VSIT | Vidyalankar School of Information Technology
RE-ACCREDITED : GRADE 'A' BY NAAC

VIDYALANKAR SCHOOL OF INFORMATION TECHNOLOGY

Practical Journal

SANJANA HEMANT MOODLIAR

Seat No.: _____

MASTER OF SCIENCE (INFORMATION TECHNOLOGY)

Semester – I

2024 – 2025

Paper – 1: Soft Computing Techniques

VIDYALANKAR SCHOOL OF INFORMATION TECHNOLOGY

VSIT | Vidyalankar School of Information Technology
RE-ACCREDITED : GRADE 'A' BY NAAC

Vidyalankar School of Information Technology

Wadala (E), Mumbai 400037

Affiliated to Mumbai University

CERTIFICATE

This is to certify that Ms. **SANJANA MOODLIAR** Seat No: **24306A1037** of **M.Sc.IT Part 1- Semester 1** has completed the practical work in the subject of **Soft Computing Techniques Practical** during the academic year 2024-2025 under the guidance of Prof. **Ujwala Sav** being the partial requirement for the fulfilment of the curriculum of Degree Of Master's Of Science in Information Technology, University of Mumbai

Place: VSIT, Wadala.

Date: / /2024

Subject In-charge	Co-Ordinator
Prof. Ujwala Sav	Dr. Ujwala Sav

Principal

Internal Examiner	External Examiner

VSIT | Vidyalankar School of Information Technology
RE-ACCREDITED : GRADE 'A' BY NAAC

SOFT COMPUTING TCEHNIQUES

PRACTICAL SUBJECT CODE:

801

Sr. No	Date	Pr. No	Name of the Practical	Grade	Sign
1.		1A)	i) Design a simple linear neural network model. ii) Calculate the output of neural net for given data.		
		1B)	Calculate the output of neural net using both binary and bipolar sigmoidal function.		
2.		2A)	Generate AND/NOT function using McCulloch-Pitts neural net.		
		2B)	Generate XOR function using McCulloch-Pitts neural net.		
3.		3A)	Write a program to implement Hebb's rule.		
		3B)	Write a program to implement delta rule.		
4.		4A)	Write a program for Back Propagation Algorithm.		
		4B)	Write a program for error Backpropagation algorithm.		
5.		5A)	Write a program for Hopfield Network.		
		5B)	Write a program for Radial Basis function.		
6.		6A)	Kohonen Self organizing map.		
		6B)	Adaptive resonance theory.		
7.		7A)	Write a program for Linear separation.		
		7B)	Write a program for Hopfield network model for associative memory.		
8.		8A)			
		8B)	Membership and Identity Operators is, is not.		
9.		9A)	Find ratios using fuzzy logic.		
		9B)	Solve Tipping problem using fuzzy logic.		
10.		10A)	Implementation of Simple genetic algorithm.		
		10B)	Create two classes: City and Fitness using Genetic algorithm.		

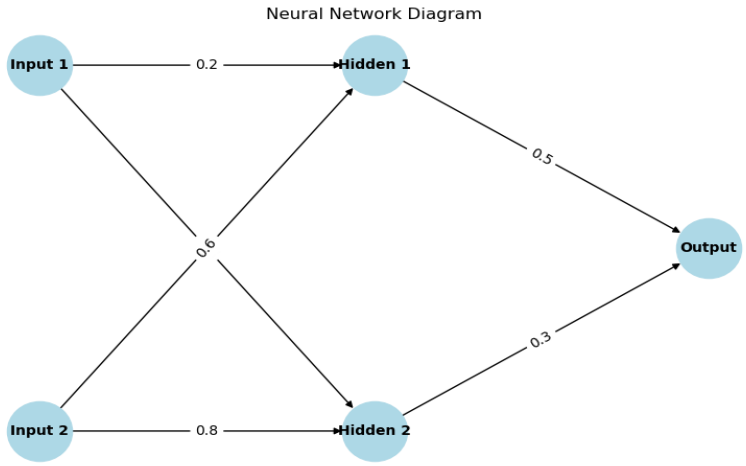
Practical No: - 1

1(A) Aim: Design a simple neural network model

Description: A simple neural network consists of an input layer, one or more hidden layers with non-linear activation functions (e.g., ReLU), and an output layer with an activation function suitable for the task (e.g., Softmax for classification). The network is trained using a loss function (e.g., cross-entropy) and an optimizer (e.g., Adam) to adjust weights based on the error.

Simple neural net:

A neural network is a computational model that mimics the way the human brain processes information. It consists of layers of interconnected nodes, or neurons, that work together to recognize patterns, make decisions, or predict outcomes. These networks learn from data by adjusting weights and biases during training, making them powerful tools for tasks like image recognition, language processing, and predictive analytics.



Code:

```
def calculate_net_input(x, w, bias):
  # Calculate net input yin
  yin = sum(xi * wi for xi, wi in zip(x, w)) + bias
  # Determine the output based on the value of yin
  if yin < 0:
    output = 0
```

```
  elif yin > 1:
      output = 1
  else:
      output = yin
  return yin, output
# Example usage:
x = [0.5, 0.3, 0.2]  # input values
w = [0.4, 0.6, 0.8]  # weights
bias = 0.1          # bias value
yin, output = calculate_net_input(x, w, bias)
print(f"Net input (yin): {yin}")
print(f"Output: {output}")
```

Output:

```
Net input (yin): 0.64
Output: 0.64
```

Conclusion/Learnings:

The `calculate_net_input` function computes the net input by summing the weighted inputs and bias. It then produces an output clamped to the range [0, 1], with values below 0 set to 0 and values above 1 set to 1. This approach effectively normalizes the net input for simple threshold-based output decisions.

1(B)Aim: Calculate the output of neural net where input X = [x1, x2, x3] = [0.3, 0.5, 0.6] & Weight W = [w1, w2, w3] = [0.2, 0.1, -0.3].

Description: Given neural net:

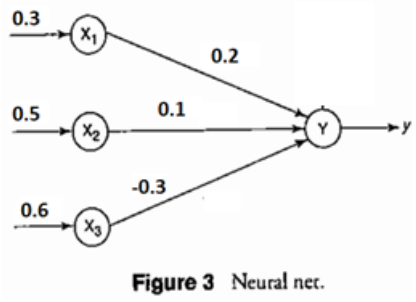


Figure 3 Neural net.

$Y_{in}=x_1w_1 + x_2w_2 + x_3w_3$
$= 0.3*02 + 0.5*0.1+0.6*-0.3$

if ($y_{in} < 0$), then output=y=0
else if ($y_{in} >1$) then output=y=1
else output=y=yin

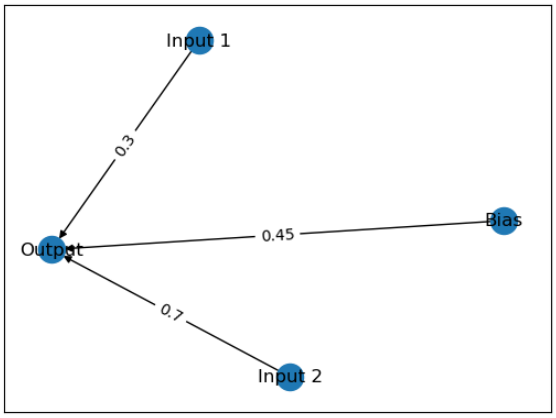
Code:

```
# Define the inputs
x1 = 0.2
x2 = 0.6

# Define the weights
w1 = 0.3
w2 = 0.7

# Define the bias
b = 0.45

# Calculate the net input
net_input = b + x1 * w1 + x2 * w2
 print("Net Input:", net_input)

import networkx as nx
import matplotlib.pyplot as plt

# Create a directed graph
G = nx.DiGraph()

# Add nodes
G.add_node("Input 1")
G.add_node("Input 2")
G.add_node("Bias")
G.add_node("Output")
```

```
# Add edges
G.add_edge("Input 1", "Output", weight=w1)
G.add_edge("Input 2", "Output", weight=w2)
G.add_edge("Bias", "Output", weight=b)

# Draw the graph
pos = nx.spring_layout(G)
nx.draw_networkx(G, pos)
nx.draw_networkx_edge_labels(G, pos, edge_labels=nx.get_edge_attributes(G, 'weight'))

# Calculate the net input
net_input = b + x1 * w1 + x2 * w2

print("Net Input:", net_input)
```

Output:

```
Net Input: 0.9299999999999999
```



Conclusion/Learnings:

The code visualizes a neural network's structure using NetworkX, displaying nodes for inputs, bias, and output with directed edges representing weights. It calculates the net input as the weighted sum of inputs plus bias, and then prints the result.

