

ANNAMALAI UNIVERSITY

FACULTY OF ENGINEERING AND TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

B.E. COMPUTER SCIENCE AND ENGINEERING

(ARTIFICIAL INTELLIGENCE AND MACHINE LEARNING)

V – SEMESTER

22AICP510 - NEURAL COMPUTING LAB

Name: _____

Reg No: _____

ANNAMALAI UNIVERSITY

FACULTY OF ENGINEERING AND TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

B.E. COMPUTER SCIENCE AND ENGINEERING

(ARTIFICIAL INTELLIGENCE AND MACHINE LEARNING)

V – SEMESTER

22AICP510 - NEURAL COMPUTING LAB

Certified that this is the Bonafide Record of work done by
Mr./Ms_____ **Reg. No.**_____ of
V semester **B.E. Computer Science and Engineering (Artificial Intelligence
and Machine Learning)** in the **22AICP510 - Neural Computing Laboratory**
during the odd semester (**June2024 – Nov2024**).

Staff Incharge

External Examiner

Internal Examiner

Place: Annamalai Nagar

Date: / / 2024

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

VISION

To provide a congenial ambience for individuals to develop and blossom as academically superior, socially conscious and nationally responsible citizens.

MISSION

M1: Impart high quality computer knowledge to the students through a dynamic scholastic environment wherein they learn to develop technical, communication and leadership skills to bloom as a versatile professional.

M2: Develop life-long learning ability that allows them to be adaptive and responsive to the changes in career, society, technology, and environment

M3: Build student community with high ethical standards to undertake innovative research and development in thrust areas of national and international needs.

M4: Expose the students to the emerging technological advancements for meeting the demands of the industry.

PROGRAMME EDUCATIONAL OBJECTIVES (PEO)

PEO	PEO STATEMENTS
PEO1	To prepare graduates with potential to get employed in the right role and/or become entrepreneurs to contribute to the society.
PEO2	To provide the graduates with the requisite knowledge to pursue higher education and carry out research in the field of Computer Science and Engineering.
PEO3	To equip the graduates with skills required to stay motivated and adapt to dynamically changing world so as to remain successful in their career.
PEO4	To train the graduates to communicate effectively, work collaboratively and exhibit levels of professionalism and ethical responsibility.

PROGRAMME SPECIFIC OUTCOMES

PSO1: Ability to investigate challenging problems in various domains and exhibit programming skills to build automation solutions.

PSO2: Ability to apply the algorithms and computational techniques to construct robust and resilient computer systems and applications.

PSO3: Ability to comprehend and implement the contemporary trends in industry and research environment paving the way for innovative solutions to existing and emerging problems.

COURSE OBJECTIVES

- To provide the most comprehensive concept of neural networks in the engineering perspective
- To understand the important design concepts of neural architectures in different applications
- To acquaint student with various computing algorithms in FLNN using software tools.
- To understand operation of basic elements in fuzzy logic and neural network through simulation.

LIST OF EXERCISES

1. Write a program to implement the concept of Perceptron.
2. Write a program to simulate Back-Propagation Neural Networks.
3. Write a program to implement the Radial Basis Function Neural Networks.
4. Write a program to implement a real-world application using Support Vector Machine.
5. Write a program to implement a Self-Organizing Map and a Learning Vector Quantization for an application.
6. Write a program to implement fuzzy set operations.
7. Write a program to implement fuzzy set properties.
8. Write a program to implement fuzzy membership functions.
9. Write a program to develop a fuzzy inference system.
10. Write a program to implement defuzzification methods using fuzzy toolbox.
11. Write a program to develop an application using Fuzzy logic.

COURSE OUTCOMES

At the end of this course, the students will be able to

1. Demonstrate basic concepts fuzzy logic and neural network through simulation.
2. Develop the logic given in problem statement using operations in fuzzy logic and basics of toolbox studied.
3. Demonstrate an ability to listen and answer viva questions related to programming skills for solving real world problems in Computer Science and Engineering.

MAPPING OF COURSE OUTCOMES WITH PROGRAMME OUTCOMES

	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12
CO1	2	3	3	3	2	-	-	-	-	-	-	-
CO2	2	3	2	2	2	-	-	-	-	-	-	-
CO3	2	2	-	-	-	-	-	-	-	-	-	2

Rubric for CO3 in Laboratory Courses				
Rubric	Distribution of 10 Marks for CIE/SEE Evaluation Out of 40/60 Marks			
	Up To 2.5 Marks	Up To 5 Marks	Up To 7.5 Marks	Up To 10 marks
Demonstrate an ability to listen and answer the viva questions related to programming skills needed for solving real-world problems in Computer Science and Engineering.	Poor listening and communication skills. Failed to relate the programming skills needed for solving the problem.	Showed better communication skill by relating the problem with the programming skills acquired but the description showed serious errors.	Demonstrated good communication skills by relating the problem with the programming skills acquired with few errors.	Demonstrated excellent communication skills by relating the problem with the programming skills acquired and have been successful in tailoring the description.

Table of Contents

S. No.	Date	Title	Page	Mark	Signature
1.		a) Perceptron Algorithm on Randomly Generated Dataset b) Perceptron Algorithm for OR Gate	1		
2.		a) Back Propagation Neural Network b) BPNN for OR Gate	8		
3.		a) Radial Basis Function Neural Network b) RBFNN for XOR Gate	13		
4.		a) Support Vector Machine on Ads Dataset b) Support Vector Machine for XOR Gate	18		
5.		a) Self-Organizing Map b) Learning Vector Quantization	24		
6.		Fuzzy Set Operations	27		
7.		Fuzzy Set Properties	30		
8.		Fuzzy Membership Functions	36		
9.		Fuzzy Inference System	38		
10.		Defuzzification Methods	43		
11.		Fuzzy Controller System	46		

Ex. No. 1(a) Perceptron Algorithm on Randomly Generated Dataset

Date:

Aim:

To implement perceptron algorithm using Python programming.

Perceptron Learning:

The Perceptron algorithm is a two-class (binary) classification machine learning algorithm. It is the simplest type of neural network model. It consists of a single node or neuron that takes a row of data as input and predicts a class label. This is achieved by calculating the weighted sum of the inputs and a bias (set to 1). The weighted sum of the input of the model is called the activation given as

$$\text{Activation} = \text{Weights} * \text{Inputs} + \text{Bias}$$

If the activation is above 0.0, the model will output 1.0; otherwise, it will output 0.0 given as:

Predict 1: If Activation > 0.0

Predict 0: If Activation <= 0.0

The Perceptron is a linear classification algorithm. This means that it learns a decision boundary that separates two classes using a line (called a hyperplane) in the feature space. The coefficients of the model are referred to as input weights and are trained using the stochastic gradient descent optimization algorithm.

Examples from the training dataset are shown to the model one at a time, the model makes a prediction, and error is calculated. The weights of the model are then updated to reduce the errors for the example. This is called the Perceptron update rule. This process is repeated for all examples in the training dataset, called an epoch. This process of updating the model using examples is then repeated for many epochs.

Model weights are updated with a small proportion of the error each batch, and the proportion is controlled by a hyperparameter called the learning rate, typically set to a small value as shown below.

$$\text{weights}(t + 1) = \text{weights}(t) + \text{learning_rate} * (\text{expected_i} - \text{predicted_}) * \text{input_i}$$

Training is stopped when the error made by the model falls to a low level or no longer improves, or a maximum number of epochs is performed. The initial values for the model weights are set to small random values.

Algorithm:

1. Set a threshold value
2. Multiply all inputs with their weights
3. Sum all the results
4. Activate the output and return true if greater than the threshold.

Program:

```
import numpy as np
from sklearn import datasets
import matplotlib.pyplot as plt
X, y = datasets.make_blobs(n_samples=150, n_features=2, centers=2, cluster_std=1.05,
                           random_state=2)

#Plotting
fig = plt.figure(figsize=(10,8))
plt.plot(X[:, 0][y == 0], X[:, 1][y == 0], 'r^')
plt.plot(X[:, 0][y == 1], X[:, 1][y == 1], 'bs')
plt.xlabel("feature 1")
plt.ylabel("feature 2")
plt.title('Random Classification Data with 2 classes')

def step_func(z):
    return 1.0 if (z > 0) else 0.0

def perceptron(X, y, lr, epochs):
    # X --> Inputs.
    # y --> labels/target.
    # lr --> learning rate.
    # epochs --> Number of iterations.
    # m-> number of training examples
    # n-> number of features
    m, n = X.shape
    # Initializing parameters(theta) to zeros.
    # +1 in n+1 for the bias term.
    theta = np.zeros((n+1,1))
    # Empty list to store how many examples were
    # misclassified at every iteration.
```



```

n_miss_list = []

# Training.
for epoch in range(epochs):
    # variable to store #misclassified.
    n_miss = 0
    # looping for every example.
    for idx, x_i in enumerate(X):
        # Insering 1 for bias, X0 = 1.
        x_i = np.insert(x_i, 0, 1).reshape(-1,1)
        # Calculating prediction/hypothesis.
        y_hat = step_func(np.dot(x_i.T, theta))
        # Updating if the example is misclassified.
        if (np.squeeze(y_hat) - y[idx]) != 0:
            theta += lr*((y[idx] - y_hat)*x_i)
            # Incrementing by 1.
            n_miss += 1
    # Appending number of misclassified examples
    # at every iteration.
    n_miss_list.append(n_miss)
return theta, n_miss_list

