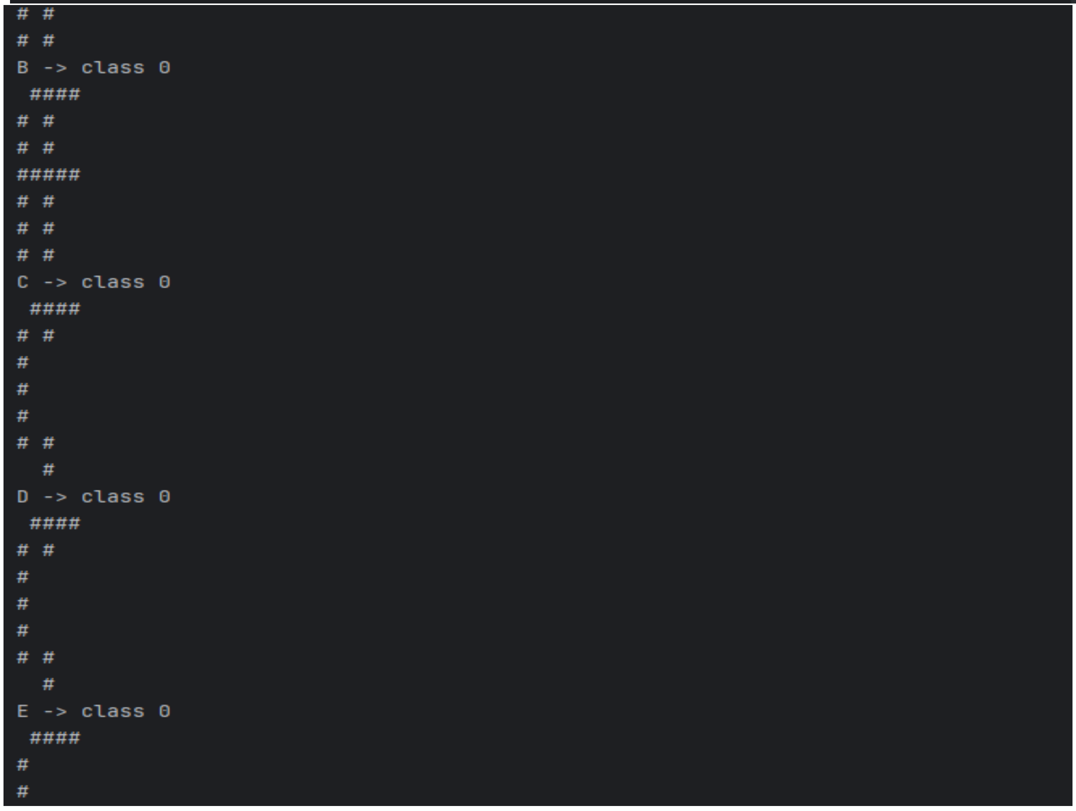
```
data = [" O ", " O O ", " O ", " O O ", " O ", " O O ", " O ", " OO O",
        " OO ", " OO O", " OO ", "OOO ", "OO ", "O ", "OO ", "OOO ", "OOOO ",
        "OOOOO"]

# Process each string in 'data' and feed it to the network
# Each input X needs to be of length 5 (n=5 for this network)
for i_idx, s in enumerate(data):
    # Convert string to binary array, pad/truncate to length 5
    input_vec = np.zeros(5)  # Expected input size for network
    for j_idx in range(min(len(s), 5)):  # Truncate if string is longer than 5
        input_vec[j_idx] = (s[j_idx] == 'O')
    Z, k = network.learn(input_vec)
    print(f"|{s}| -> class {k}")

# Second ART network for character recognition
A = letter_to_array([' #### ', '# #', '# #', '######', '# #', '# #', '# #'])
B = letter_to_array(['##### ', '# #', '# #', '##### ', '# #', '# #', '##### '])
C = letter_to_array([' #### ', '# #', '# ', '# ', '# ', '# #', ' #### '])
D = letter_to_array(['##### ', '# #', '# #', '# #', '# #', '# #', '##### '])
E = letter_to_array(['######', '# ', '# ', '#### ', '# ', '# ', '# '])
F = letter_to_array(['######', '# ', '# ', '#### ', '# ', '# ', '# '])
samples = [A, B, C, D, E, F]

# The second network expects input size 6*7 = 42, which matches the flattened letters.
network = ART(6 * 7, 10, rho=0.15)
for i in range(len(samples)):
    Z, k = network.learn(samples[i].ravel())
    print("%c" % (ord('A') + i), "-> class", k)
    if Z is not None:  # Check if a class was learned before trying to print
        print_letter(Z.reshape(7,6))
```

```
else:
    print("No class learned for this sample")
```