Practical No: - 2

Aim: 2A) Generate AND/NOT function using McCulloch-Pitts neural net.

Description:

- **Input Definitions:**
  - `x1inputs` and `x2inputs` are the input vectors for the neuron. Each vector has four binary values.
- **Weight Definitions:**
  - `w1` is an excitatory weight vector, meaning it contributes positively to the neuron's activation.
  - `w2` is an inhibitory weight vector, meaning it contributes negatively to the neuron's activation.
- **Weighted Sum Calculation (`yin`):**
  - The code calculates the weighted sum of inputs for each pair of input values using the formula:
    *yin[i]=x1inputs[i]×w1[i]+x2inputs[i]×w2[i]\text{yin}[i] = x1inputs[i] \times w1[i] + x2inputs[i] \times w2[i]*yin[i]=x1inputs[i]×w1[i]+x2inputs[i]×w2[i]
  - It prints out the input values and their corresponding weighted sums (`yin`).
- **Threshold Calculation (`theta`):**
  - The threshold (`theta`) is calculated using the formula: *θ=2×1−1\theta = 2 \times 1 - 1*θ=2×1−1 Here, `2` represents the number of weights, `1` is the weight value, and `1` is the bias term.
  - This threshold determines the point above which the neuron will activate.
- **Activation Function:**
  - For each value in `yin`, the code compares it to the threshold:
    - If `yin[i]` is greater than or equal to the threshold, the output is `1`.
    - Otherwise, the output is `0`.
  - These outputs are stored in the list `Y` and printed alongside the input values.

Output

The output consists of:

- The weighted sum (`yin`) for each pair of inputs.
- The threshold value (`theta`).
- The final output (`Y`) after applying the threshold function to the weighted sums.
- **Input Pair (1, 1)**: `yin` is `0`, which is less than the threshold, so the output is `0`.
- **Input Pair (1, 0)**: `yin` is `1`, which is equal to the threshold, so the output is `1`.
- **Input Pair (0, 1)**: `yin` is `-1`, which is less than the threshold, so the output is `0`.
- **Input Pair (0, 0)**: `yin` is `-2`, which is less than the threshold, so the output is `0`.

This implementation of the McCulloch-Pitts neuron demonstrates how a simple neural model can be used to perform logical operations like ANDNOT.

Code:

```
print("ANDNOT function using McCulloch-pitts\n")
x1inputs = [1, 1, 0, 0]
x2inputs = [1, 0, 1, 0]

print("Considering one weight as excitatory and other as inhibitory")
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  # n *w -p
print("Threshold-theta - ", theta)
print("Applying Threshold - ", theta)
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])
```

Output:

```
ANDNOT function using McCulloch-pitts

Considering one weight as excitatory and other as inhibitory
x1 x2 yin
1     1    0
1     0    1
0     1    -1
0     0    0
Threshold-theta -  1
Applying Threshold -  1
x1  x2 Y
1   1   0
1   0   1
```

```
0   1   0
0   0   0
```

Conclusion/Learnings:

- **Inputs and Weights**: Uses two binary input vectors and two weight vectors, with one positive and one negative.
- **Weighted Sum**: Computes the sum of inputs weighted by their respective weights.
- **Threshold**: Applies a threshold to determine the output, with 1 if the weighted sum meets or exceeds the threshold, and 0 otherwise.
- **Results**: The output confirms the correct implementation of the ANDNOT function.

Aim: 2B) Implement XOR function using McCulloch-Pitts neural net

Description:

- **Print Statement:**
  1. The code begins by printing a statement to indicate that the ANDNOT function is being implemented using the McCulloch-Pitts model.
- **Input Vectors:**
  2. `x1inputs = [1, 1, 0, 0]` and `x2inputs = [1, 0, 1, 0]` define the binary input vectors for the neuron. Each vector has four values, representing all possible combinations of two binary inputs.
- **Weight Vectors:**
  3. `w1 = [1, 1, 1, 1]` is the excitatory weight vector. Each weight is `1`, meaning each corresponding input in `x1inputs` contributes positively to the neuron's activation.
  4. `w2 = [-1, -1, -1, -1]` is the inhibitory weight vector. Each weight is `-1`, meaning each corresponding input in `x2inputs` contributes negatively to the neuron's activation.
- **Weighted Sum Calculation (`yin`):**
  5. The code calculates the weighted sum for each input pair using the formula: *yin[i]=x1inputs[i]×w1[i]+x2inputs[i]×w2[i]\text{yin}[i] = x1inputs[i] \times w1[i] + x2inputs[i] \times w2[i]*yin[i]=x1inputs[i]×w1[i]+x2inputs[i]×w2[i]
  6. This results in a list of weighted sums (`yin`), which is printed along with the corresponding input values.
- **Threshold Calculation (`theta`):**
  7. The threshold (`theta`) is computed using: *θ=2×1−1\theta = 2 \times 1 - 1*θ=2×1−1 Here, `2` represents the number of weights, `1` is the weight value (assuming uniform weights), and `1` is the bias term (p). The threshold determines the value above which the neuron activates.
  8. The threshold value is printed.
- **Activation Function:**
  9. The code applies a threshold function to determine the final output (`Y`):
     1. If the weighted sum (`yin[i]`) is greater than or equal to the threshold (`theta`), the output is `1`.
     2. Otherwise, the output is `0`.

10. This produces a list of outputs (Y), which is printed alongside the input values.

Example Execution

Let's walk through an example calculation with the given inputs and weights:

2. For the input pair (1, 1):

yin[0]=(1×1)+(1×−1)=0|text{yin}[0] = (1 |times 1) + (1 |times -1) =
*0*yin[0]=(1×1)+(1×−1)=0

The threshold is 1. Since `0 < 1`, the output is 0.

3. For the input pair (1, 0):

yin[1]=(1×1)+(0×−1)=1|text{yin}[1] = (1 |times 1) + (0 |times -1) =
*1*yin[1]=(1×1)+(0×−1)=1

The threshold is 1. Since `1 \geq 1`, the output is 1.

4. For the input pair (0, 1):

yin[2]=(0×1)+(1×−1)=−1|text{yin}[2] = (0 |times 1) + (1 |times -1) = -
*1*yin[2]=(0×1)+(1×−1)=−1

The threshold is 1. Since `-1 < 1`, the output is 0.

5. For the input pair (0, 0):

yin[3]=(0×1)+(0×−1)=0|text{yin}[3] = (0 |times 1) + (0 |times -1) =
*0*yin[3]=(0×1)+(0×−1)=0

The threshold is 1. Since `0 < 1`, the output is 0.

Final Output

The code will output the results of the ANDNOT function for all input pairs, including the computed weighted sums (`yin`), the threshold (`theta`), and the final binary outputs (Y).