```

```

def plot_decision_boundary(X, theta):
    # X --> Inputs
    # theta --> parameters
    # The Line is  $y=mx+c$ 
    # So, Equate  $mx+c = \theta_0.X_0 + \theta_1.X_1 + \theta_2.X_2$ 
    # Solving we find m and c
    x1 = [min(X[:,0]), max(X[:,0])]
    m = -theta[1]/theta[2]
    c = -theta[0]/theta[2]
    x2 = m*x1 + c
    # Plotting
    fig = plt.figure(figsize=(10,8))
    plt.plot(X[:, 0][y==0], X[:, 1][y==0], "r^")

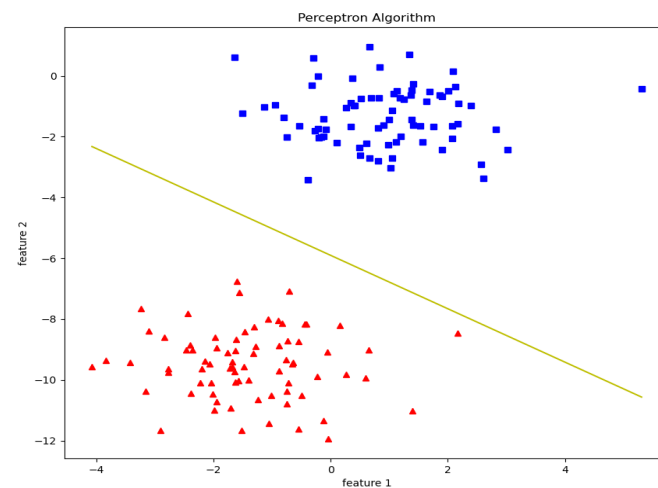
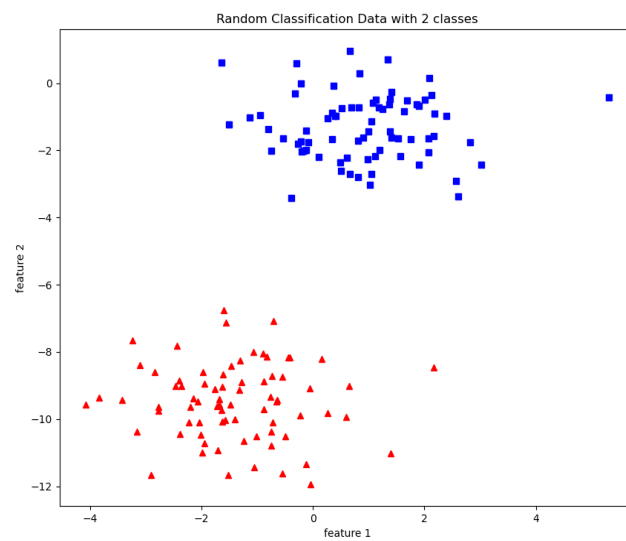
```

```

plt.plot(X[:, 0][y==1], X[:, 1][y==1], "bs")
plt.xlabel("feature 1")
plt.ylabel("feature 2")
plt.title('Perceptron Algorithm')
plt.plot(x1, x2, 'y-')
theta, miss_1 = perceptron(X, y, 0.5, 100)
plot_decision_boundary(X, theta)

```

Input and Output:



Result:

Thus a python program has been written and executed to implement perceptron algorithm.

Ex. No. 1(b) Implementation of OR Gate using Perceptron Algorithm

Date:

Aim:

To implement OR gate using perceptron algorithm in Python programming.

Program:

```
import numpy as np

class Perceptron:
    def __init__(self) -> None:
        self.weights = np.zeros((2, 1))
        self.bias = 0

    def predict(self, x: np.array) -> np.array:
        return np.where(np.dot(x, self.weights) + self.bias > 0, 1, 0)

    def fit(self, x: np.array, y: np.array, epoch:int = 100, learning_rate: float= 0.1) -> None:
        for _ in range(epoch):
            for x_i, y_i in zip(x, y):
                y_hat = self.predict(x_i).squeeze()
                self.weights += learning_rate * (y_i - y_hat) * x_i.reshape(-1, 1)
                self.bias += learning_rate * (y_i - y_hat)

    def score(self, x: np.array, y: np.array) -> float:
        y_hat = self.predict(x).squeeze()
        return np.mean(y_hat == y)

if __name__ == '__main__':
    X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
    y = np.array([0, 1, 1, 1])

    perceptron = Perceptron()
    print("OR Gate: ")
    for x, y_i in zip(X, y):
        print(x, y_i)
    print("Initial Weights:")
```

```
print(perceptron.weights, perceptron.bias)
```

```
perceptron.fit(X, y, epoch=3)
```

```
print("Trained Weights:")
```

```
print(perceptron.weights, perceptron.bias)
```

```
print("OR Gate(predicted): ")
```

```
for x in X:
```

```
    print(x, perceptron.predict(x))
```

```
print("Accuracy:", perceptron.score(X, y))
```

Input and Output:

OR Gate:

[0 0] 0

[0 1] 1

[1 0] 1

[1 1] 1

Initial Weights:

[[0.]

[0.]] 0

Trained Weights:

[[0.1]

[0.1]] 0.0

OR Gate(predicted):

[0 0] [0]

[0 1] [1]

[1 0] [1]

[1 1] [1]

Accuracy: 1.0

Result:

Thus the python program to implement OR gate with perceptron algorithm is successfully executed and the output is verified.

Ex. No. 2 (a)**BACKPROPAGATION NEURAL NETWORK**

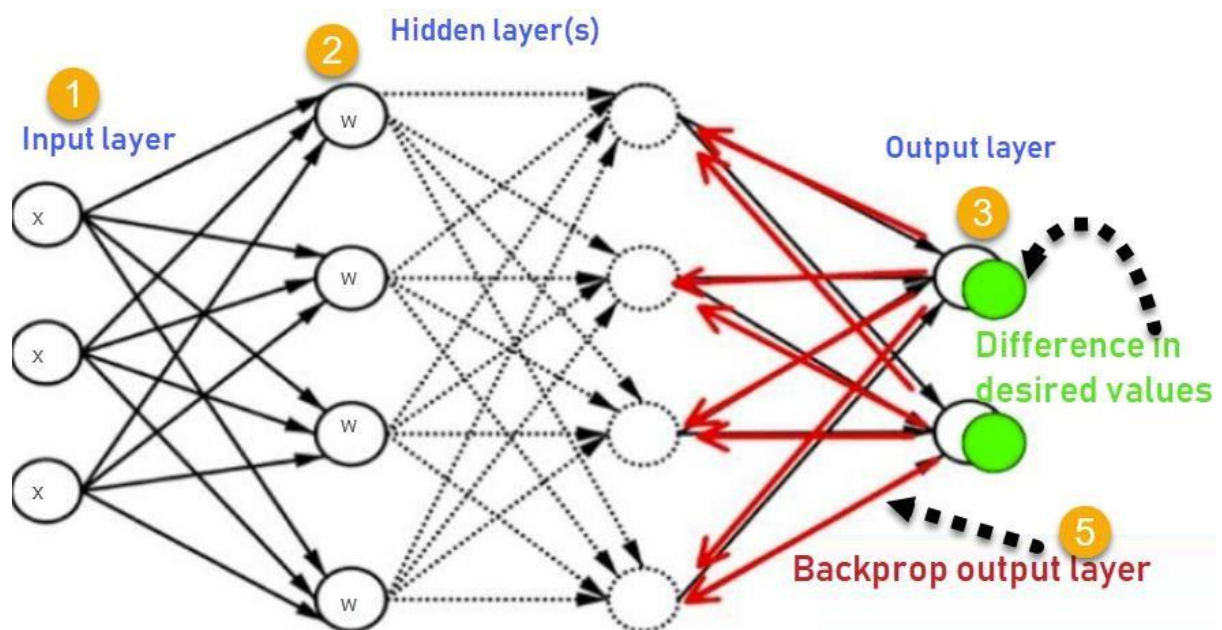
Date:

Aim:

To implement Back Propagation Neural Network using Python programming.

Algorithm:

The Back propagation algorithm in neural network computes the gradient of the loss function for a single weight by the chain rule. It efficiently computes one layer at a



time, unlike a native direct computation

Consider the following Back propagation Neural Network.

1. Inputs X , arrive through the preconnected path.
2. Input is modeled using real weights W . The weights are usually randomly selected.
3. Calculate the output for every neuron from the input layer, to the hidden layers, to the output layer.
4. Calculate the error in the outputs
$$\text{Error} = \text{Actual Output} - \text{Desired Output}$$
5. Travel back from the output layer to the hidden layer to adjust the weights such that the error is decreased.

Keep repeating the process until the desired output is achieved.

Program:

```
import numpy as np

# X = (hours sleeping, hours studying), y = score on test
X = np.array([[2, 9], [1, 5], [3, 6]], dtype=float)
y = np.array([[92], [86], [89]], dtype=float)

print(X)

print(y)

# scale units
X = X/np.amax(X, axis=0) # maximum of X array
y = y/100 # max test score is 100

print(X)

print(y)

class Neural_Network(object):

    def __init__(self):

        #parameters

        self.inputSize = 2

        self.outputSize = 1

        self.hiddenSize = 3

        #weights

        self.W1 = np.random.randn(self.inputSize, self.hiddenSize) # (3x2) weight matrix from
input to hidden layer

        self.W2 = np.random.randn(self.hiddenSize, self.outputSize) # (3x1) weight matrix from
hidden to output layer

    def forward(self, X):

        #forward propagation through our network

        self.z = np.dot(X, self.W1) # dot product of X (input) and first set of 3x2 weights

        self.z2 = self.sigmoid(self.z) # activation function
```

```
self.z3 = np.dot(self.z2, self.W2) # dot product of hidden layer (z2) and second set of
3x1 weights
```

```
o = self.sigmoid(self.z3) # final activation function
```

```
return o
```

```
def sigmoid(self, s):
```

```
    # activation function
```

```
    return 1/(1+np.exp(-s))
```

```
def sigmoidPrime(self, s):
```

```
    #derivative of sigmoid
```

```
    return s * (1 - s)
```

```
def backward(self, X, y, o):
```

```
    # backward propagate through the network
```

```
    self.o_error = y - o # error in output
```

```
    self.o_delta = self.o_error*self.sigmoidPrime(o) # applying derivative of sigmoid to error
```

```
    self.z2_error = self.o_delta.dot(self.W2.T) # z2 error: how much our hidden layer
weights contributed to output error
```

```
    self.z2_delta = self.z2_error*self.sigmoidPrime(self.z2) # applying derivative of
sigmoid to z2 error
```

```
    self.W1 += X.T.dot(self.z2_delta) # adjusting first set (input --> hidden) weights
```

```
    self.W2 += self.z2.T.dot(self.o_delta) # adjusting second set (hidden --> output) weights
```

```
def train(self, X, y):
```

```
    o = self.forward(X)
```

```
    self.backward(X, y, o)
```

```

NN = Neural_Network()

for i in range(1000): # trains the NN 1,000 times

    print ("Input: \n" + str(X))

    print ("Actual Output: \n" + str(y))

    print ("Predicted Output: \n" + str(NN.forward(X)))

    print ("Loss: \n" + str(np.mean(np.square(y - NN.forward(X)))) # mean sum squared loss

    print ("\n")

    NN.train(X, y)

```

Input and Output:

Input Data in First epoch:

```

[[ 0.66666667  1.      ]
 [ 0.33333333  0.55555556]
 [ 1.         0.66666667]]

```

Actual Output:

```

[[ 0.92]
 [ 0.86]
 [ 0.89]]

```

Predicted Output:

```

[[ 0.64293334]
 [ 0.64778178]
 [ 0.63319242]]

```

Loss:

```

0.0625842148581

```

Input in second epoch:

```

[[ 0.66666667  1.      ]
 [ 0.33333333  0.55555556]
 [ 1.         0.66666667]]

```

Actual Output:

```

[[ 0.92]
 [ 0.86]
 [ 0.89]]

```

Predicted Output:

```

[[ 0.66685628]
 [ 0.67120326]
 [ 0.65596939]]

```

Loss:

```

0.0514987603276

```

```

.....
.....
.....