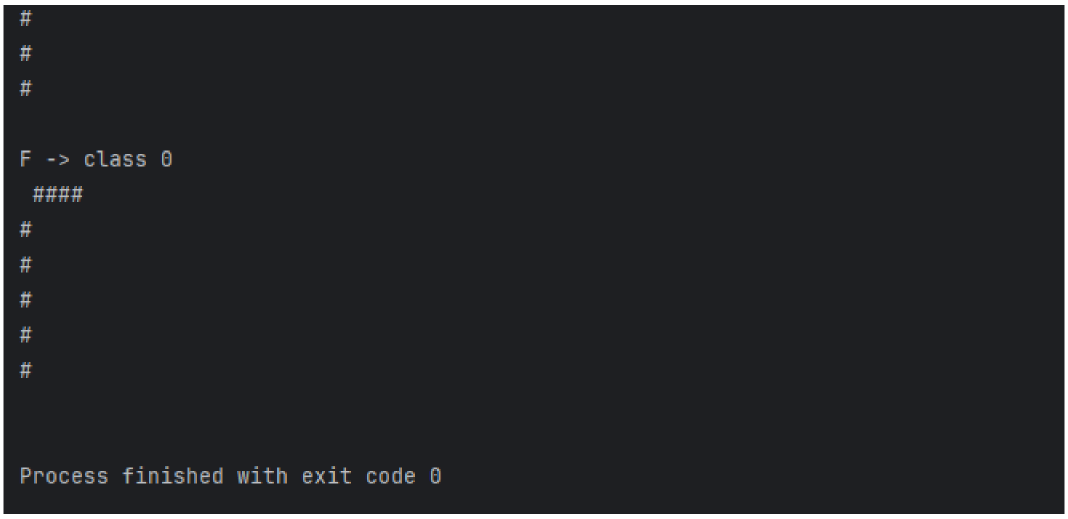
Output:



```
C:\Users\ITCS\PyCharmMiscProject\.venv\Scripts\python.exe C:\Users\ITCS\PyCharmMisc
| O | -> class 0
| O O | -> class 1
| O | -> class 2
| O O | -> class 1
| O | -> class 2
| O O | -> class 1
| O | -> class 2
| OO O| -> class 3
| OO | -> class 3
| OO O| -> class 4
| OO | -> class 3
|OOO | -> class 5
|OO | -> class 6
|O | -> class 6
|OO | -> class 7
|OOO | -> class 8
|OOOO | -> class 8
|OOOOO| -> class 9
A -> class 0
 ####
# #
# #
######
# #
# #
# #
B -> class 0
 ####
# #
```



```
# #
# #
B -> class 0
 ####
# #
# #
#####
# #
# #
# #
C -> class 0
 ####
# #
#
#
#
# #
  #
D -> class 0
 ####
# #
#
#
#
# #
  #
E -> class 0
 ####
#
#
```



```
#
#
#

F -> class 0
 ####
#
#
#
#
#


Process finished with exit code 0
```

PRACTICAL: 7

Aim: To write a program for Linear Separation.

Introduction:

Linear separability refers to the ability to separate data points into two classes using a straight line (in 2D) or a plane (in higher dimensions).

If such a boundary exists, the data is said to be linearly separable**.**

Method:

Perceptron Algorithm:

A simple supervised learning algorithm that iteratively adjusts weights to find a line (or plane) separating two data classes.

Support Vector Machine (SVM):

A powerful classification algorithm that finds the optimal separating line or plane with the maximum margin between two classes.

Function:

Linear separation enables data classification into different categories.

It forms the foundation of many machine learning algorithms like Logistic Regression and Support Vector Machines.

Conclusion:

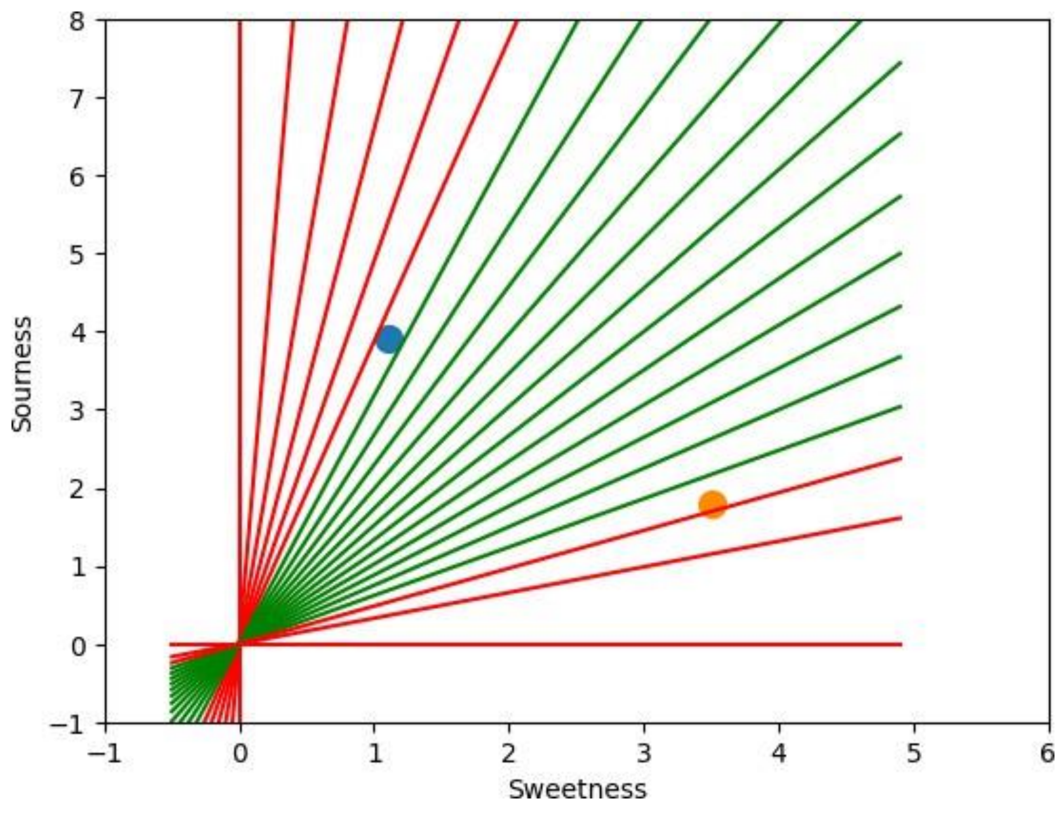
Linear Separation is a core concept in machine learning that allows data to be divided accurately into different classes using a straight line or plane.