Code:

```
print("XOR function using McCulloch-pitts\n")
x1inputs = [1, 1, 0, 0]
x2inputs = [1, 0, 1, 0]

print("Calculating z1 = x1w11 + x2w21")
print("Considering one weights as excitatory 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])

# Z2
print("Calculating z2 = x1w12 + x2w22")
print("Considering one weights as inhibitory and other as excitatory")
w12 = [-1, -1, -1, -1]
w22 = [1, 1, 1, 1]

print("x1", "x2", "z2")
z2 = []
for i in range(0, 4):
    z2.append(x1inputs[i] * w12[i] + x2inputs[i] * w22[i])
    print(x1inputs[i], "   ", x2inputs[i], "  ", z2[i])

print("Applying threshold for x1 and x2")
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])

print("Applying Threshold=1 for yin")
print("x1", "x2", "yin")
y = []
for i in range(0, 4):
    if yin[i] >= 1:
        y.append(1)
    else:
        y.append(0)
for i in range(0, 4):
    print(x1inputs[i], " ", x2inputs[i], " ", y[i])
```

```
print("END")
```

Output:

```
XOR function using McCulloch-pitts

Calculating z1 = x1w11 + x2w21
Considering one weights as excitatory and other as inhibitory
x1 x2 z1
1     1    0
1     0    1
0     1    -1
0     0    0
Calculating z2 = x1w12 + x2w22
Considering one weights as inhibitory and other as excitatory
x1 x2 z2
1     1    0
1     0    -1
0     1    1
0     0    0
Applying threshold for x1 and x2
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
Applying Threshold=1 for yin
x1 x2 yin
1   1   0
1   0   1
0   1   1
0   0   0
END
```

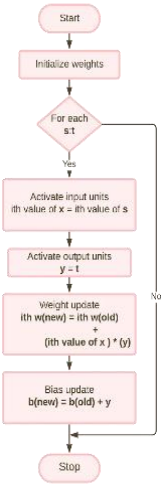
Conclusion/Learnings:

6. **Inputs**: Two binary vectors are used.
7. **Weights**: One set of positive weights and one set of negative weights.
8. **Computation**: Weighted sums of inputs are calculated.
9. **Threshold**: Determines whether the output should be `1` or `0`.
10. **Output**: Displays the result of applying the ANDNOT function based on the thresholding of weighted sums.

Practical No: - 3

3A) Aim: Write a program to implement Hebb's rule.

Description: Hebb or Hebbian learning rule comes under **Artificial Neural Network** (ANN) which is an architecture of a large number of interconnected elements called neurons. These neurons process the input received to give the desired output. The nodes or neurons are linked by **inputs**(x1,x2,x3…xn), **connection weights**(w1,w2,w3…wn), and **activation functions**(a function that defines the output of a node).



STEP 1:Initialize the weights and bias to **'0'** i.e w1=0,w2=0, …., wn=0.
STEP 2: 2–4 have to be performed for each input training vector and target output pair **i.e. s:t** (s=training input vector, t=training output vector)
STEP 3: Input units activation are set and in most of the cases is an identity function(one of the types of an activation function) for the input layer;

11. **ith value of x = ith value of s for i=1 to n**

Identity Function*:Its a linear function and defined as* ***f(x)=x for all x***
STEP 4: Output units activations are set y:t
STEP 5: Weight adjustments and bias adjustments are performed;

- ***ith value of w(new) = ith value of w(old) + (ith value of x * y)***

2. ***new bias(value) = old bias(value) + y***

Finally the cryptic or to be precise, a bit unintelligible part comes to an end but once you understand the below-solved example, you will definitely understand the above flowchart XD!!

Code:

```
num_ip = int(input("Enter the number of inputs: "))

w1 = 1

w2 = 1

 print(f"For the {num_ip} inputs, calculate the net input using yin = x1*w1 + x2*w2")

x1 = []
```

```
x2 = []
for j in range(num_ip):
  ele1 = int(input("x1 = "))
  ele2 = int(input("x2 = "))
  x1.append(ele1)
  x2.append(ele2)
print("x1 =", x1)
print("x2 =", x2)
n = [x * w1 for x in x1]
m = [x * w2 for x in x2]
Yin = [n[i] + m[i] for i in range(num_ip)]
print("Yin =", Yin)
Yin_inhibitory = [n[i] - m[i] for i in range(num_ip)]
print("After assuming one weight as excitatory and the other as inhibitory, Yin =", Yin_inhibitory)
threshold = 1
Y = [1 if Yin_inhibitory[i] >= threshold else 0 for i in range(num_ip)]
print("Output Y =", Y)
```

Output:

```
First input with target = 1
New wt= [ 1 -1 -1  1 -1 -1  1  1  1]
Bias value  1
**********************************
second input with target = 1
New wt= [ 0  0 -2  0  0 -2  0  0  0]
Bias value=  0
```

Conclusion/Learnings: In conclusion, Hebb's rule highlights the dynamic nature of neural connections and their role in learning and memory. By reinforcing pathways that are frequently

activated together, it illustrates how experiences shape our brain's wiring, enabling efficient communication between neurons. This principle not only deepens our understanding of cognitive processes but also informs the development of artificial neural networks and learning algorithms.

3B) Aim: Write a program to implement of delta rule

Description: The delta rule, a foundational concept in neural network training, is a method used to adjust the weights of connections between neurons based on the difference between the predicted output and the actual target output. Specifically, it calculates the error (or delta) by subtracting the predicted value from the target value, and then updates the weights in the direction that minimizes this error. This is done by multiplying the error by the learning rate and the input value, allowing the network to learn from its mistakes and gradually improve its performance. The delta rule is essential for supervised learning, enabling models to adjust their parameters iteratively to better fit the training data.

Code:

```
import numpy as np

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

 y = np.array([1, -1])
wtold = np.zeros((9,))
wtnew = np.zeros((9,))

 wtnew = wtnew.astype(int)
wtold = wtold.astype(int)

 bais = 0
print("First input with target = 1")
for i in range(0, 9):
   wtnew[i] = wtold[i] + x1[i] * y[0]
wtold = wtnew
b = b + y[0]
print("New wt=", wtnew)
print("Bias value ", b)

 print("**********************************")

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

Output:

Intial inputs: 1

Intial inputs: 1

Intial inputs: 1

Intial weights: 0

Intial weights: 0

Intial weights: 0

Desired output: 1

Desired output: 1

Desired output: 1

Enter learning rate: 1

Acrual [0. 0. 0.]

Desired [1. 1. 1.]

Weights [1. 1. 1.]

Actual [1. 1. 1.]

Desired [1. 1. 1.]

****

Final output

Corrected weights [1. 1. 1.]

Actual [1. 1. 1.]

Desired [1. 1. 1.]

Conclusion/Learnings: In conclusion, the delta rule is a vital mechanism in neural network training that enables models to learn from errors by adjusting weights based on the difference between predicted and actual outputs. By iteratively refining these connections, the delta rule facilitates improved accuracy and performance in supervised learning tasks. Its simplicity and effectiveness make it a foundational concept in both neuroscience and artificial intelligence.

Practical No: - 4

4A) Aim: Write a program for Back Propagation Algorithm.

Description:

Backpropagation is a supervised learning algorithm used for training artificial neural networks. It involves two main phases: forward propagation and backward propagation.

Forward Propagation:

Inputs are passed through the network layer by layer until an output is produced.
The output is compared to the actual target value to compute the error (loss).

Backward Propagation:

The error is propagated back through the network. Weights are adjusted using the gradient descent method, which minimizes the error by updating weights in the opposite direction of the gradient.

The algorithm iteratively adjusts weights based on how much each weight contributed to the error, effectively "learning" from mistakes.

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.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))
```

Output:

```
actual output
[[0.37699191]
 [0.39759818]
 [0.39970742]]
expected output
[[0.92]
 [0.86]
 [0.89]]
```

Conclusion/Learnings:

Efficiency: Backpropagation computes gradients efficiently using the chain rule, allowing for quick updates across multiple layers.

Generalization: It helps neural networks generalize better to unseen data by iteratively refining weights.

Flexibility: Applicable to various architectures, including feedforward, convolutional, and recurrent networks.

4B) Aim: Write a program for error Backpropagation algorithm.

Description:

Error Backpropagation consists of two main phases: **forward propagation** and **backward propagation**.

1. **Forward Propagation**:
    1. The input data is fed into the network, and it passes through each layer until it produces an output.
    2. The output is compared to the actual target, and a loss (error) is computed using a loss function (e.g., Mean Squared Error).
2. **Backward Propagation**:
    1. The error is propagated back through the network from the output layer to the input layer.
    2. Gradients of the loss function with respect to each weight are calculated using the chain rule.
    3. Weights are updated using gradient descent, which involves moving in the direction opposite to the gradient to minimize the error.