```


Input after 1000 epochs

```
[[ 0.66666667  1.      ]  
 [ 0.33333333  0.55555556]  
 [ 1.         0.66666667]]
```

Actual Output:

```
[[ 0.92]  
 [ 0.86]  
 [ 0.89]]
```

Predicted Output after 1000 epochs

```
[[ 0.90015511]  
 [ 0.8766163 ]  
 [ 0.89265864]]
```

Loss:

```
0.000225663124995
```

Result:

Thus a python program has been written and executed to implement Backpropagation Neural Network model.

Ex. No. 2(b) BACK PROPAGATION NEURAL NETWORK FOR OR GATE

Date:

Aim:

To implement Back Propagation Neural Network for OR gate using Python programming.

Program:

```
import numpy as np
X = np.array([[0,0],[0,1],[1,0],[1,1]],dtype=float)
Y = np.array([[0],[1],[1],[1]],dtype=float)
print('value X->',X)
print('value Y->',y)
print('xrow,xcolm->',X.shape)
print('yrow,ycolm->',y.shape)

class NeuralNetwork(object):
    def __init__(self):
        self.inputs=2
        self.outputs=1
        self.hidden=2

        self.w1 = np.array([[0.6, -0.3], [-0.1, 0.4]]) # Input to hidden layer
        self.b1 = np.array([[0.3, 0.5]]) # Bias for hidden layer (1x2)

        self.w2 = np.array([[0.4], [0.1]]) # Hidden to output layer (2x1)
        self.b2 = np.array([[0.2]]) # Bias for output layer (1x1)
        self.lr=np.array([[0.25]])

    def forward(self,X):
        self.z=np.dot(X,self.w1) +self.b1
        print('value of Z ->\n',self.z)
        print('-----')
        self.z2=self.sigmoid(self.z)
        print('value of z2 -> \n',self.z2)
        print('-----')
        self.z3=np.dot(self.z2,self.w2) +self.b2
        print('value of z3 ->\n',self.z3)
        print('-----')
        o=self.sigmoid(self.z3)
        print('value of O\n',o)
        print("-----FORWARD OVER-----")
        print('-----\n\n')
        return o
```

```

def sigmoid(self,s):
    return 1/(1+np.exp(-s))

def sigmoidprime(self,s):
    return s*(1-s)

def backward(self,X,y,o):
    self.o_error=y-o          #ERROR CALCULATION
    print('value of o_error->\n',self.o_error)
    print('-----')

    self.o_delta=self.o_error*self.sigmoidprime(o) #ERROR X DERIVATIVE
    print('value of o_delta->\n',self.o_delta)
    print('-----')

    self.z2_error=self.o_delta.dot(self.w2.T) #
    print('value of z2_error->\n',self.z2_error)
    print('-----')
    self.z2_delta=self.z2_error*self.sigmoidprime(self.z2)
    print('value of z2_delta->\n',self.z2_delta)
    print('-----')
    self.w1+=X.T.dot(self.z2_delta) * 0.25
    print('Value for w1 after updation->\n',self.w1)
    print('-----')
    self.w2+=self.z2.T.dot(self.o_delta)*0.25
    print('Value for w2 after updation->\n',self.w2)
    print('-----')

def train(self,X,y):
    o=self.forward(X)
    self.backward(X,y,o)

nn=NeuralNetwork()

nn.train(X,y)

print("input:\n"+str(X))
print('-----')
print("Actual output:"+str(y))
print('-----')
print("loss:\n"+str(np.mean(np.square(y-nn.forward(X)))))
print('-----')
print("\n")

```

Input and Output:

```
value X-> [[0. 0.]
[0. 1.]
[1. 0.]
[1. 1.]]
value Y-> [[0.]
[1.]
[1.]
[1.]]
xrow,xcolm-> (4, 2)
yrow,ycolm-> (4, 1)
```

```
value of Z ->
[[0.3 0.5]
[0.2 0.9]
[0.9 0.2]
[0.8 0.6]]
```

```
-----
value of z2 ->
[[0.57444252 0.62245933]
[0.549834   0.7109495 ]
[0.7109495  0.549834  ]
[0.68997448 0.64565631]]
```

```
-----
value of z3 ->
[[0.09202294]
[0.09102855]
[0.1393632 ]
[0.14055542]]
```

```
-----
value of O
[[0.52298951]
[0.52274144]
[0.53478452]
[0.53508112]]
```

```
----FORWARD OVER-----..
0.2280669610416859
-----
```

Result:

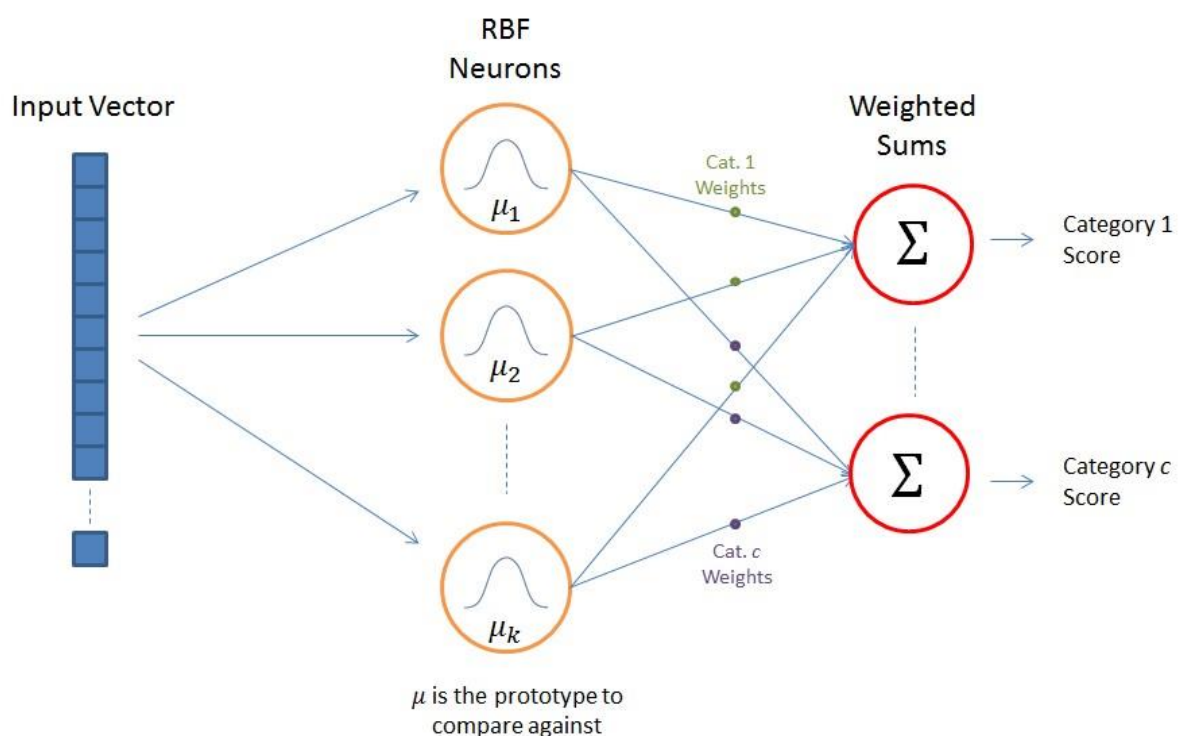
Thus a python program has been written and executed to implement Backpropagation Neural Network model for an OR gate.

Ex. No. 3 (a)**RADIAL BASIS FUNCTION NEURAL NETWORK****Date:****Aim:**

To implement Radial Basis Function Neural Network using Python Programming.

Radial Basis Function Neural Network (RBFNN):

A Radial Basis Function Network is a particular type of neural network. In this article, An RBFNN performs classification by measuring the input's similarity to examples from the training set. Each RBFNN neuron stores a “prototype”, which is just one of the examples from the training set. When we want to classify a new input, each neuron computes the Euclidean distance between the input and its prototype. If the input more closely resembles the class A prototypes than the class B prototypes, it is classified as class A.

**RBFNN Architecture:**

The above figure shows the typical architecture of an RBF Network. It consists of an input vector, a layer of RBF neurons, and an output layer with one node per category or class of data.

The Input Vector

The input vector is the n -dimensional vector that you are trying to classify. The entire input vector is shown to each of the RBF neurons.

The RBF Neurons

Each RBF neuron stores a “prototype” vector which is just one of the vectors from the training set. Each RBF neuron compares the input vector to its prototype, and outputs a value between 0 and 1 which is a measure of similarity. If the input is equal to the prototype, then the output of that RBF neuron will be 1. As the distance between the input and prototype grows, the response falls off exponentially towards 0. The shape of the RBF neuron’s response is a bell curve, as illustrated in the network architecture diagram. The neuron’s response value is also called its “activation” value. The prototype vector is also often called the neuron’s “center”, since it’s the value at the center of the bell curve.

The Output Nodes

The output of the network consists of a set of nodes, one per category that we are trying to classify. Each output node computes a sort of score for the associated category. Typically, a classification decision is made by assigning the input to the category with the highest score.

The score is computed by taking a weighted sum of the activation values from every RBF neuron. By weighted sum we mean that an output node associates a weight value with each of the RBF neurons, and multiplies the neuron’s activation by this weight before adding it to the total response.

Because each output node is computing the score for a different category, every output node has its own set of weights. The output node will typically give a positive weight to the RBF neurons that belong to its category, and a negative weight to the others.

RBF Neuron Activation Function

Each RBF neuron computes a measure of the similarity between the input and its prototype vector (taken from the training set). Input vectors which are more similar to the prototype return a result closer to 1. There are different possible choices of similarity functions, but the most popular is based on the Gaussian. Below is the equation for a Gaussian with a one-dimensional input.

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

Where x is the input, μ is the mean, and σ is the standard deviation. This produces the familiar bell curve shown below, which is centered at the mean, μ (in the below plot the mean is 5 and σ is 1). The RBF neuron activation function is slightly different, and is typically written as:

$$\phi(x) = e^{-\beta \|x - \mu\|^2}$$

In the Gaussian distribution, μ refers to the mean of the distribution. Here, it is the prototype vector which is at the center of the bell curve.