Algorithms such as Perceptron and SVM rely on this concept for effective classification.

Code:

```
import numpy as np
import matplotlib.pyplot as plt
def create_distance_function(a, b, c):
    def distance(x, y):
        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, "o", 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: 8 A

Aim: To demonstrate Membership and Identity Operators (in, not in) in Python.

Introduction:

Membership operators are used to check whether a value exists within a sequence (like list, tuple, or string).

- **in → Returns True if the value is present.**
- **not in → Returns True if the value is not present.**

This program uses a function overlapping(list1, list2) to check if two lists share any common elements using the in operator**.**

Program Explanation:

1. Define a function overlapping(list1, list2).
2. Loop through elements in list1 and check if each element is in list2.
3. Return True if a match is found, otherwise False.
4. Display the result in the main program.

Keywords:

- in: Tests membership in a sequence.
- not in: Tests absence in a sequence.
- for loop: Iterates through list elements.
- return: Sends result from function. list: Stores multiple values.

Conclusion:

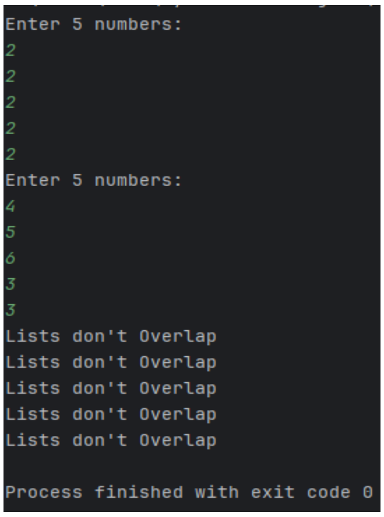
This program demonstrates how membership operators (in, not in) help check common elements between sequences. These operators are essential for search, comparison, and validation in Python programming**.**

Code:

```
list1 = []

print("Enter 5 numbers: ")
```

```
for i in range(0, 5):
    v = input()
    list1.append(v)
    list2 = []
print("Enter 5 numbers: ")
for i in range(0, 5):
    v = input()
    list2.append(v)
    flag =[]
for i in list1:
    if i in list2:
        flag = 1
    if (flag == 1):
        print("Lists Overlap")
    else:
        print("Lists don't Overlap")
```

Output:-



```
Enter 5 numbers:
2
2
2
2
2
Enter 5 numbers:
4
5
6
3
3
Lists don't Overlap
Lists don't Overlap
Lists don't Overlap
Lists don't Overlap
Lists don't Overlap

Process finished with exit code 0
```

PRACTICAL: 8 B

Aim: To demonstrate Identity Operators (is, is not) in Python.

Introduction:

Identity operators check whether two variables refer to the same object in memory. They differ from equality operators (==, !=), which compare values.

- is → Returns True if both refer to the same object.
- is not → Returns True if they refer to different objects.

This program compares variable types using identity operators.

Program Explanation:

1. Define variables x = 5 (int) and y = 5.2 (float).
2. Use type() to check each variable's data type.
3. Apply is and is not to compare whether the type matches int.
4. Display appropriate output messages**.**

Keywords:

- is: Checks if operands refer to the same object.
- is not: Checks if operands refer to different objects.
- type(): Returns an object's data type. int / float: Basic numeric data types.
- name: Returns the name of a class or type.

Conclusion:

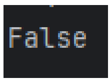
This program shows how identity operators (is, is not) compare object identity rather than value.

It clarifies the difference between object reference and value equality, which is crucial for debugging and data type validation in Python.

Code:-

```
x = 5
if(type(x) is not int):
    print('True')
else:
    print('False')
```

Output:-

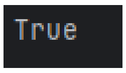


```
False
```

Code:-

```
x = 5
if(type(x) is int):
    print('True')
else:
    print('False')
```

Output:-

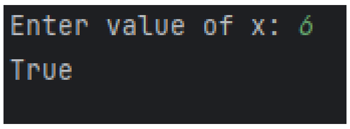


```
True
```

Code:-

```
x = int(input('Enter value of x: '))
if(type(x) is int):
    print('True')
else:
    print('False')
```

output:-

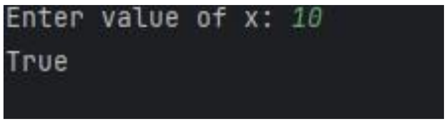


```
Enter value of x: 6
True
```

Code:-

```
x=int(input('Enter value of x: '))
if(type(x) is int):
    print('True')
else:
    print('False')
```

Output:-