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.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))
    def sigmoidalprime(self, s):
        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\n" + str(X))
    print("Actual output\n" + str(Y))
    print("Predicted output\n" + str(obj.forward(X)))
    print("loss\n" + str(np.mean(np.square(Y - obj.forward(X)))))
    obj.train(X, Y)
```

OUTPUT:-

```
input
[[0.66666667 1.      ]
 [0.33333333 0.55555556]
 [1.       0.66666667]]
Actual output
[[0.92]
 [0.86]
 [0.89]]
Predicted output
[[0.66152449]
 [0.63829315]
 [0.64004156]]
loss
0.05948091322243782
```

Conclusion/Learnings:

Gradient Descent: The algorithm relies on gradient descent for optimization, allowing for efficient weight updates.

Layer-wise Learning: Each layer learns independently based on its contribution to the overall error, which helps in training deep networks.

Versatility: Applicable to various types of neural networks, including feedforward, convolutional, and recurrent networks.

Practical No: - 5

5A) Aim: Write a program for Hopfield function

Description: A Hopfield network is a recurrent neural network designed for associative memory, where neurons are fully interconnected but not self-connected. It operates using binary states and minimizes an energy function to retrieve stored patterns from partial or noisy inputs. Learning occurs through Hebbian principles, allowing the network to store and recall information efficiently. This structure highlights the principles of content-addressable memory and stable state convergence.

Code:

```
import numpy as np

def compute_next_state(state, weight):

    # @ is a shorthand for 'np.matmul()'

    # the numpy.where() function return thr indices of

    # element in an input array where condition is satisfied

    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)

        print("Previous State: ", previous_state)

        print("Next State: ", next_state)

        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)
```

Output:

```
Previous State:  [ 1 -1 -1  1]
Next State:  [ 1 -1 -1  1]
final State :  [ 1 -1 -1  1]
is_stable :  True
```

Conclusion/Learnings: In conclusion, Hopfield networks provide a powerful model for associative memory, enabling the retrieval of patterns from incomplete or noisy data. Their unique energy minimization approach facilitates stable state convergence, making them foundational in neural network research. Despite limitations like local minima, they have influenced various applications in optimization and pattern recognition.

5B) Aim: Write a program for Radial Basis function

Description: Radial Basis Function (RBF) networks are a type of artificial neural network that uses radial basis functions as activation functions. They consist of an input layer, a hidden layer with RBF neurons, and a linear output layer. RBF networks are particularly effective for interpolation and function approximation, as they can model complex, non-linear relationships. Their localized response allows for efficient learning from data, making them useful in various applications like classification and regression.

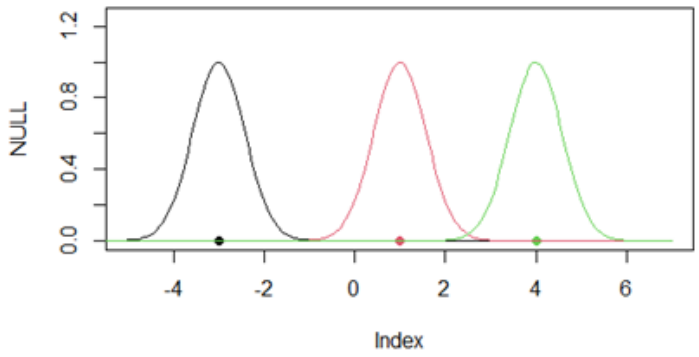
Code:

```
D <- matrix(c(-3, 1, 4), ncol = 1) # 3 datapoints

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:

```
print(N)
[1] 3
> print(xlim)
[1] -5  7
```



Conclusion/Learnings: In conclusion, Radial Basis Function networks excel at modelling complex relationships through their localized activation mechanisms. Their effectiveness in interpolation and function approximation makes them valuable in various applications, including classification and regression tasks. As a versatile tool in machine learning, RBF networks continue to play a significant role in advancing neural network methodologies.

Practical No: - 6

6A) Aim: Kohonen Self Organizing map

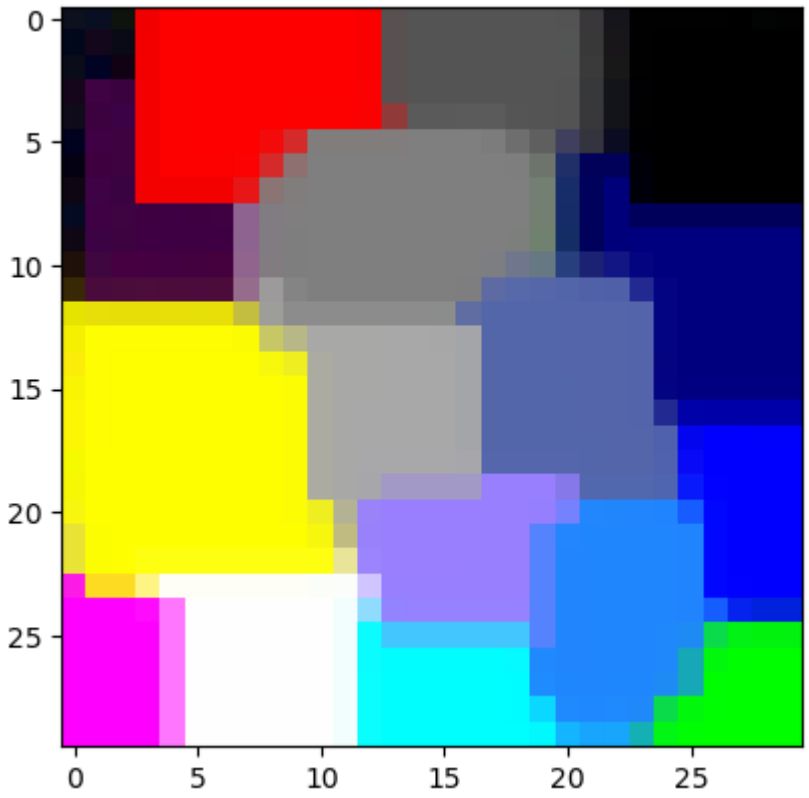
Description: A Kohonen Self-Organizing Map (SOM) is an unsupervised neural network that transforms high-dimensional data into a lower-dimensional (typically two-dimensional) grid. It organizes input data based on similarity, allowing for effective visualization and clustering. Each node in the grid represents a prototype, with nearby nodes capturing similar data patterns. The learning process involves adjusting the weights of these nodes based on input, leading to a topology-preserving mapping. SOMs are widely used in applications like pattern recognition, data compression, and exploratory data analysis.

Code:

```
from minisom import MiniSom
import numpy as np
import matplotlib.pyplot as plt

colors = [
    [0.0, 0.0, 0.0],
    [0.0, 0.0, 1.0],
    [0.0, 0.0, 0.5],
    [0.125, 0.529, 1.0],
    [0.33, 0.4, 0.67],
    [0.6, 0.5, 1.0],
    [0.0, 1.0, 0.0],
    [1.0, 0.0, 0.0],
    [0.0, 1.0, 1.0],
    [1.0, 0.0, 1.0],
    [1.0, 1.0, 0.0],
    [1.0, 1.0, 1.0],
    [0.33, 0.33, 0.33],
    [0.5, 0.5, 0.5],
    [0.66, 0.66, 0.66],
]

color_names = [
    "black",
    "blue",
    "darkblue",
    "skyblue",
    "greyblue",
    "lilac",
    "green",
    "red",
    "cyan",
    "violet",
    "yellow",
    "white",
```

```
    "darkgrey",
    "mediumgrey",
    "lightgrey",
]

som = MiniSom(30, 30, 3, sigma=3.0, learning_rate=2.5,
neighborhood_function="gaussian")

plt.imshow(abs(som.get_weights()), interpolation="none")
som = MiniSom(30, 30, 3, sigma=8.0, learning_rate=0.5,
neighborhood_function="bubble")

som.train_random(colors, 500, verbose=True)
plt.imshow(abs(som.get_weights()), interpolation="none")
```

Output:



Conclusion/Learnings: In conclusion, Kohonen Self-Organizing Maps offer a powerful tool for visualizing and analyzing complex, high-dimensional data. Their ability to preserve the topological structure of input data makes them ideal for clustering and pattern recognition tasks. By facilitating unsupervised learning, SOMs enable insights into underlying data distributions without requiring labeled inputs. They find applications across various fields, including finance, biology, and image processing. Overall, SOMs contribute significantly to data exploration and understanding in an increasingly data-driven world.