Program:

```
import numpy as np

import matplotlib.pyplot as plt

from sklearn.cluster import KMeans

from sklearn.datasets import make_classification

from sklearn.model_selection import train_test_split

from sklearn.metrics import accuracy_score

from scipy.spatial.distance import cdist

from sklearn.linear_model import LogisticRegression

X, y = make_classification(n_samples=1000, n_features=2, n_classes=2,
n_clusters_per_class=2, random_state=42, n_redundant=0)

print("X and y\n")

print(X[:5], y[:5], sep="\n\n")

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)

num_neurons = 10

kmeans = KMeans(n_clusters=num_neurons, random_state=42, n_init=10)

kmeans.fit(X_train)

centers = kmeans.cluster_centers_

print("Center:", centers)

std_dev = np.mean(cdist(centers, centers, 'euclidean')) / np.sqrt(2*num_neurons)

print("Standard Deviation:", std_dev)

def rbf_activation(X, centers, std_dev):

    return np.exp(-cdist(X, centers, 'sqeuclidean') / (2 * std_dev**2))

rbf_train = rbf_activation(X_train, centers, std_dev)

rbf_test = rbf_activation(X_test, centers, std_dev)

rbf_train = np.hstack([rbf_train, np.ones((rbf_train.shape[0], 1))])
```

```

rbf_test = np.hstack([rbf_test, np.ones((rbf_test.shape[0], 1))])

clf = LogisticRegression()

clf.fit(rbf_train, y_train)

y_pred = clf.predict(rbf_test)

accuracy = accuracy_score(y_pred, y_test)

print("Model Accuracy: ", accuracy*100)

# Visualization

def plot_result(X, y, centers=None):

    plt.scatter(X[y==0, 0], X[y==0, 1], label="0")

    plt.scatter(X[y==1, 0], X[y==1, 1], label="1")

    if centers is not None:

        plt.plot(centers[:, 0], centers[:, 1], 'rx', label="centers")

    plt.legend(loc="upper right")

rbf_x = rbf_activation(X, centers=centers, std_dev=std_dev)

activated_x = np.hstack([rbf_x, np.ones((rbf_x.shape[0], 1))])

plt.figure(figsize=(12, 6))

plt.subplot(121)

plt.title("Original")

plot_result(X, y)

plt.subplot(122)

plt.title("Predicted")

plot_result(X, clf.predict(activated_x), centers=centers)

```

Input and Output:

X and y

```

[[-0.99910178 -0.66386  ]
 [ 1.24668618  1.15359685]
 [ 0.96277683  0.85939747]
 [-2.95744095  2.03364529]]

```


[1.14116527 1.05944863]]

[1 1 1 1 1]

Center:

[[-0.8692364 -0.65864468]

[2.53446783 2.35687357]

[1.98458412 -1.101229]

[-2.16151064 2.60967372]

[-0.13595674 0.47263506]

[-1.41909706 -2.21144242]

[-1.89697275 0.99213769]

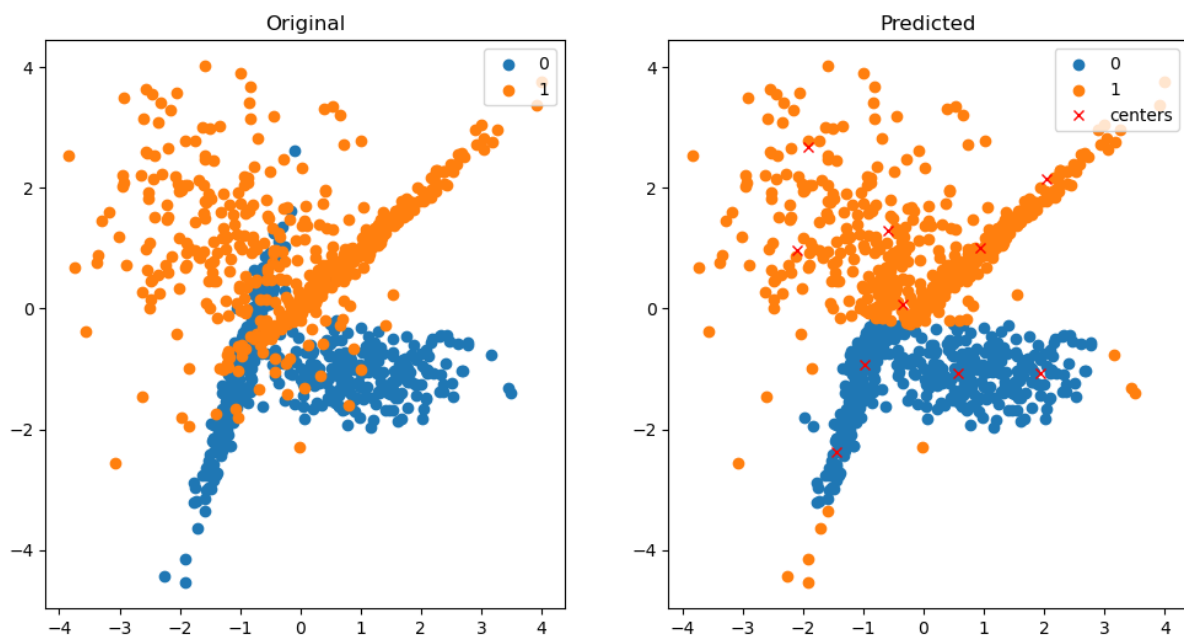
[1.23312034 1.21393961]

[0.61193541 -1.041979]

[-0.16678152 2.0800995]]

Standard Deviation: 0.6120404511235887

Model Accuracy: 85.0



Result:

Thus a python program has been written and executed to implement Radial Basis function Neural Network model.

Ex. No. 3(b)**RADIAL BASIS FUNCTION FOR XOR GATE****Date:****Aim:**

To implement Radial Basis Function Neural Network for XOR gate using Python Programming.

Program:

```
import numpy as np
import pandas as pd
from sklearn.cluster import KMeans
from scipy.spatial.distance import cdist

# Data
X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
y = np.array([0, 1, 1, 0])
weight = np.array([-1, 1, 1, -1])

# Use X as centers (no clustering in this case)
centers = np.copy(X)
print("Centers:\n", centers)

# Standard deviation
std_dev = 1

# RBF activation function
def rbf_activation(X, centers, std_dev):
    return np.around(np.exp(-cdist(X, centers, 'sqeuclidean') / (2 * std_dev**2)), decimals=1)

# Calculate RBF activations
rbf_x = rbf_activation(X, centers, std_dev)
```

```

# Prediction function

def pred():
    return np.dot(rbf_x, weight)

# Binary step function

def step_fun(s):
    return 1 if s > 0 else 0

# Calculate predictions

pred_values = pred()
step_predictions = np.array([step_fun(val) for val in pred_values])

# Create DataFrame

df = pd.DataFrame({
    'Input': list(map(tuple, X)),
    'RBF Activation': list(map(tuple, rbf_x)),
    'Predicted Value': pred_values,
    'Predicted Output': step_predictions
})

print("\nDataFrame:\n", df)

```

Input and Output:

Centers:

```

[[0 0]
 [0 1]
 [1 0]
 [1 1]]

```

DataFrame:

	Input	RBF Activation	Predicted Value	Predicted Output
0	(0, 0)	(1.0, 0.6, 0.6, 0.4)	-0.2	0
1	(0, 1)	(0.6, 1.0, 0.4, 0.6)	0.2	1
2	(1, 0)	(0.6, 0.4, 1.0, 0.6)	0.2	1
3	(1, 1)	(0.4, 0.6, 0.6, 1.0)	-0.2	0

Result:

Thus a python program has been written and executed to implement Radial Basis function Neural Network model for an XOR gate.

Ex. No. 4(a)**SUPPORT VECTOR MACHINE ON
SOCIAL MEDIA ADS DATASET****Date:****Aim:**

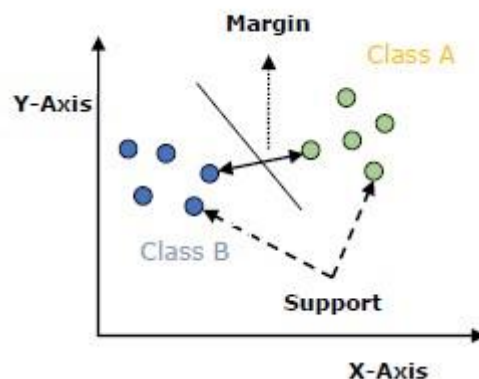
To implement a Support Vector Machine on Social Media Ads dataset using Python programming.

Support Vector Machine (SVM):

Support vector machines (SVMs) are powerful yet flexible supervised machine learning algorithms that are used both for classification and regression. But generally, they are used in classification problems. In the 1960s, SVMs were first introduced but later they got refined in 1990. SVMs have their unique way of implementation as compared to other machine learning algorithms. Lately, they are extremely popular because of their ability to handle multiple continuous and categorical variables.

Working of SVM

An SVM model is basically a representation of different classes in a hyperplane in multidimensional space. The hyperplane will be generated iteratively by SVM so that the error can be minimized. The goal of SVM is to divide the datasets into classes to find



a maximum marginal hyperplane (MMH).

The following are important concepts in SVM:

- **Support Vectors** – Datapoints that are closest to the hyperplane are called support vectors. Separating lines will be defined with the help of these data points.
- **Hyperplane** – As we can see in the above diagram, it is a decision plane or space which is divided between a set of objects having different classes.
- **Margin** – It may be defined as the gap between two lines on the closet data points of different classes. It can be calculated as the perpendicular distance from the line to the support vectors. A large margin is considered as a good margin and a small margin is considered as a bad margin.

The main goal of SVM is to divide the datasets into classes to find a maximum marginal hyperplane (MMH) and it can be done in the following two steps –

- First, SVM will generate hyperplanes iteratively that segregate the classes in the best way.
- Then, it will choose the hyperplane that separates the classes correctly.

Program:

```
import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

from matplotlib.colors import ListedColormap

from sklearn.preprocessing import StandardScaler

from sklearn.model_selection import train_test_split

from sklearn.metrics import confusion_matrix, accuracy_score

from sklearn.svm import SVC

from sklearn import datasets, svm, metrics

df = pd.read_csv('datasets/Social_Network_Ads.csv')

display(df.head())

X = df.iloc[:, [2, 3]].values

y = df.iloc[:, 4].values

x_train, x_test, y_train, y_test = train_test_split(X, y, test_size=0.25, random_state=0)

sc = StandardScaler()

x_train = sc.fit_transform(x_train)

x_test = sc.transform(x_test)

classifier = SVC(kernel='rbf', random_state=0)

classifier.fit(x_train, y_train)

y_pred = classifier.predict(x_test)

cm = confusion_matrix(y_pred, y_test)

accuracy = accuracy_score(y_pred, y_test)
```

```

print("Confusion Matrix:\n", cm)

print("Accuracy:", accuracy)

X1, X2 = np.meshgrid(
    np.arange(start=x_test[:, 0].min()-1, stop=x_test[:, 0].max() + 1, step=0.01),
    np.arange(start=x_test[:, 1].min()-1, stop=x_test[:, 1].max() + 1, step=0.01)
)

plt.contourf(
    X1, X2,
    classifier.predict(np.array([X1.ravel(), X2.ravel()]).T).reshape(X1.shape),
    alpha=0.75,
    cmap = ListedColormap(('red', 'green'))
)

plt.xlim(X1.min(), X1.max())

plt.xlim(X2.min(), X2.max())

for i, j in enumerate(np.unique(y_test)):
    plt.scatter(x_test[y_test==j, 0], x_test[y_test==j, 1], color=ListedColormap(('red',
'green'))(i), label=j)

plt.title('SVM(Test set)')

plt.xlabel('Age')

plt.xticks(())

plt.yticks(())

plt.ylabel('Estimated Salary')

plt.legend()

plt.show()