```
Enter value of x: 10
True
```

PRACTICAL: 9 A

Aim: To find ratios using Fuzzy Logic.

Introduction:

Fuzzy logic ratios measure the similarity between strings or data values. They use algorithms like Levenshtein Distance or Token Ratio to show how close two items are, even if they are not exactly the same. A score near 100 means very similar, and near 0 means very different.

Explanation:

Fuzzy Logic is a way of reasoning that allows approximate values instead of just true or false. It helps in decision-making under uncertainty, such as in automation, AI, and data matching systems.

Components:

1. Fuzzifier: Converts exact data to fuzzy values.
2. Membership Function: Defines range between 0 and 1.
3. Inference Engine: Applies fuzzy rules.
4. Defuzzifier: Converts fuzzy result to crisp value.

Method:

A method is a function that performs operations or calculations. In fuzzy logic, methods calculate similarity and output matching ratios.

Libraries & Packages:

1. FuzzyWuzzy:

- Calculates string similarity using ratios.
- Used for matching names, spellings, etc.

2. Scikit-Fuzzy:

- Builds fuzzy logic systems.
- Used for control systems and decision-making.

Real-World Examples:

- FuzzyWuzzy: Used in search engines to find results even with wrong spellings.
- Scikit-Fuzzy: Used in self-driving cars, air conditioners, and smart devices to make decisions with uncertain data.

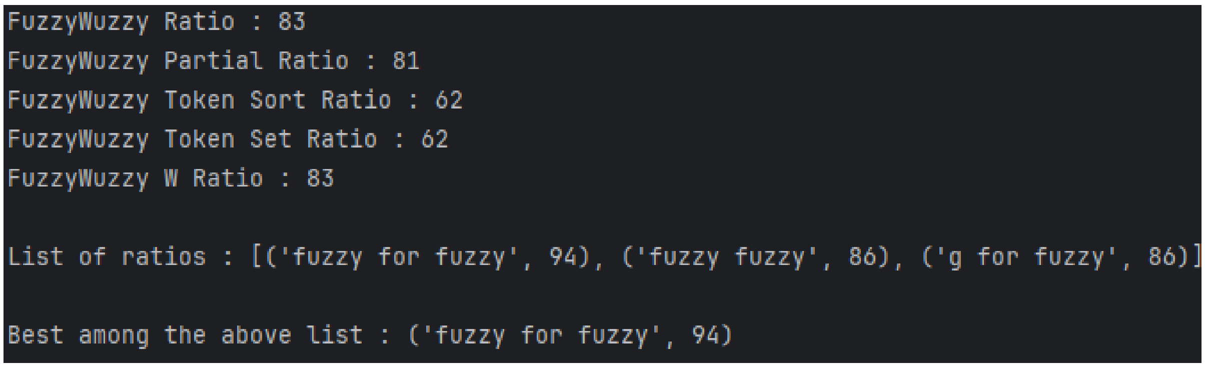
Conclusion:

Fuzzy logic helps in making intelligent decisions even when data is uncertain or incomplete. It is widely used in modern AI and automation systems.

Code:

```
from fuzzywuzzy import fuzz

from fuzzywuzzy import process


s1 = "I love fuzzyforfuzzys"

s2 = "I am loving fuzzy-forfuzzys"


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

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

print("FuzzyWuzzy Token Sort Ratio :", fuzz.token_sort_ratio(s1, s2))

print("FuzzyWuzzy Token Set Ratio :", fuzz.token_set_ratio(s1, s2))

print("FuzzyWuzzy W Ratio :", fuzz.WRatio(s1, s2), "\n")


query = "fuzzys for fuzzys"

choices = ["fuzzy for fuzzy", "fuzzy fuzzy", "g for fuzzy"]


print("List of ratios :", process.extract(query, choices), "\n")

print("Best among the above list :", process.extractOne(query, choices))
```

Output:-



```
FuzzyWuzzy Ratio : 83
FuzzyWuzzy Partial Ratio : 81
FuzzyWuzzy Token Sort Ratio : 62
FuzzyWuzzy Token Set Ratio : 62
FuzzyWuzzy W Ratio : 83