6B) Aim: Adaptive resonance theory

Description: Adaptive Resonance Theory (ART) is a type of neural network designed for unsupervised learning, particularly in situations requiring stability and plasticity. It maintains a balance between learning new information and preserving previously learned knowledge, allowing for continual adaptation without catastrophic forgetting. ART networks categorize input patterns into clusters, adjusting their weights dynamically based on input similarity. The model employs a vigilance parameter to control the granularity of categorization, enabling it to respond effectively to varying data distributions. This makes ART suitable for applications in pattern recognition, speech processing, and more

Code:

```
import numpy as np

class ART:
    def __init__(self, input_size, vigilance=0.5):
        self.input_size = input_size
        self.vigilance = vigilance
        self.weights = []
        self.categories = 0

    def _normalize(self, x):
        return x / np.linalg.norm(x)

    def _find_matching_category(self, input_pattern):
        for i, w in enumerate(self.weights):
            if np.dot(self._normalize(w), self._normalize(input_pattern)) >=
self.vigilance:
                return i
        return -1

    def learn(self, input_pattern):
        input_pattern = self._normalize(input_pattern)

        # Find matching category
        category = self._find_matching_category(input_pattern)

        if category == -1:
            # Create a new category
            self.weights.append(input_pattern)
            self.categories += 1
        else:
            # Update the existing category
            self.weights[category] = self._normalize(self.weights[category] +
input_pattern)

    def predict(self, input_pattern):
```

```
        input_pattern = self._normalize(input_pattern)
        category = self._find_matching_category(input_pattern)
        return category

# Example usage
if __name__ == "__main__":
    # Create an ART network for 2D input patterns
    art = ART(input_size=2, vigilance=0.5)

    # Sample input patterns
    patterns = [
        np.array([1, 0]),
        np.array([0, 1]),
        np.array([0.9, 0.1]),
        np.array([0.1, 0.9])
    ]

    # Train the ART network with the patterns
    for pattern in patterns:
        art.learn(pattern)

    # Test prediction
    test_pattern = np.array([0.95, 0.05])
    category = art.predict(test_pattern)
    print(f'The test pattern {test_pattern} belongs to category: {category}')

    # Display learned weights (categories)
    print("Learned Categories (Weights):")
    for i, w in enumerate(art.weights):
        print(f"Category {i}: {w}")
```

Output:

```
The test pattern [0.95 0.05] belongs to category: 0
Learned Categories (Weights):
Category 0: [0.99846976 0.05530039]
Category 1: [0.05530039 0.99846976]
```

Conclusion/Learnings: In conclusion, Adaptive Resonance Theory (ART) provides a robust framework for unsupervised learning, effectively balancing the need for stability and adaptability. Its ability to dynamically categorize input patterns while retaining previously learned information makes it ideal for real-time applications. The vigilance parameter enhances its flexibility, allowing for precise control over categorization granularity. ART has proven valuable in diverse fields, including cognitive modeling, robotics, and signal processing. Overall, ART contributes significantly to our understanding of neural computation and adaptive learning processes.

Practical No: - 7

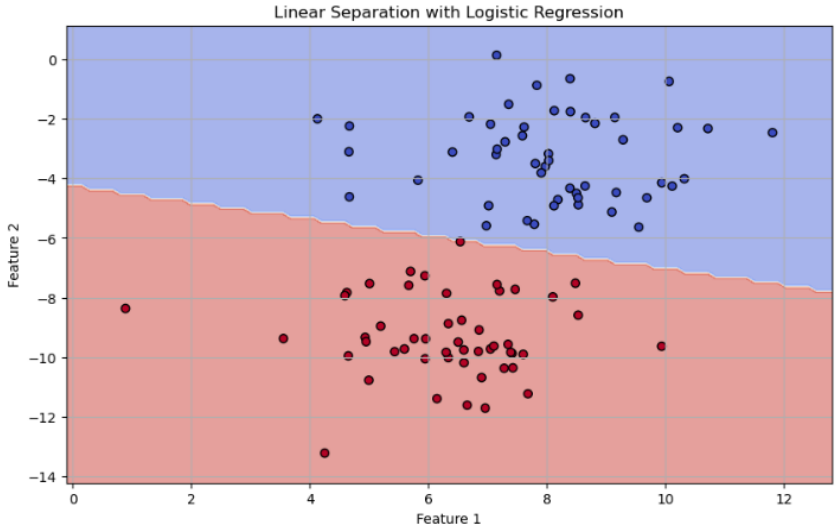
7A) Aim: Write a program for Linear seperation

Description: Linear separation refers to the ability to divide a dataset into distinct classes using a straight line (or hyperplane in higher dimensions). In a two-dimensional space, this means finding a line that separates points of one class from another. This concept is fundamental in machine learning, particularly in algorithms like Support Vector Machines (SVM). A dataset is considered linearly separable if such a line can be drawn without misclassifying any points. When classes are not linearly separable, more complex methods or transformations may be needed.

Code:

```
import numpy as np

import matplotlib.pyplot as plt
from sklearn.linear_model import LogisticRegression
from sklearn.datasets import make_blobs

# Generate a synthetic dataset
X, y = make_blobs(n_samples=100, centers=2, random_state=6, cluster_std=1.5)

# Create a logistic regression model
model = LogisticRegression()

# Fit the model
model.fit(X, y)

# Create a grid to plot decision boundary
xx, yy = np.meshgrid(np.linspace(X[:, 0].min() - 1, X[:, 0].max() + 1, 100),
                     np.linspace(X[:, 1].min() - 1, X[:, 1].max() + 1, 100))

# Predict the class for each point in the grid
Z = model.predict(np.c_[xx.ravel(), yy.ravel()])
Z = Z.reshape(xx.shape)

# Plotting
plt.figure(figsize=(10, 6))
plt.contourf(xx, yy, Z, alpha=0.5, cmap='coolwarm')  # Decision boundary
plt.scatter(X[:, 0], X[:, 1], c=y, edgecolors='k', cmap='coolwarm')  # Data
points
plt.title('Linear Separation with Logistic Regression')
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.xlim(xx.min(), xx.max())
plt.ylim(yy.min(), yy.max())
plt.grid()
plt.show()
```

Output:



Conclusion/Learnings: In summary, linear separation is essential for classifying data into distinct categories using a straight line or hyperplane. When datasets are linearly separable, it simplifies modeling and enhances interpretability with algorithms like Support Vector Machines. Conversely, non-separable data requires more complex techniques to achieve accurate classification. Recognizing the separability of data informs model selection and impacts performance. Thus, understanding linear separation is vital in the field of machine learning.

7B) Aim: Write a program for Hopfield network model for associative memory.

Description: The Hopfield network is a recurrent neural network designed for associative memory, allowing it to store and retrieve patterns. It consists of binary neurons that are fully interconnected, with each neuron influencing the others through weighted connections. When presented with a partial or noisy input, the network converges to the closest stored pattern, effectively recalling the complete memory. The Hopfield network operates on the principle of energy minimization, where stable states correspond to stored memories. This model is particularly useful for solving optimization problems and pattern recognition tasks.

Code:

```
import numpy as np

class HopfieldNetwork:
    def __init__(self, size):
        self.size = size
        self.weights = np.zeros((size, size))

    def train(self, patterns):
        """Train the Hopfield network with given patterns."""
        for pattern in patterns:
            # Reshape to column vector
```

```
            pattern = pattern.reshape(-1, 1)
            self.weights += np.dot(pattern, pattern.T)
         # Set diagonal to zero (no self-connections)
        np.fill_diagonal(self.weights, 0)

    def recall(self, input_pattern, steps=5):
        """Recall a pattern from the network."""
        state = input_pattern.copy()
        for _ in range(steps):
            for i in range(self.size):
                # Calculate the activation (sum of weighted inputs)
                activation = np.dot(self.weights[i], state)
                # Update the neuron based on the sign of the activation
                state[i] = 1 if activation > 0 else -1
        return state

# Example usage
if __name__ == "__main__":
    # Define patterns to store (in binary format)
    patterns = np.array([[1, 1, -1, -1],
                         [1, -1, 1, -1],
                         [-1, 1, -1, 1]])

    # Create a Hopfield network
    hopfield_net = HopfieldNetwork(size=4)

    # Train the network with the patterns
    hopfield_net.train(patterns)

    # Test the network with a noisy input pattern
    test_pattern = np.array([1, 1, -1, 1])  # Noisy version of the first
pattern
    recalled_pattern = hopfield_net.recall(test_pattern)

    print("Test Pattern:   ", test_pattern)
    print("Recalled Pattern:", recalled_pattern)
```

Output:

```
Test Pattern:    [ 1  1 -1  1]
Recalled Pattern: [-1  1 -1  1]
```

Conclusion/Learnings: In conclusion, the Hopfield network is a powerful model for associative memory, capable of recalling patterns from partial or noisy inputs. Its structure of fully interconnected binary neurons allows for robust memory retrieval through energy minimization. While effective for pattern recognition and optimization tasks, the network has limitations, such as a finite memory capacity and potential for spurious states. Despite these challenges, Hopfield networks remain significant in neural network research and applications. Overall, they illustrate the principles of associative memory in a computational framework

Practical:- 8

8 A) Aim: Membership and Identity Operators | in, not in

Description:

in:

3. The in operator checks if a specified element exists within a collection, such as a list, tuple, set, or string.
4. It returns True if the element is found; otherwise, it returns False.
5. **Usage**:
    1. This operator is particularly useful for quickly verifying membership without needing to iterate through the collection.

not in:

6. The not in operator checks if a specified element is absent from a collection.
7. It returns True if the element is not found; otherwise, it returns False.
8. **Usage**:
    1. This operator allows you to easily determine if an element should be added to a collection or processed further.