```

Input and Output:

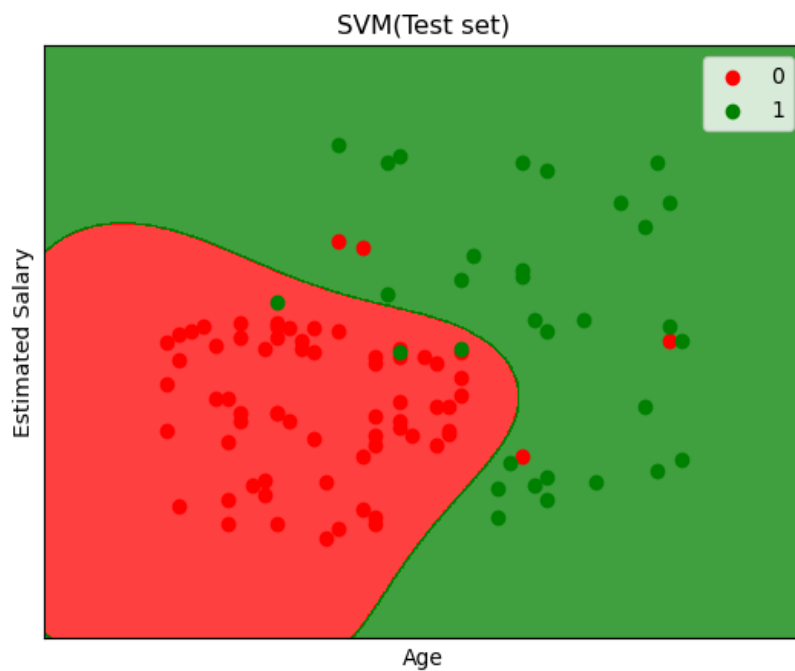
	Users	Gender	Age	EstimatedSalary	Purchased
25	15631159	Male	47	20000	1
26	15792818	Male	49	28000	1
27	15633531	Female	47	30000	1
28	15744529	Male	29	43000	0
29	15669656	Male	31	18000	0

Confusion Matrix:

[[64 3]

[4 29]]

Accuracy: 0.93



Result:

Thus a python program to implement Support Vector Machine on Social Media Ads Dataset has been written and executed.

Ex. No. 4(b) IMPLEMENTATION OF XOR GATE USING SUPPORT VECTOR MACHINE

Date:

Aim:

To implement XOR gate with Support Vector Machine using Python programming.

Program:

```
import numpy as np

import matplotlib.pyplot as plt

from sklearn.metrics import classification_report

from sklearn.svm import SVC

from sklearn.model_selection import train_test_split

np.random.seed(0)

print("X:")

X = np.random.randn(200, 2)

print(X[:5])

y_xor = np.logical_xor(X[:, 0] > 0, X[:, 1] > 0)

def plot_gate(X, y, title):

    plt.title(title)

    plt.scatter(X[y==1, 0], X[y==1, 1], c='b', marker='x', label='1')

    plt.scatter(X[y==0, 0], X[y==0, 1], c='r', marker='x', label='1')

    plt.show()

plot_gate(X, y_xor, "XOR")

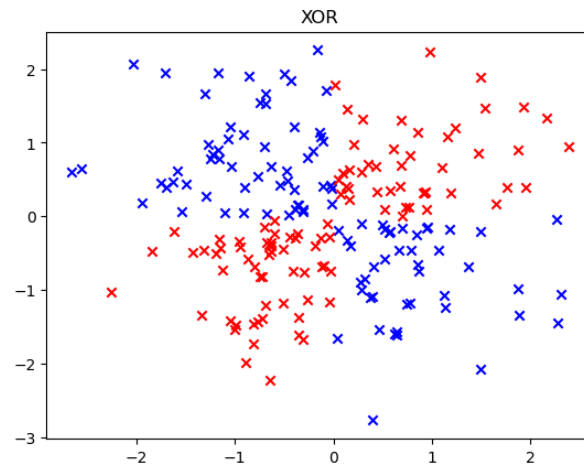
(train_data, test_data, train_labels, test_labels) = train_test_split(X, y_xor, test_size=0.25,
random_state=42)

model = SVC(kernel='poly', degree=2, coef0=1)

model.fit(train_data, train_labels)

print("Classification Report")

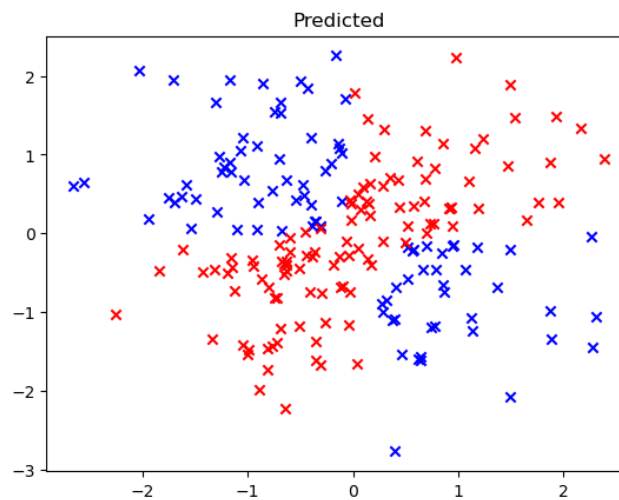
print(classification_report(test_labels, model.predict(test_data)))
```



Output:

Classification Report:

	precision	recall	f1-score	support
False	0.92	1.00	0.96	22
True	1.00	0.93	0.96	28
accuracy			0.96	50
macro avg	0.96	0.96	0.96	50
weighted avg	0.96	0.96	0.96	50



Result:

Thus a python program to implement XOR gate using Support Vector Machine has been written and executed.

Date:**Aim:**

To implement a Self-Organizing Map using Python Programming.

Self-Organizing Maps:

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

Working Principle of SOM:

Let's say an input data of size (m, n) where m is the number of training examples and n is the number of features in each example. First, it initializes the weights of size (n, C) where C is the number of clusters. Then iterating over the input data, for each training example, it updates the winning vector (weight vector with the shortest distance (e.g. Euclidean distance) from training example). Weight updation rule is given by:

$$w_{ij} = w_{ij}(\text{old}) + \alpha(t) * (x_{ik} - w_{ij}(\text{old}))$$

where alpha is a learning rate at time t, j denotes the winning vector, i denotes the i^{th} feature of training example and k denotes the k^{th} training example from the input data. After training the SOM network, trained weights are used for clustering new examples. A new example falls in the cluster of winning vectors.

Algorithm:

1. Weight initialization
2. For 1 to N number of epochs
3. Select a training example
4. Compute the winning vector
5. Update the winning vector
6. Repeat steps 3, 4, 5 for all training examples.
7. Clustering the test sample

Program:

```
import math

class SOM :

    # Function here computes the winning vector by Euclidean distance

    def winner( self, weights, sample ) :

        D0 = 0

        D1 = 0

        for i in range(len(sample)):

            D0 += (sample[i] - weights[0][i]) ** 2

            D1 += (sample[i] - weights[1][i]) ** 2

        If D0 < D1:

            return 0

        return 1 # Else Return 1

    # Function here updates the winning vector

    def update(weights, sample, j, alpha ) :

        for i in range(len(sample)):

            weights[j][i] = weights[j][i] + alpha * (sample[i] - weights[j][i])

        return weights

# Driver code

if __name__ == "__main__":

    # Training Examples ( m, n )

    T = [ [ 1, 1, 0, 0 ], [ 0, 0, 0, 1 ], [ 1, 0, 0, 0 ], [ 0, 0, 1, 1 ] ]

    m, n = len( T ), len( T[0] )

    # weight initialization ( n, C )

    weights = [ [ 0.2, 0.6, 0.5, 0.9 ], [ 0.8, 0.4, 0.7, 0.3 ] ]

    # training

    ob = SOM()

    epochs = 3
```

```

alpha = 0.5

for i in range( epochs ) :

    for j in range( m ) :

        # training sample

        sample = T[j]

        # Compute winner vector

        J = ob.winner( weights, sample )

        # Update winning vector

        weights = ob.update( weights, sample, J, alpha )

# classify test sample

s = [ 1, 1, 0, 1 ]

J = ob.winner( weights, s )

print( "Test Sample s belongs to Cluster : ", J )

print( "Trained weights : ", weights )

```

Input and Output:

Test sample s belongs to: 1

Trained weights: [[0.003125, 0.009375, 0.6640625, 0.9984375], [0.996875, 0.334375, 0.0109375, 0.0046875]]

Result:

Thus a python program has been written and executed to implement Self Organizing Map.

Ex. No. 5 (b)**Learning Vector Quantization****Date:****Aim:**

To implement Learning Vector Quantization using Python Programming.

Program:

```
import math

class LVQ:

    # Function to compute the winning vector by Euclidean distance

    def winner(self, weights, sample):

        distances = [sum(math.pow(sample[i] - weight[i], 2) for i in range(len(sample))) for
weight in weights]

        return distances.index(min(distances)) # Return index of the closest weight

    # Function to update the winning vector

    def update(self, weights, sample, J, alpha, actual):

        if actual == J: # If the predicted class matches the actual class

            for i in range(len(weights[J])):

                weights[J][i] += alpha * (sample[i] - weights[J][i]) # Move closer

        else:

            for i in range(len(weights[J])):

                weights[J][i] -= alpha * (sample[i] - weights[J][i]) # Move away

# Driver code

def main():

    # Training Samples (m, n) with their class vector

    X = [[0, 0, 1, 1], [1, 0, 0, 0],

        [0, 0, 0, 1], [0, 1, 1, 0],
```

```

[1, 1, 0, 0], [1, 1, 1, 0]]

Y = [0, 1, 0, 1, 1, 1] # Class labels
m, n = len(X), len(X[0])

# Weight initialization (n, c)
weights = []
weights.append(X.pop(0)) # First prototype
weights.append(X.pop(0)) # Second prototype

# Samples used in weight initialization will not be used in training
m -= 2
Y.pop(0)
Y.pop(0)

# Training
ob = LVQ()
epochs = 3
alpha = 0.1

for _ in range(epochs):
    for j in range(m):
        # Sample selection
        T = X[j]

        # Compute winner
        J = ob.winner(weights, T)

```

```

# Update weights
ob.update(weights, T, J, alpha, Y[j])

# Classify new input sample
T = [0, 0, 1, 0]
J = ob.winner(weights, T)
print("Sample T belongs to class:", J)
print("Trained weights:", weights)

if __name__ == "__main__":
    main()

```

Input and Output:

```

Sample T belongs to class: 0

Trained weights: [[0.0, -0.1791, 0.70299, 1.1791], [0.919, 0.52170310000000001,
0.312949000000000003, 0.0]]

```

Result:

Thus a python program has been written and executed to implement LVQ.