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

Best among the above list : ('fuzzy for fuzzy', 94)
```

PRACTICAL: 9 B

Aim: To solve the tipping problem using fuzzy logic.

Introduction:

The tipping problem uses fuzzy logic to simulate human-like reasoning when calculating an appropriate tip amount in a restaurant.

It evaluates factors such as food quality and service quality and applies fuzzy rules to determine the tip percentage

Explanation:

Fuzzy logic helps make decisions that are similar to human reasoning, allowing values between 0 and 1 instead of just true or false.

For example, if the food quality is good (0.8) and the service is average (0.5), the fuzzy system will use inference rules to compute a fair tip value**.**

Methodology:

1. Fuzzification:

Convert precise (crisp) input values such as food and service ratings into fuzzy sets using membership functions (e.g., poor, average, good).

2. Inference:

Apply fuzzy rules (in the form of IF–THEN statements) to logically combine inputs.

Example rule:

IF service is good AND food is good THEN tip is high.

3. Defuzzification:

Convert the fuzzy output (like low, medium, high) into a crisp value that represents the final tip percentage.

Real-World Examples:

1. Restaurant Tipping System:

A fuzzy logic system suggests a tip between 10% and 25% based on food and service ratings.

Example:

If the service is excellent but the food is average, the system may recommend a 17% tip instead of a flat 20%, showing nuanced reasoning.

2. Air Conditioner Temperature Control:

Fuzzy logic adjusts the cooling level based on room temperature and humidity.

Example:

IF temperature is slightly high AND humidity is moderate THEN fan speed is medium.

Libraries:

Libraries are reusable collections of code that perform specific functions, making coding simpler and more efficient.

Example: scikit-fuzzy – a Python library that helps in implementing fuzzy logic systems easily**.**

Packages:

Packages are organized containers that include libraries, data, and related resources, making it easier to manage and reuse code in multiple projects**.**

Conclusion:

The tipping problem demonstrates how fuzzy logic mimics human decision-making under uncertainty.

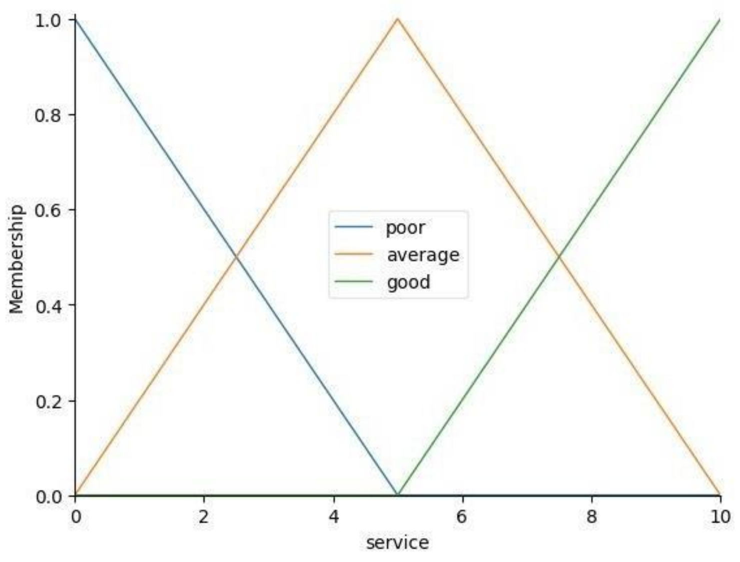
By using linguistic variables and fuzzy rules, it provides realistic, flexible, and logical outcomes for complex, real-world scenarios like restaurant tipping, temperature control, and smart appliances.

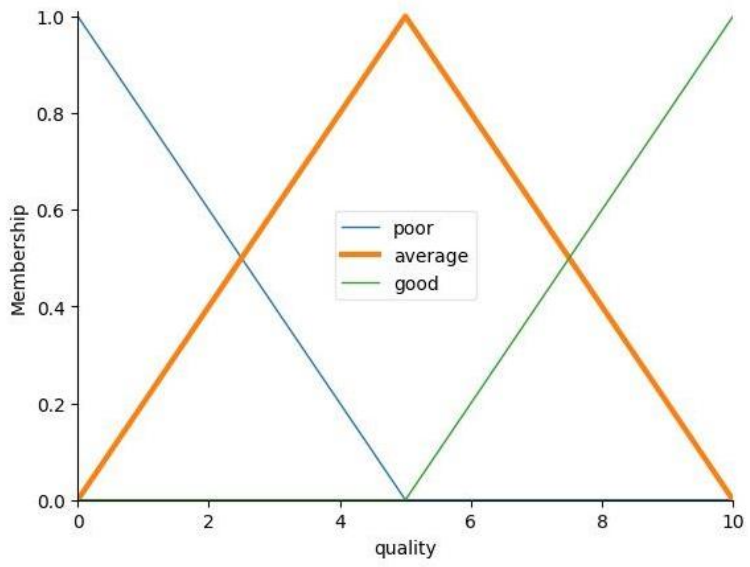
Code -

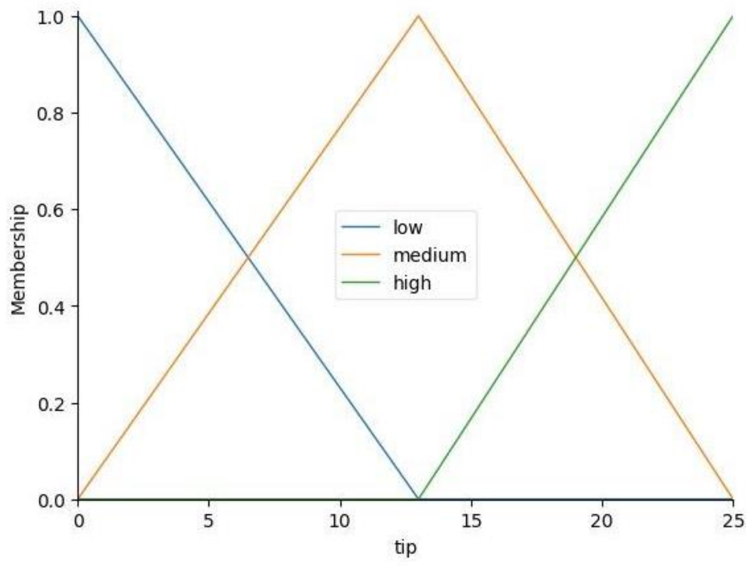
```
!pip install -U scikit-fuzzy

import numpy as np

import skfuzzy as fuzz

from skfuzzy import control as ctrl


quanlity = ctrl.Antecedent(np.arange(0, 11, 1), 'quantity')

service = ctrl.Antecedent(np.arange(0, 11, 1), 'quality')

tip = ctrl.Consequent(np.arange(0, 26, 1), 'tip')


quality.autom(3)