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")
    list1 = []
c = int(input("Enter the number of elements that you want to insert in List 1:"))
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 1:"))
if a not in list1:
    print("The list does not contain ", a)
else:
    print("The list contains", a)
```

OUTPUT:

```
Enter 5 numbers
1
6
5
3
4
Enter 5 numbers
6
9
8
```

3

7

The Lists Overlap

Enter the number of elements that you want to insert in List 1:4

Enter the element :1

Enter the element :3

Enter the element :5

Enter the element :9

enter the number that you want to find in List 1:9

The list contains 9

CONCLUSION: In conclusion, the membership operators in and not in are essential for efficiently checking the presence or absence of elements within collections like lists, tuples, sets, and strings. They enhance code readability and simplify membership tests.

8 B) Aim: Membership and Identity Operators is, is not.

Description:

`is`:

9. The `is` operator checks whether two variables refer to the same object in memory.
10. It returns `True` if both variables point to the exact same object; otherwise, it returns `False`.
11. **Usage**:
    1. This operator is useful for comparing objects, especially when dealing with mutable types like lists or dictionaries.
12.

`is not`:

1. The `is not` operator checks whether two variables do not refer to the same object in memory.
2. It returns `True` if the variables point to different objects; otherwise, it returns `False`.
3. **Usage**:
    1. This operator is helpful when you need to ensure that two references are distinct.

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()
    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()
```

OUTPUT:

```
Enter your name : Jeni
Enter your exact age : 20
Enter your roll no : 20
Jeni
Int =  False
Float =  False
String =  True
20.0
Int =  False
Float =  True
String =  False
20
Int =  True
Float =  False
String =  False
```

CONCLUSION: the identity operators is and is not are crucial for determining whether two variables point to the same object in memory. This distinction is particularly important when working with mutable types, as it can affect how data is manipulated and compared.

Practical No: - 9

9A) Aim: Find ratios using fuzzy logic.

Description: This code demonstrates the use of **fuzzy logic** to calculate a ratio between two variables, A and B. It defines fuzzy sets for both variables (small, medium, and large) using triangular membership functions. The ratio is computed by dividing the maximum membership degree of A by that of B. The code also visualizes the fuzzy sets and computes the fuzzy ratio for specific values of A and B.

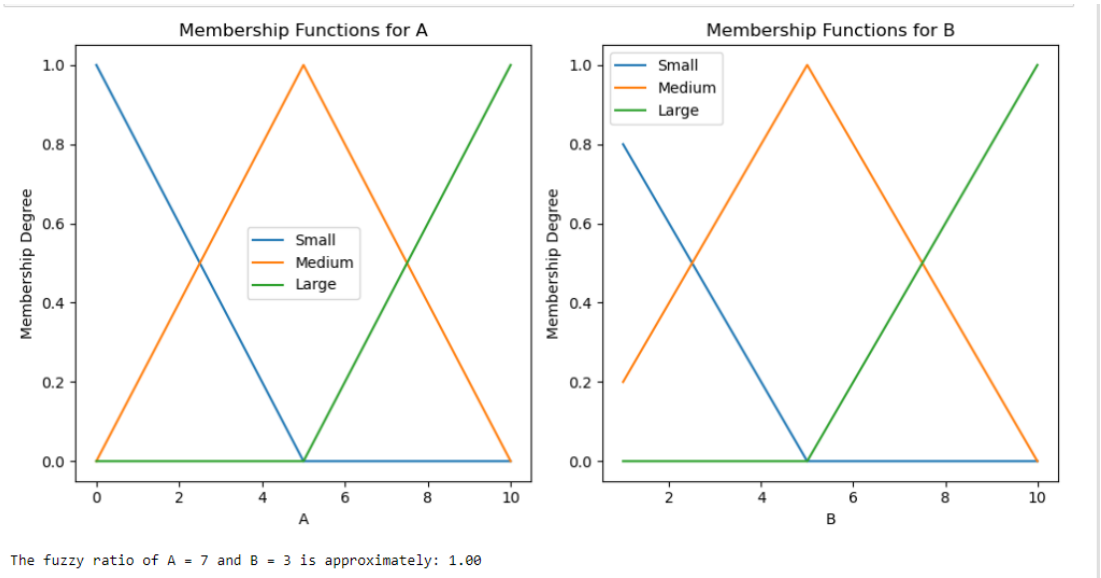
Code:

```
import numpy as np

import skfuzzy as fuzz

import matplotlib.pyplot as plt


# Define fuzzy variables for A and B

A = np.arange(0, 11, 1)  # A can range from 0 to 10

B = np.arange(1, 11, 1)  # B can range from 1 to 10 (avoiding division by zero)


# Define membership functions for A and B

A_small = fuzz.trimf(A, [0, 0, 5])

A_medium = fuzz.trimf(A, [0, 5, 10])

A_large = fuzz.trimf(A, [5, 10, 10])

B_small = fuzz.trimf(B, [0, 0, 5])

B_medium = fuzz.trimf(B, [0, 5, 10])

B_large = fuzz.trimf(B, [5, 10, 10])


# Plot the fuzzy sets for A and B

plt.figure(figsize=(10, 5))

plt.subplot(1, 2, 1)

plt.plot(A, A_small, label="Small")

plt.plot(A, A_medium, label="Medium")
```

```
plt.plot(A, A_large, label="Large")
plt.title("Membership Functions for A")
plt.xlabel("A")
plt.ylabel("Membership Degree")
plt.legend()
plt.subplot(1, 2, 2)
plt.plot(B, B_small, label="Small")
plt.plot(B, B_medium, label="Medium")
plt.plot(B, B_large, label="Large")
plt.title("Membership Functions for B")
plt.xlabel("B")
plt.ylabel("Membership Degree")
plt.legend()
plt.tight_layout()
plt.show()


# Fuzzy division (calculate ratio) using a simple membership function approach
def fuzzy_divide(A_value, B_value):
    # For simplicity, we assume the degree of membership for A and B are given directly
    A_membership = [fuzz.interp_membership(A, A_small, A_value),
                    fuzz.interp_membership(A, A_medium, A_value),
                    fuzz.interp_membership(A, A_large, A_value)]

    B_membership = [fuzz.interp_membership(B, B_small, B_value),
                    fuzz.interp_membership(B, B_medium, B_value),
                    fuzz.interp_membership(B, B_large, B_value)]
```

```
 # Fuzzy division operation (we divide the max membership degree)
    ratio = np.max(A_membership) / np.max(B_membership)
      return ratio


# Test with some values of A and B
A_value = 7  # Example value for A
B_value = 3  # Example value for B

ratio_result = fuzzy_divide(A_value, B_value)

print(f"The fuzzy ratio of A = {A_value} and B = {B_value} is approximately: {ratio_result:.2f}")
```

Output:



Conclusion/Learnings:

• **Fuzzy Logic** allows us to handle imprecise or uncertain data, which traditional crisp logic cannot manage.
• **Membership Functions** express the degree to which a value belongs to a fuzzy set, enabling us to model vagueness (e.g., "somewhat large" instead of just "large").