Ex. No. 6**FUZZY SET OPERATIONS****Date:****Aim:**

To write a MATLAB program to find the algebraic sum, algebraic subtraction, algebraic product, bounded sum, bounded subtraction, and bounded product of two fuzzy sets.

Algorithm:

1. Read the values of the two fuzzy sets.
2. Perform the algebraic sum operation by,

$$A + B = (a + b) - (a * b)$$

3. Perform the algebraic subtraction operation by,

$$A - B = (a + b^{\text{'}}) \quad \text{where } b^{\text{'}} = 1 - b$$

4. Perform the algebraic product operation by,

$$A * B = (a * b)$$

5. Perform the bounded sum operation by,

$$A \oplus B = \min [1, (a + b)]$$

6. Perform bounded subtraction operation by,

$$A \ominus B = \max [0, (a - b)]$$

7. Perform bounded product operation by,

$$A \odot B = \max [0, (a + b - 1)]$$

8. Display the results

Program:

```
a= input('Enter the fuzzy set a' )
```

```
b= input('Enter the fuzzy set b')
```

```
c= a + b
```

```
d= a .* b
```

```
as= c - d
```

```
e= 1 - b
```

```
ad= min(a , e)
```

```
f= a - b
```

```
bs= min (1, c)
```

```

bd= max (0, f)

g= c - 1

bp= max (0,g)

disp('The algebraic sum')

disp(as)

disp('The algebraic difference')

disp(ad)

disp('The algebraic product')

disp(d)

disp('The bounded sum')

disp(bs)

disp('The bounded difference')

disp (bd)

disp('The bounded product')

disp(bp)

```

Input and Output:

```

Enter fuzzy set a [1  0.5]
Enter fuzzy set b [0.4  0.2]
The algebraic sum
[1.0000  0.6000 ]
The algebraic difference
[1  0.9000]
The algebraic product
[0.4000  0.1000]
The bounded sum
[1.0000  0.7000]
The bounded difference
[0.6000  0.3000]
The bounded product
[0.4000  0]

```

Result:

Thus, a MATLAB program to perform simple fuzzy set operations has been executed and successfully verified.

Ex. No. 7**FUZZY SET PROPERTIES****Date:****Aim:**

To write a program in MATLAB to verify the properties of fuzzy sets.

Algorithm:

To Verify the following Fuzzy Set Properties:

Commutative property

$$\underline{A} \cup \underline{B} = \underline{B} \cup \underline{A}$$

$$\underline{A} \cap \underline{B} = \underline{B} \cap \underline{A}$$

Associative property

$$(\underline{A} \cup \underline{B}) \cup \underline{C} = \underline{A} \cup (\underline{B} \cup \underline{C})$$

$$(\underline{A} \cap \underline{B}) \cap \underline{C} = \underline{A} \cap (\underline{B} \cap \underline{C})$$

Distributive property

$$\underline{A} \cup (\underline{B} \cap \underline{C}) = (\underline{A} \cup \underline{B}) \cap (\underline{A} \cup \underline{C})$$

$$\underline{A} \cap (\underline{B} \cup \underline{C}) = (\underline{A} \cap \underline{B}) \cup (\underline{A} \cap \underline{C})$$

Absorption property

$$\underline{A} \cup (\underline{A} \cap \underline{C}) = \underline{A}$$

$$\underline{A} \cap (\underline{A} \cup \underline{C}) = \underline{A}$$

Idempotency property

$$\underline{A} \cup \underline{A} = \underline{A}$$

$$\underline{A} \cap \underline{A} = \underline{A}$$

De Morgan's law

$$(\underline{A} \cup \underline{B})' = \underline{A}' \cap \underline{B}'$$

$$(\underline{A} \cap \underline{B})' = \underline{A}' \cup \underline{B}'$$

Identity property

$$\underline{A} \cup \phi = \underline{A}$$

$$\underline{A} \cap X = \underline{A}$$

Axiom of Excluded Middle

$$\underline{A} \cap \varphi = \varphi$$

Axiom of Contradiction

$$\underline{A} \cup X = X$$

Program:

```
a = [1.0, 0.4, 0.2];
b = [0.2,0.43,0.44];
c = [0.2,0.93,0.24];
e = [1 1 1];
o = [0,0,0];

disp('A: ');
disp(a);
disp('B: ');
disp(b);
disp('C: ');
disp(c);
a_U_b = max(a, b);
a_N_b = min(a, b);
disp('A U B: ');
disp(a_U_b);
disp('A N B: ');
disp(a_N_b);
% Commutative Property
disp('Commutative Property: ');
disp('A U B: '); disp(a_U_b); disp('B U A: '); disp(max(b, a));
disp('A N B: '); disp(a_N_b); disp('B N A: '); disp(min(b, a));

% Associative Property
disp('Associative Property: ');
```

```

disp('(A U B) U C: '); disp(max(max(a, b), c)); disp('A U (B U C): '); disp(max(a, max(b,
c)));
disp('(A N B) N C: '); disp(min(min(a, b), c)); disp('A N (B N C): '); disp(min(a, min(b, c)));

```

% Distributive Property

```

disp('Distributive Property: ');
disp('A U (B N C): '); disp(max(a, min(b, c))); disp('(A U B) N (A U C): ');
disp(min(max(a,b) , max(a, c)));
disp('A N (B U C): '); disp(min(a, max(b, c))); disp('(A N B) U (A N C): ');
disp(max(min(a,b) , min(a, c)));

```

% Absorption Property

```

disp('Absorption Property: ');
disp('A U (A N B): '); disp(max(a, min(a, b))); disp('A: '); disp(a);
disp('A N (A U B): '); disp(min(a, max(a, b))); disp('A: '); disp(a);

```

% Idempotent Property

```

disp('Idempotent Law: ');
disp('A U A: '); disp(max(a, a)); disp('A: '); disp(a);
disp('A N A: '); disp(min(a, a)); disp('A: '); disp(a);

```

% Involution Property

```

disp('Involution: ');
disp('(A_i)_i: '); disp(1-(1- a)); disp('A: '); disp(a);

```

% De Morgan's Law

```

disp("De Morgan's Law: ");
disp('(A U B)_i: '); disp( 1 - a_U_b); disp('A_i N B_i: '); disp(min(1-a, 1-b));
disp('(A N B)_i: '); disp( 1 - a_N_b); disp('A_i U B_i: '); disp(max(1-a, 1-b));

```

% Identity Property

```

disp('Identity Property:');

```

```

disp('A U o: '); disp(max(a, o)); disp('A: '); disp(a);
disp('A N o: '); disp(min(a, o)); disp('o: '); disp(o);
% Axiom of Excluded middle
disp('Axiom of Excluded middle:')
disp('A U A_i'); disp(max(a, 1-a)); disp('E: '); disp(e);
% Axiom of Contradiction
disp('Axiom of Contradiction:')
disp('A N A_i'); disp(min(a, 1-a)); disp('o: '); disp(o);

```

Input and Output:

```

A:
  1.0000  0.4000  0.2000
B:
  0.2000  0.4300  0.4400
C:
  0.2000  0.9300  0.2400
A U B:
  1.0000  0.4300  0.4400
A N B:
  0.2000  0.4000  0.2000

```

Commutative Property:

```

A U B:
  1.0000  0.4300  0.4400
B U A:
  1.0000  0.4300  0.4400
A N B:
  0.2000  0.4000  0.2000
B N A:
  0.2000  0.4000  0.2000

```

Associative Property:

```

(A U B) U C:
  1.0000  0.9300  0.4400
A U (B U C):
  1.0000  0.9300  0.4400
(A N B) N C:
  0.2000  0.4000  0.2000
A N (B N C):
  0.2000  0.4000  0.2000

```

Distributive Property:

$A \cup (B \cap C)$:

1.0000 0.4300 0.2400

$(A \cup B) \cap (A \cup C)$:

1.0000 0.4300 0.2400

$A \cap (B \cup C)$:

0.2000 0.4000 0.2000

$(A \cap B) \cup (A \cap C)$:

0.2000 0.4000 0.2000

Absorption Property:

$A \cup (A \cap B)$:

1.0000 0.4000 0.2000

A:

1.0000 0.4000 0.2000

$A \cap (A \cup B)$:

1.0000 0.4000 0.2000

A:

1.0000 0.4000 0.2000

Idempotent Law:

$A \cup A$:

1.0000 0.4000 0.2000

A:

1.0000 0.4000 0.2000

$A \cap A$:

1.0000 0.4000 0.2000

A:

1.0000 0.4000 0.2000

Involution Property:

$(A_i)_i$:

1.0000 0.4000 0.2000

A:

1.0000 0.4000 0.2000

De Morgan's Law:

$(A \cup B)_i$:

0 0.5700 0.5600

$A_i \cap B_i$:

0 0.5700 0.5600

$(A \cap B)_i$:

0.8000	0.6000	0.8000
A _i U B _i :		
0.8000	0.6000	0.8000

Identity Property:

A U o:		
1.0000	0.4000	0.2000
A:		
1.0000	0.4000	0.2000
A N o:		
0	0	0
o:		
0	0	0

Axiom of Excluded middle:

A U A _i		
1.0000	0.6000	0.8000
E:		
1	1	1

Axiom of Contradiction:

A N A _i		
0	0.4000	0.2000
o:		
0	0	0

Result:

Thus, a MATLAB program to verify fuzzy set properties has been executed and successfully verified.

Ex. No. 8**FUZZY MEMBERSHIP FUNCTIONS****Date:****Aim:**

To write a program in MATLAB to plot triangular, trapezoidal and bell shaped membership functions.

Algorithm:

1. Set the limits of x-axis.
2. Calculate y using trimf() function with three parameters for triangular membership function.
3. Calculate y using trapmf() function with four parameters for trapezoidal membership function.
4. Calculate y using gbellmf() function with three parameters for bell shaped membership function.
5. Plot x and y values.