service.autom(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()
```

Output:


poor
average
good
Membership
service


poor
average
good
Membership
quality


low
medium
high
Membership
tip

PRACTICAL: 10 A

Aim: Simple Genetic Algorithm.

Introduction:

A Genetic Algorithm (GA) is an optimization and search technique inspired by natural selection and genetics.

It mimics biological evolution to find the best solution by repeatedly modifying a population of potential solutions.

Each individual (chromosome) represents a possible solution, and its fitness determines its chance of reproduction

in the next generation. Using selection, crossover, and mutation operations, the algorithm evolves over generations to produce better solutions

Keywords:

Genetic Algorithm, Chromosome, Mutation, Fitness Function, Population, Evolution, Crossover, Natural Selection.

Packages Used:

- random: Used to perform random selection, crossover, and mutation operations while generating new populations.

Algorithm:

1,Initialize Population:

Create a random population of chromosomes (strings) using a set of possible characters.

2. Calculate Fitness:

Measure how close each chromosome is to the target string. Fewer mismatches indicate higher fitness.

3. Selection:

Select the fittest individuals from the current generation based on their fitness scores.

4. Crossover:

Combine genes from two parent chromosomes to create a child chromosome using a probabilitybased method.

5,Mutation:

Randomly modify certain genes to introduce diversity and prevent early convergence.

6. Replacement:

Replace the old population with the newly generated population.

7. Termination:

Repeat the process until an individual perfectly matches the target string or the best fitness is achieved.

Advantages:

1. Handles complex, non-linear optimization problems effectively.
2. Works even when gradient information is not available.
3. Produces multiple solutions instead of a single local optimum.
4. Flexible and adaptable to a wide range of problem domains.
5. Encourages exploration of the solution space due to mutation and crossover operations.

Conclusion:

This experiment successfully demonstrates the working of a Simple Genetic Algorithm.

By simulating the process of natural evolution, the algorithm efficiently evolves a random population of strings

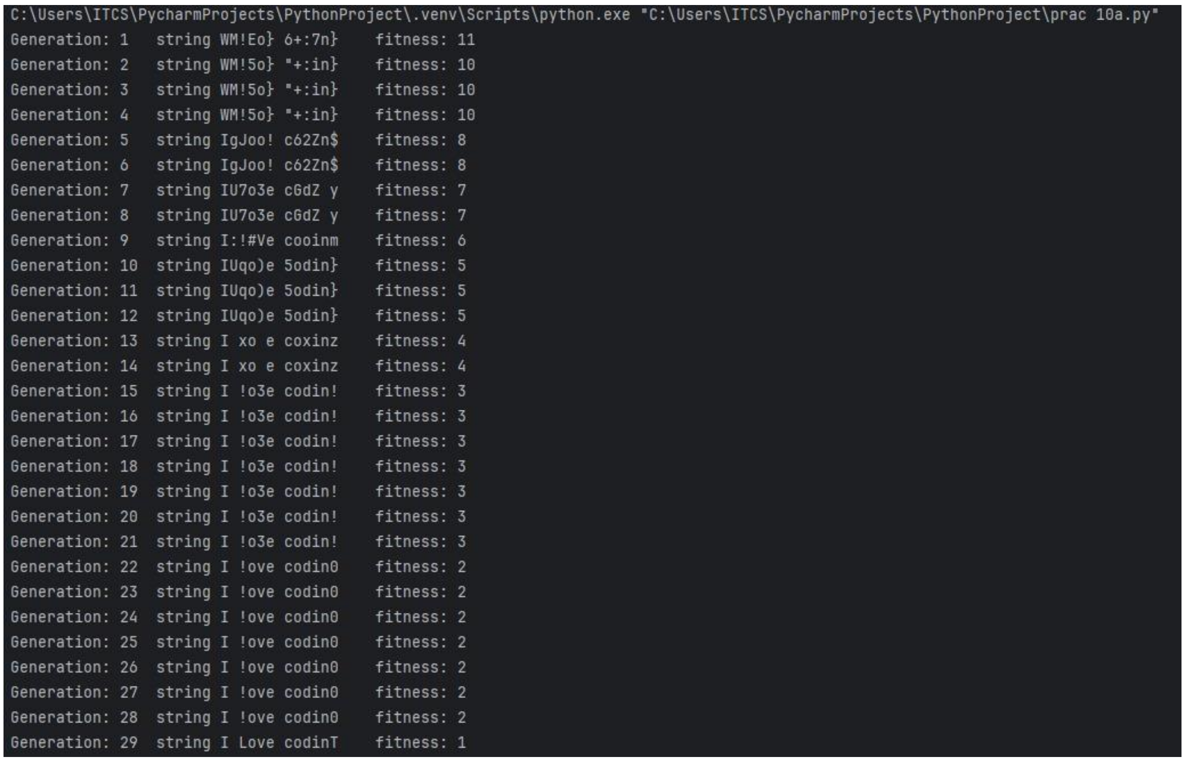
to exactly match a predefined target, showing how genetic principles can be applied to computational problem-solving.

Code:

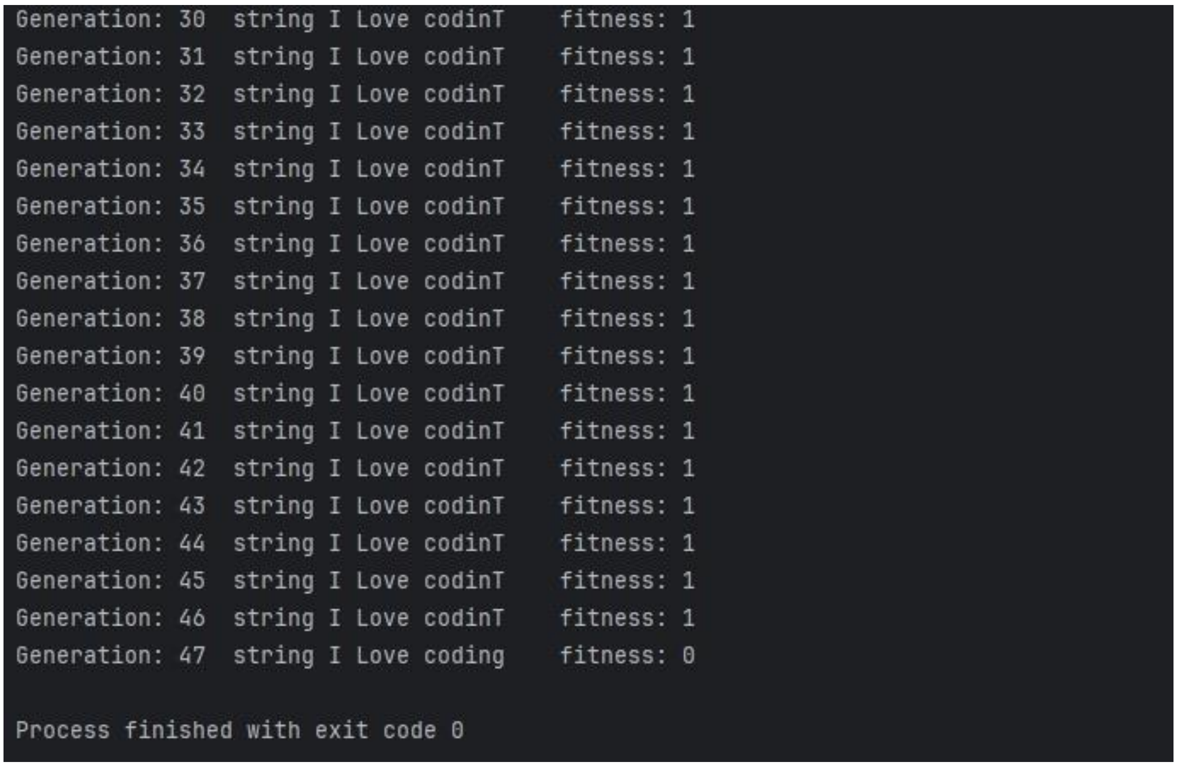
```
import random