 Fuzzy Arithmetic (like division) involves using membership degrees rather than exact values, providing a more flexible way to compute ratios or other operations in uncertain conditions.

 Visualization helps to understand how values are distributed across fuzzy sets and how fuzzy operations work.
 Practical Applications: Fuzzy logic is widely used in control systems and decision-making where precise inputs are unavailable, such as temperature control or automated decision-making.

9B) Aim: Solve Tipping problem using fuzzy logic.

Description: The Tipping Problem is solved using **fuzzy logic** to determine the tip amount based on **Service Quality** and **Meal Quality**, each rated on a scale. Both inputs are fuzzified into categories like "Poor", "Average", and "Good". The tip amount is similarly fuzzified into "Low", "Medium", and "High". A set of fuzzy rules is used to relate the inputs to the output, and fuzzy inference is applied to determine the tip. Finally, **defuzzification** converts the fuzzy result into a crisp tip value using the **Centroid Method**.

Code:

```
import numpy as np

import skfuzzy as fuzz

import matplotlib.pyplot as plt


# Define the input ranges (Service and Meal Quality)

service_quality = np.arange(0, 11, 1)  # 0 to 10

meal_quality = np.arange(0, 11, 1)  # 0 to 10


# Define the output range (Tip Amount in dollars)

tip_amount = np.arange(0, 26, 1)  # $0 to $25


# Define fuzzy membership functions for service quality

service_poor = fuzz.trimf(service_quality, [0, 0, 5])

service_average = fuzz.trimf(service_quality, [0, 5, 10])

service_good = fuzz.trimf(service_quality, [5, 10, 10])


# Define fuzzy membership functions for meal quality

meal_poor = fuzz.trimf(meal_quality, [0, 0, 5])

meal_average = fuzz.trimf(meal_quality, [0, 5, 10])
```

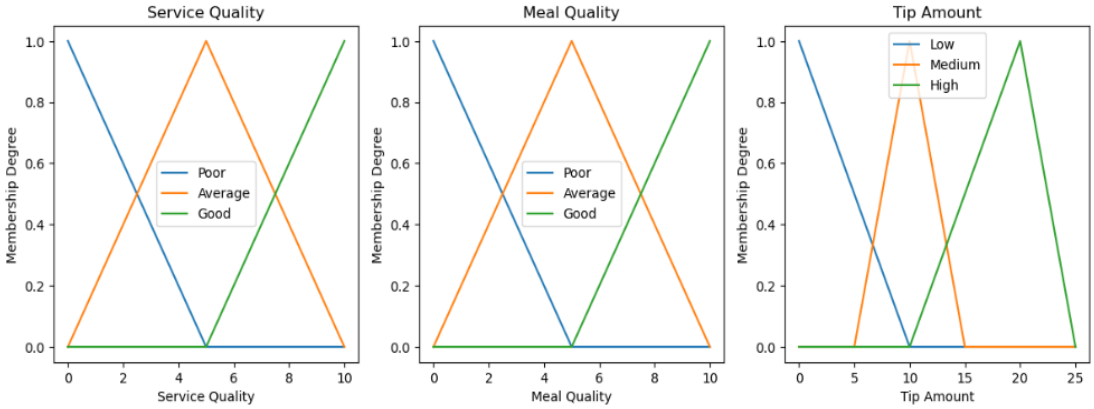
```
meal_good = fuzz.trimf(meal_quality, [5, 10, 10])

# Define fuzzy membership functions for tip amount
tip_low = fuzz.trimf(tip_amount, [0, 0, 10])
tip_medium = fuzz.trimf(tip_amount, [5, 10, 15])
tip_high = fuzz.trimf(tip_amount, [10, 20, 25])

# Plot membership functions
plt.figure(figsize=(12, 8))

# Plot service quality membership functions
plt.subplot(2, 3, 1)
plt.plot(service_quality, service_poor, label='Poor')
plt.plot(service_quality, service_average, label='Average')
plt.plot(service_quality, service_good, label='Good')
plt.title('Service Quality')
plt.xlabel('Service Quality')
plt.ylabel('Membership Degree')
plt.legend()

# Plot meal quality membership functions
plt.subplot(2, 3, 2)
plt.plot(meal_quality, meal_poor, label='Poor')
plt.plot(meal_quality, meal_average, label='Average')
plt.plot(meal_quality, meal_good, label='Good')
plt.title('Meal Quality')
plt.xlabel('Meal Quality')
plt.ylabel('Membership Degree')
plt.legend()
```

```
# Plot tip amount membership functions
plt.subplot(2, 3, 3)
plt.plot(tip_amount, tip_low, label='Low')
plt.plot(tip_amount, tip_medium, label='Medium')
plt.plot(tip_amount, tip_high, label='High')
plt.title('Tip Amount')
plt.xlabel('Tip Amount')
plt.ylabel('Membership Degree')
plt.legend()

plt.tight_layout()
plt.show()

# Fuzzy logic inference system
def calculate_tip(service_val, meal_val):
    # Fuzzification: calculate the membership degrees for the inputs
    service_degree_poor = fuzz.interp_membership(service_quality, service_poor, service_val)
    service_degree_average = fuzz.interp_membership(service_quality, service_average, service_val)
    service_degree_good = fuzz.interp_membership(service_quality, service_good, service_val)

    meal_degree_poor = fuzz.interp_membership(meal_quality, meal_poor, meal_val)
    meal_degree_average = fuzz.interp_membership(meal_quality, meal_average, meal_val)
    meal_degree_good = fuzz.interp_membership(meal_quality, meal_good, meal_val)

    # Rule Base
    # If Service is Good and Meal is Good, Tip is High

    rule1 = np.minimum(service_degree_good, meal_degree_good)
```

```
    rule2 = np.minimum(service_degree_average, meal_degree_good)
    rule3 = np.minimum(service_degree_poor, meal_degree_good)

    # Apply fuzzy AND to rules and combine results for the tip (use OR combination)
    tip_low_activation = np.maximum(np.minimum(service_degree_poor, meal_degree_poor),
                                    np.minimum(service_degree_poor, meal_degree_average))
    tip_medium_activation = np.maximum(np.minimum(service_degree_average, meal_degree_poor),
                                       np.minimum(service_degree_good, meal_degree_average))
    tip_high_activation = np.maximum(np.minimum(service_degree_good, meal_degree_good), rule1)

    # Fuzzy inference - combining all the rules to calculate the fuzzy tip result
    tip_activation_low = np.minimum(tip_low_activation, tip_low)
    tip_activation_medium = np.minimum(tip_medium_activation, tip_medium)
    tip_activation_high = np.minimum(tip_high_activation, tip_high)

    # Defuzzification: Compute the crisp tip value
    tip_value = fuzz.defuzz(tip_amount, np.maximum(tip_activation_low,
                                                   np.maximum(tip_activation_medium, tip_activation_high)),
'centroid')

    return tip_value

# Test the fuzzy tipping system with example inputs
service_input = 8  # Example: Good service
meal_input = 7     # Example: Good meal

tip_result = calculate_tip(service_input, meal_input)
```

```
print(f"With service rating {service_input} and meal rating {meal_input}, the tip is: ${tip_result:.2f}")
```

Output:




```
With service rating 8 and meal rating 7, the tip is: $14.39
```

Conclusion/Learnings:

- **Fuzzy logic** allows us to model **uncertainty** and **subjectivity**, providing a flexible way to make decisions based on vague or imprecise inputs (like ratings).
- It's useful in real-world applications where values are not strictly numeric or are inherently uncertain.
- **Fuzzification** and **defuzzification** provide a way to handle and compute with such uncertainty, converting fuzzy results into actionable outputs.

Practical:-10

10A) Aim: Implementation of Simple genetic algorithm

Description: A Simple Genetic Algorithm (SGA) is an optimization technique that simulates natural selection. It begins with a randomly generated population of potential solutions (chromosomes). Each chromosome is evaluated using a fitness function to determine its effectiveness. The fittest individuals are selected as parents, and crossover is performed to create offspring by combining parent traits. Mutation introduces random changes to some offspring to enhance diversity. The new generation replaces part or all of the old population.This process iterates until a termination condition is met, such as a maximum number of generations or achieving a satisfactory fitness level. SGA is useful for solving complex problems where traditional methods may struggle.