Program:

% Triangular membership function

x=(0.0:1.0:10.0)';

y1= trimf(x, [1 3 5]);

subplot(311)

plot(x,[y1]);

% Trapezoidal membership function

x=(0.0:1.0:10.0)';

y1= trapmf(x, [1 3 5 7]);

subplot(312)

plot(x, [y1]);

% Bell-shaped membership function

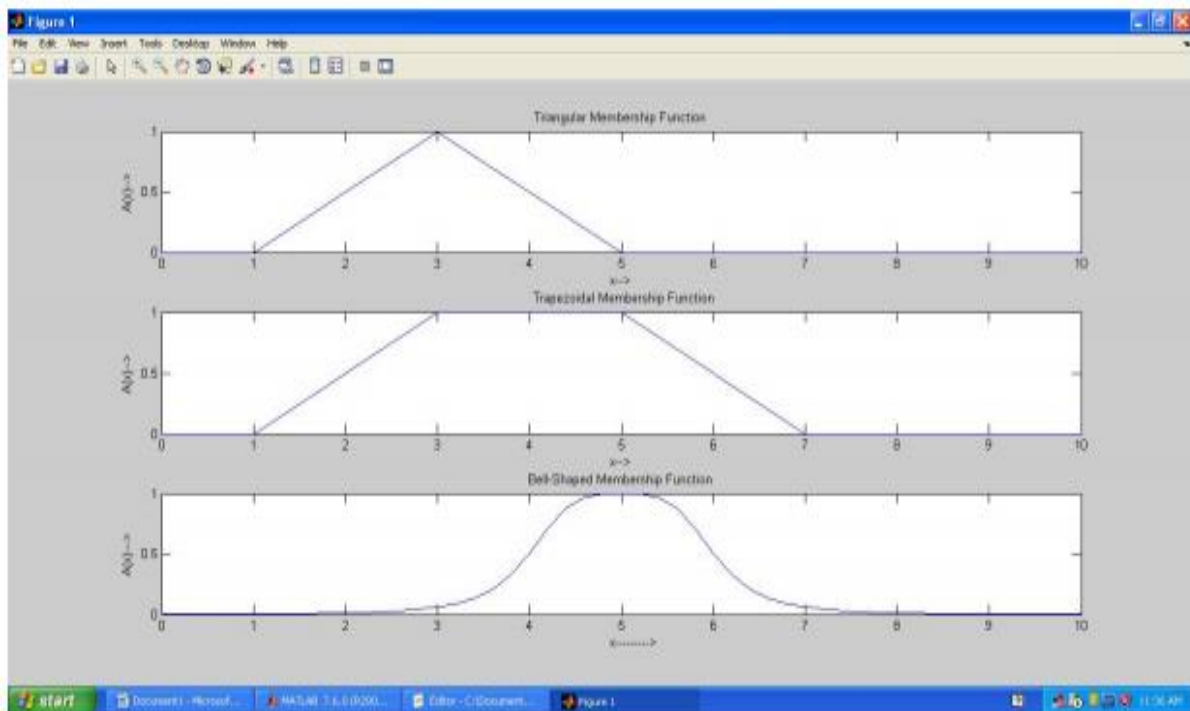
x=(0.0:0.2:10.0);

y1=gbellmf (x,[3 57]);

subplot(313)

plot(x, [y1]);

Output:



Result:

Thus, the MATLAB program for plotting membership functions has been executed successfully and the output is verified.

Ex. No. 9**FUZZY INFERENCE SYSTEM****Date:****Aim:**

To implement a Fuzzy Inference System (FIS) using MATLAB.

Inputs: Temperature and Cloud Cover**Temperature:** {Freeze, Cool, Warm and Hot}**Cloud Cover:** {Sunny, Partly Cloud, and Overcast}**Output:** Speed**Speed** : {Fast and Slow}**Rules:**

1. If cloud cover is Sunny and temperature is warm, then drive Fast
Sunny (Cover) and Warm (Temp) -> Fast (Speed)
2. If the cloud cover is cloudy and the temperature is cool, then drive Slow
Cloudy (Cover) and Cool (Temp) -> Slow (Speed)

Procedure:

1. Go to the command window in Matlab and type fuzzy.
2. Now, a new Fuzzy Logic Designer window will be opened.
3. Input / Output Variable
 - a. Go to Edit Window and click Add variable.
 - b. As per our requirements create two input variables, Temperature and Cloud Cover.
 - c. Create one output variable, Speed.
4. Temperature:
 - a. Double click the Temperature input variable in the Fuzzy Logic Designer window.
 - b. A new window will be opened and remove all the Membership Functions.
 - c. Now, Go to Edit, Click Add MFs and select the 4 Parameters for Temperature Class.
 - d. Change the following fields as mentioned data in the given below table.

Inputs : Temperature □ Freezing, Cool, Warm and Hot			
MF1: Range : [0 110] Name : Freezing Type :trapmf Parameter [0 0 30 50]	MF2: Range : [0 110] Name : Cool Type :trimf Parameter [30 50 70]	MF3: Range : [0 110] Name : Warm Type :trimf Parameter [50 70 90]	MF4: Range : [0 110] Name : Hot Type :trapmf Parameter [70 90 110 110]

5. Similarly, add the datas to the Cloud Cover variables and Speed variables.

6. Cloud Cover:

Inputs : Cloud Cover □ Sunny, Partly Cloud and Overcast		
MF1: Range : [0 100] Name : Sunny Type :trapmf Parameter [0 0 20 40]	MF2: Range : [0 100] Name : Partly Cloud Type :trimf Parameter [20 50 80]	MF3: Range : [0 100] Name : Overcast Type :trapmf Parameter [60 80 100]

7. Speed:

Output : Speed □ Slow and Fast	
MF1: Range : [0 100] Name : Slow Type :trapmf Parameter [0 0 25 75]	MF2: Range : [0 100] Name : Fast Type :trapmf Parameter [25 75 100 100]

8. Goto Rules: Edit ☐ Rules

9. Add the Rules

Rule-1: Sunny (Cover) and Warm (Temp) -> Fast (Speed)

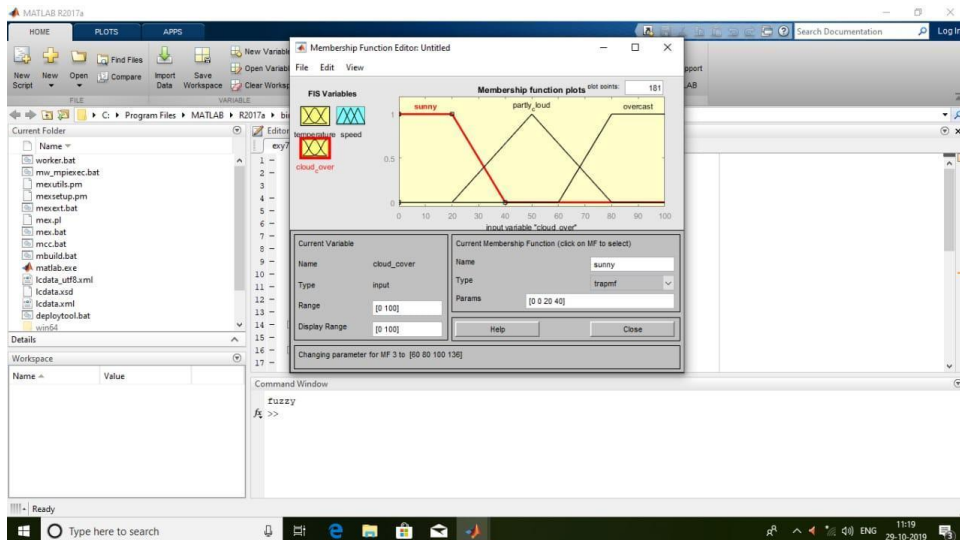
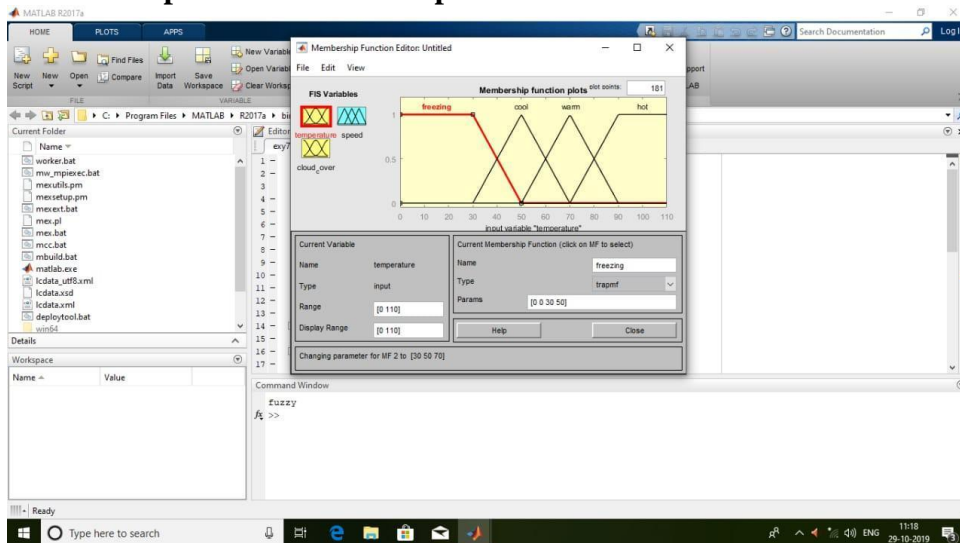
Rule-2: Cloudy (Cover) and Cool (Temp) -> Slow (Speed)