POPULATION_SIZE = 100

GENES = '''abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ 1234567890, .-;:_!"#%&/()+?@${[]}'''

TARGET = "I Love coding"


class Individual(object):

    def __init__(self, chromosome):

        self.chromosome = chromosome

        self.fitness = self.cal_fitness()


    @classmethod

    def mutated_genes(cls):

        global GENES

        gene = random.choice(GENES)

        return gene
    @classmethod
    def create_gnome(cls):
        global TARGET
        gnome_len = len(TARGET)
        return [cls.mutated_genes() for _ in range (gnome_len)]

    def mate(self, par2):
        child_chromosome = []
        for gp1, gp2 in zip(self.chromosome, par2.chromosome):
            prob = random.random()
            if prob < 0.45:
                child_chromosome.append(gp1)
            elif prob < 0.90:
                child_chromosome.append(gp2)
            else:
                child_chromosome.append(self.mutated_genes())
        return Individual(child_chromosome)

    def cal_fitness(self):
        global TARGET
        fitness = 0
        for gs, gt in zip(self.chromosome, TARGET):
            if gs != gt:
                fitness += 1
        return fitness

def main():
    global POPULATION_SIZE
    generation = 1
    found = False
    population = []

    for _ in range(POPULATION_SIZE):
        gnome = Individual.create_gnome()
        population.append(Individual(gnome))

    while not found:
        population = sorted(population, key=lambda x: x.fitness)
        if population[0].fitness <= 0:
            found = True
            break

        new_generation = []

        s = int((10 * POPULATION_SIZE) / 100)
        new_generation.extend(population[:s])

        s = int((90 * POPULATION_SIZE) / 100)
        for _ in range(s):
            parent1 = random.choice(population[:50])
            parent2 = random.choice(population[:50])
            child = parent1.mate(parent2)
            new_generation.append(child)

        population = new_generation
        print("Generation: {}\tstring {}\tfitness: {}".format(generation,
"".join(population[0].chromosome), population[0].fitness))
        generation += 1

    print("Generation: {}\tstring {}\tfitness: {}".format(generation,
"".join(population[0].chromosome), population[0].fitness))
```

```
if __name___== '__main__':
    main()
```

Output:



```
C:\Users\ITCS\PycharmProjects\PythonProject\.venv\Scripts\python.exe "C:\Users\ITCS\PycharmProjects\PythonProject\prac 10a.py"
Generation: 1   string WM!Eo} 6+:7n}    fitness: 11
Generation: 2   string WM!5o} "+:in}    fitness: 10
Generation: 3   string WM!5o} "+:in}    fitness: 10
Generation: 4   string WM!5o} "+:in}    fitness: 10
Generation: 5   string IgJoo! c62Zn$    fitness: 8
Generation: 6   string IgJoo! c62Zn$    fitness: 8
Generation: 7   string IU7o3e cGdZ y    fitness: 7
Generation: 8   string IU7o3e cGdZ y    fitness: 7
Generation: 9   string I:!#Ve cooinm    fitness: 6
Generation: 10  string IUqo)e 5odin}    fitness: 5
Generation: 11  string IUqo)e 5odin}    fitness: 5
Generation: 12  string IUqo)e 5odin}    fitness: 5
Generation: 13  string I xo e coxinz    fitness: 4
Generation: 14  string I xo e coxinz    fitness: 4
Generation: 15  string I !o3e codin!    fitness: 3
Generation: 16  string I !o3e codin!    fitness: 3
Generation: 17  string I !o3e codin!    fitness: 3
Generation: 18  string I !o3e codin!    fitness: 3
Generation: 19  string I !o3e codin!    fitness: 3
Generation: 20  string I !o3e codin!    fitness: 3
Generation: 21  string I !o3e codin!    fitness: 3
Generation: 22  string I !ove codin0    fitness: 2
Generation: 23  string I !ove codin0    fitness: 2
Generation: 24  string I !ove codin0    fitness: 2
Generation: 25  string I !ove codin0    fitness: 2
Generation: 26  string I !ove codin0    fitness: 2
Generation: 27  string I !ove codin0    fitness: 2
Generation: 28  string I !ove codin0    fitness: 2
Generation: 29  string I Love codinT    fitness: 1
```