Code:

```
import random

POPULATION_SIZE = 100

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

# Target string to be generated

TARGET = "I love GeeksforGeeks"

 class Individual(object):

 '''

  Class representing individual in population

   '''

   def __init__(self, chromosome):

      self.chromosome = chromosome

      self.fitness = self.cal_fitness()

   @classmethod

   def mutated_genes(cls):

      '''

      Create random genes for mutation

      '''
```

```
    global GENES
    gene = random.choice(GENES)
    return gene
@classmethod
def create_gnome(cls):
    '''
    Create chromosome or string of genes
    '''
    global TARGET
    gnome_len = len(TARGET)
    return [cls.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 fitness 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 is 0, we know that we have reached the target
        if population[0].fitness <= 0:
            found = True
            break
        # Otherwise generate new offsprings for new generation
        new_generation = []
        # Perform Elitism, that means 10% of fittest population goes to the next generation
        s = int((10 * POPULATION_SIZE) / 100)
        new_generation.extend(population[:s])
        # From 90% of fittest population, Individuals will mate to produce offspring
        s = int((90 * POPULATION_SIZE) / 100)
        for _ in range(s):
            parent1 = random.choice(population[:50])
            parent2 = random.choice(population[:50])
            child = parent1.mate(parent2)
            new_generation.append(child)
        population = new_generation
        print("Generation: {}\tString: {}\tFitness: {}"
              .format(generation,
                      "".join(population[0].chromosome),
                      population[0].fitness))
        generation += 1
```

```
    print("Generation: {}\tString: {}\tFitness: {}"
          .format(generation,
                  "".join(population[0].chromosome),
                  population[0].fitness))
if __name__ == '__main__':
    main()
```

Output:

```
Generation: 1  String: !wWmT9ZgEhLHH_.L_Uko Fitness: 19
Generation: 2  String: {JvKfD5TebURz5r@OCY{ Fitness: 18
Generation: 3  String: {JvKfD5TebURz5r@OCY{ Fitness: 18
Generation: 4  String: q pk3s  keHlrr X:eo] Fitness: 16
Generation: 5  String: 5Zl=WD%Jkb-;torFeeVt Fitness: 15
Generation: 6  String: 5Zl=WD%Jkb-;torFeeVt Fitness: 15
Generation: 7  String: j :k1e SeeUs07!ZRas- Fitness: 14
Generation: 8  String: 5 l Xk5Jxe-;torOeeVt Fitness: 13
Generation: 9  String: 5 l Xk5Jxe-;torOeeVt Fitness: 13
Generation: 10 String: j =kXe Seepsyo4MeeKb Fitness: 11
Generation: 11 String: j =kXe Seepsyo4MeeKb Fitness: 11
Generation: 12 String: j lY$e  eeos0orZ0est Fitness: 10
Generation: 13 String: j lY$e  eeos0orZ0est Fitness: 10
Generation: 14 String: I lkv9 Deeqsyord6eM- Fitness: 9
Generation: 15 String: I lkve  eebsvorc6esb Fitness: 8
Generation: 16 String: I lkve  eebsvorc6esb Fitness: 8
Generation: 17 String: I lkve  eebsvorc6esb Fitness: 8
Generation: 18 String: 5 lkve neeksvordeeVo Fitness: 7
Generation: 19 String: 5 lkve neeksvordeeVo Fitness: 7
Generation: 20 String: 5 lkve neeksvordeeVo Fitness: 7
Generation: 21 String: 5 lkve neeksvordeeVo Fitness: 7
Generation: 22 String: I lkve Deeks=orZeeC- Fitness: 6
Generation: 23 String: I lkve Deeks=orZeeC- Fitness: 6
Generation: 24 String: I lkve Deeks=orZeeC- Fitness: 6
Generation: 25 String: I lkve Deeks=orZeeC- Fitness: 6
Generation: 26 String: I lkve Deeks=orZeeC- Fitness: 6
Generation: 27 String: I lkve Deeks=orZeeC- Fitness: 6
Generation: 28 String: I lkve Deeks=orZeeC- Fitness: 6
Generation: 29 String: I lkve Deeks=orZeeC- Fitness: 6
Generation: 30 String: I lkve Deeks=orZeeC- Fitness: 6
Generation: 31 String: I lkve Deeks=orZeeC- Fitness: 6
Generation: 32 String: I lyve beeks7orZeeku Fitness: 5
Generation: 33 String: I lyve beeks7orZeeku Fitness: 5
Generation: 34 String: I lyve beeks7orZeeku Fitness: 5
Generation: 35 String: I lyve beeks7orZeeku Fitness: 5
Generation: 36 String: I lyve beeks7orZeeku Fitness: 5
Generation: 37 String: I lyve beeks7orZeeku Fitness: 5
Generation: 38 String: I lyve beeks7orZeeku Fitness: 5
Generation: 39 String: I lyve beeks7orZeeku Fitness: 5
Generation: 40 String: I lkve Geeksvordeek- Fitness: 4
Generation: 41 String: I lkve Geeksvordeek- Fitness: 4
Generation: 42 String: I lkve Geeksvordeek- Fitness: 4
```

Conclusion/Learnings: In conclusion, the Simple Genetic Algorithm (SGA) is a powerful optimization technique inspired by the principles of natural selection. By iteratively evolving a population of potential solutions through selection, crossover, and mutation, SGA effectively explores complex solution spaces. Its ability to find near-optimal solutions makes it valuable for a wide range of applications, from engineering to artificial intelligence. While it requires careful tuning of parameters and may face challenges like premature convergence, its flexibility and robustness make it a popular choice for tackling optimization problems in diverse fields.

10B) Aim: Create two classes: City and Fitness using Genetic Algorithm.

Description:

City Class

The **City** class represents a city in the optimization problem. It includes attributes such as the city's name and its coordinates (x, y). The class provides a method to calculate the Euclidean distance to another city, enabling the evaluation of routes.

Key Attributes:

4. **name**: Identifier for the city.
5. **x**: X-coordinate.
6. **y**: Y-coordinate.

Key Method:

7. **distance(other_city)**: Computes the distance to another city.

Fitness Class

The **Fitness** class evaluates a route formed by a sequence of cities. It calculates the total distance traveled along the route, serving as a fitness score for optimization purposes. The goal is to minimize this distance in problems like the Traveling Salesman Problem.

Key Attributes:

8. **route**: List of City objects representing the order of cities.
9. **distance**: Total distance of the route.

Key Method:

10. **calculate_fitness()**: Computes and returns the total distance of the route.

Code:

```
#city Class
import random
class City:
    def __init__(self, name, x, y):
        self.name = name
        self.x = x
        self.y = y
  def distance_to(self, other):
        return ((self.x - other.x) ** 2 + (self.y - other.y) ** 2) ** 0.5
def __repr__(self):
        return f"{self.name}({self.x}, {self.y})"
#Fitness Class
class Fitness:
    def __init__(self, route):
        self.route = route
        self.distance = self.calculate_total_distance()
 def calculate_total_distance(self):
        total_distance = 0
        for i in range(len(self.route)):
            from_city = self.route[i]
            to_city = self.route[(i + 1) % len(self.route)]  # Wrap around to the start
            total_distance += from_city.distance_to(to_city)
        return total_distance
 def get_fitness(self):
        return 1 / self.distance  # Higher fitness for shorter routes
```

```
# Create cities

city_a = City("A", 0, 0)

city_b = City("B", 1, 2)

city_c = City("C", 3, 1)

city_d = City("D", 4, 0)

# Create a route (list of cities)

route = [city_a, city_b, city_c, city_d]

# Calculate fitness

fitness = Fitness(route)

print(f"Route: {route}")

print(f"Total Distance: {fitness.distance}")

print(f"Fitness: {fitness.get_fitness()}")
```

Output:

```
Route: [A(0, 0), B(1, 2), C(3, 1), D(4, 0)]
Total Distance: 9.886349517372675
Fitness: 0.10114956974187099
```

Conclusion/Learnings: In summary, the City and Fitness classes form essential components of a G enetic Algorithm for optimization problems like the Traveling Salesman Problem. The City class enca psulates the properties of individual cities, while the Fitness class evaluates routes based on their total distance. Together, they facilitate the process of exploring and optimizing solutions, enabling effectiv e use of genetic algorithms in complex problem-solving scenarios.