```
Generation: 30  string I Love codinT    fitness: 1
Generation: 31  string I Love codinT    fitness: 1
Generation: 32  string I Love codinT    fitness: 1
Generation: 33  string I Love codinT    fitness: 1
Generation: 34  string I Love codinT    fitness: 1
Generation: 35  string I Love codinT    fitness: 1
Generation: 36  string I Love codinT    fitness: 1
Generation: 37  string I Love codinT    fitness: 1
Generation: 38  string I Love codinT    fitness: 1
Generation: 39  string I Love codinT    fitness: 1
Generation: 40  string I Love codinT    fitness: 1
Generation: 41  string I Love codinT    fitness: 1
Generation: 42  string I Love codinT    fitness: 1
Generation: 43  string I Love codinT    fitness: 1
Generation: 44  string I Love codinT    fitness: 1
Generation: 45  string I Love codinT    fitness: 1
Generation: 46  string I Love codinT    fitness: 1
Generation: 47  string I Love coding    fitness: 0

Process finished with exit code 0
```

PRACTICAL: 10 B

Aim: Creating Classes with Genetic Algorithm.

Definition:-A genetic algorithm is a search and optimization technique inspired by natural selection that finds optimal or near optimal solution to compare problems.

Libraeies and packages:-DEAP(Distributed Evolutionary Algorithm in Python):-

Used to implementing various evolutionary algorithm in python.

PyGAD(Python Genetic Algorithm in Python) used for implementation and app of genetic algorithm for various optimization**.**

Random:-for generating random cities shutting router.

Math:-for computing distance.

Time:-for timng generation.

Numpy:-for numerical operations.

Matplotlib:-visualizing cities.

In spyred:-framework for an evolutionary computation with GA,PSO et.

Conclusion:-using the cities and fitness classes with genetic algorithm provider an efficient and modular way to solve optimization problem like ISP.It enables global search that reduce time and allow easy adaption for different fitness criteria.

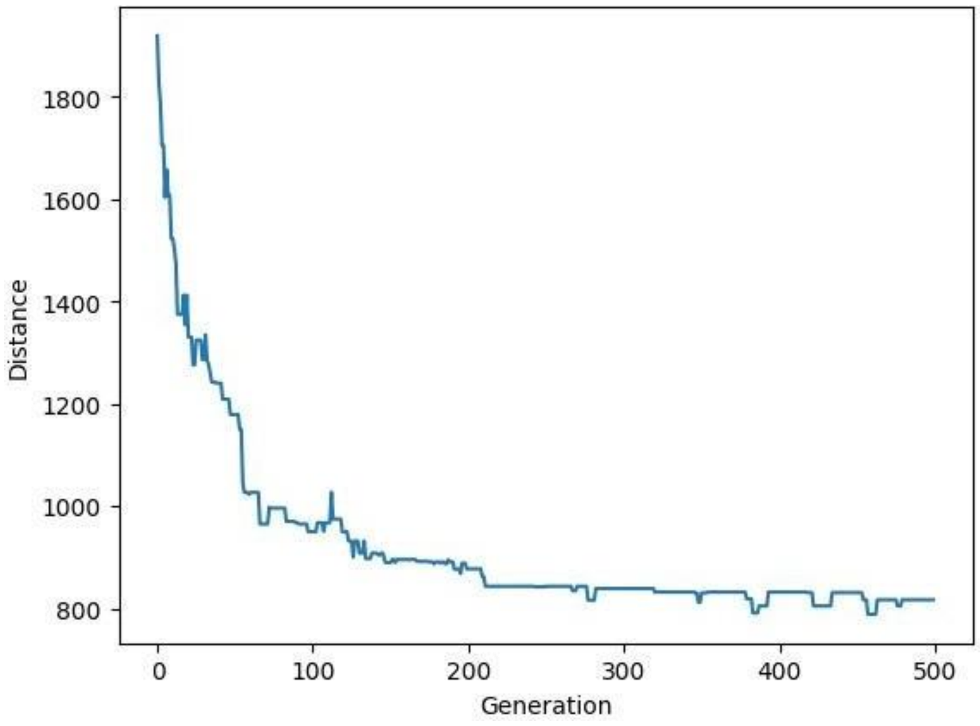
Code:

```
import random

import numpy as np, random, operator, pandas as pd, matplotlib.pyplot as plt

from tkinter import Tk, Canvas, Frame, BOTH, Text

import math

class City:

    def _init_(self, x, y):

        self.x = x

        self.y = y


    def distance(self, city):

        xDis = abs(self.x - city.x)

        yDis = abs(self.y - city.y)
        distance = math.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.distance == 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 = []
    child1 = []
    child2 = []

    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):
        child1.append(parent1[i])

    childP2 = [item for item in parent2 if item not in child1]
    child = child1 + 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_var = matingPool(currentGen, selectionResults) # Renamed to avoid conflict
    children = breedPopulation(matingpool_var, 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 = []

    for i in range(0, generations):
        pop = nextGeneration(pop, eliteSize, mutationRate)
        progress.append(1 / rankRoutes(pop)[0][1])

    plt.plot(progress)
    plt.ylabel('Distance')
    plt.xlabel('Generation')
    plt.show()


def main():
    cityList = []
    for i in range(0, 25):
        cityList.append(City(x=int(random.random() * 200), y=int(random.random() * 200)))
    geneticAlgorithmPlot(population=cityList, popSize=100, eliteSize=20, mutationRate=0.01,
generations=500)
if _name___== '_main_':
    main()
```

Output:


Distance
Generation