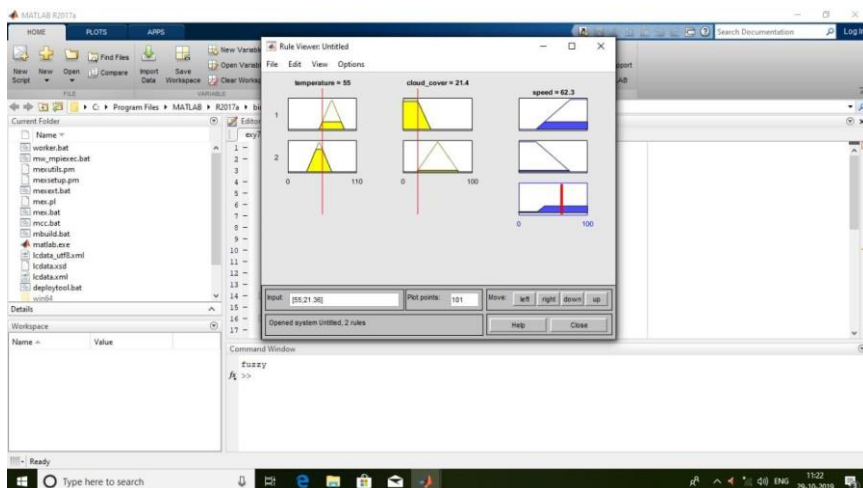
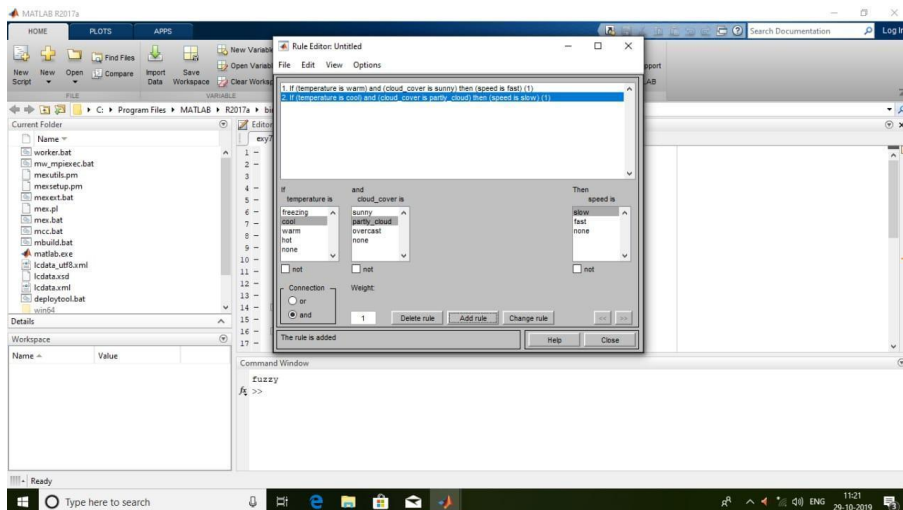
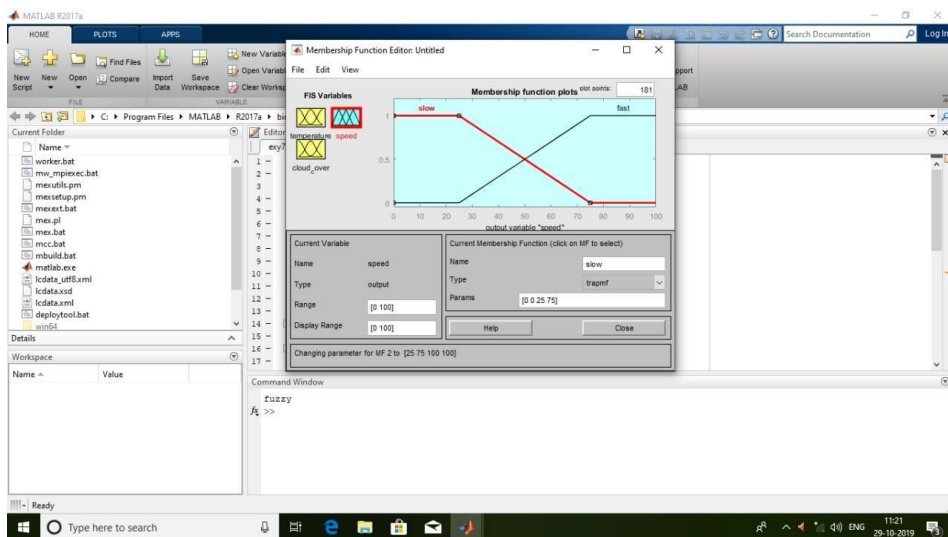
10. Go to view ☐ Rules

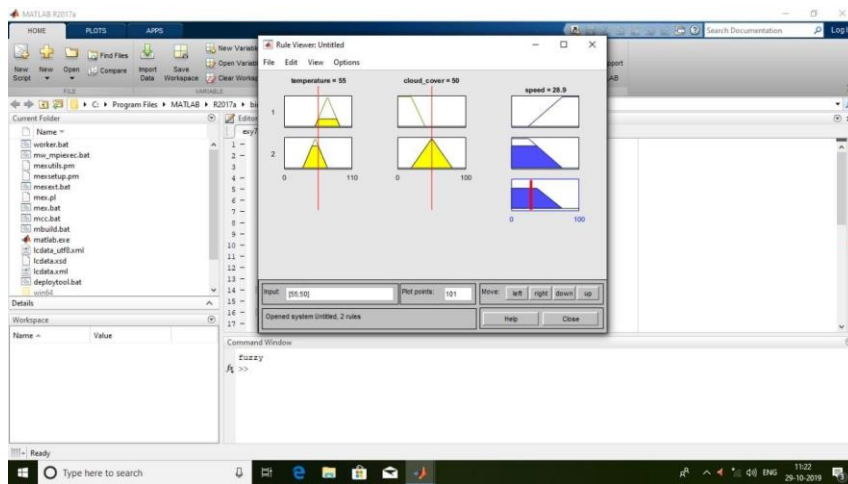
11. Exit.

Input and Output:

Membership functions for Temperature variable







Result:

Thus a Fuzzy Inference System is successfully executed using MATLAB.

Ex. No. 10**Defuzzification Methods****Date:****Aim:**

To write a program in Python to implement defuzzification methods using fuzzy toolbox.

Algorithm:

1. Import numpy, matplotlib and skfuzzy packages.
2. Trapezoidal membership function on range [0, 1] using trapmf function with four parameters.
3. Defuzzify trapezoidal membership function by using fuzzy toolbox functions centroid, bisector, mom, Som and lom.
4. Plot the defuzzified values using matplotlib.

Program:

```
import numpy as np

import matplotlib.pyplot as plt

import skfuzzy as fuzz

# Generate trapezoidal membership function on range [0, 1]

x = np.arange(0, 5.05, 0.1)

mfx = fuzz.trapmf(x, [2, 2.5, 3, 4.5])

# Defuzzify this membership function five ways

defuzz_centroid = fuzz.defuzz(x, mfx, 'centroid')

defuzz_bisector = fuzz.defuzz(x, mfx, 'bisector')

deuzz_mom = fuzz.defuzz(x, mfx, 'mom')

defuzz_som = fuzz.defuzz(x, mfx, 'som')

defuzz_lom = fuzz.defuzz(x, mfx, 'lom')
```



```

# Collect info for vertical lines

labels = ['centroid', 'bisector', 'mean of maximum', 'min of maximum', 'max of maximum']

xvals = [defuzz_centroid, defuzz_bisector, defuzz_mom, defuzz_som, defuzz_lom]

colors = ['r', 'b', 'g', 'c', 'm']

ymax = [fuzz.interp_membership(x, mfx, i) for i in xvals]

# Display and compare defuzzification results against membership function

plt.figure(figsize=(8, 5))

plt.plot(x, mfx, 'k')

for xv, y, label, color in zip(xvals, ymax, labels, colors):

    plt.vlines(xv, 0, y, label=label, color=color)

plt.ylabel('Fuzzy membership')

plt.xlabel('Universe variable (arb)')

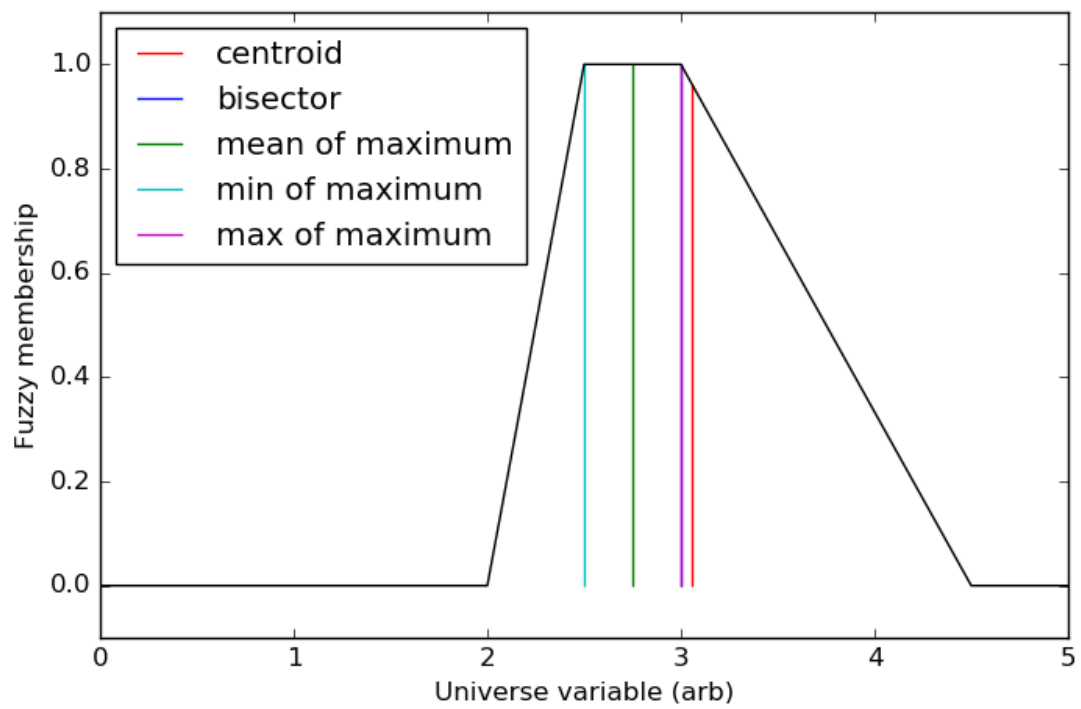
plt.ylim(-0.1, 1.1)

plt.legend(loc=2)

plt.show()

```

Output:



Result:

Thus, the Python program for implementing defuzzification methods has been executed successfully and the output is verified.

Ex. No. 11

THE TIPPING PROBLEM

Date:

Aim:

To develop a fuzzy controller for the Tipping problem using Python programming.

The Tipping Problem:

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

A fuzzy control system is created to model how you might choose to tip at a restaurant. When tipping, the service and food quality are considered and rated between 0 and 10. A tip of between 0 and 25% is suggested.

Problem Formulation:

Inputs

service

Universe (ie, crisp value range): How good was the service of the wait staff, on a scale of 0 to 10?

Fuzzy set (ie, fuzzy value range): poor, acceptable, amazing

food quality

Universe: How tasty was the food, on a scale of 0 to 10?

Fuzzy set: bad, decent, great

Consequents (Outputs)

tip

Universe: How much should we tip, on a scale of 0% to 25%

Fuzzy set: low, medium, high

Rules

IF the **service** was good **or** the **food quality** was good,
THEN the tip will be high.

IF the **service** was average, THEN the **tip** will be medium.

IF the **service** was poor **and** the **food quality** was poor
THEN the tip will be low.

Usage

If I tell this controller that I rated:
the service as 9.8, and
the quality as 6.5,
it would recommend I leave:
a 20.2% tip.

Creating the Tipping Controller Using the skfuzzy control API

```
import numpy as np
```

```
import skfuzzy as fuzz
```

```

from skfuzzy import control as ctrl

# New Antecedent/Consequent objects hold universe variables and membership
# functions

quality = ctrl.Antecedent(np.arange(0, 11, 1), 'quality')
service = ctrl.Antecedent(np.arange(0, 11, 1), 'service')
tip = ctrl.Consequent(np.arange(0, 26, 1), 'tip')

# Auto-membership function population is possible with .automf(3, 5, or 7)
quality.automf(3)
service.automf(3)

# Custom membership functions can be built interactively with a familiar,
# Pythonic API
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])

# You can see how these look with .view()
quality['average'].view()
service.view()
tip.view()

rule1 = ctrl.Rule(quality['poor'] | service['poor'], tip['low'])
rule2 = ctrl.Rule(service['average'], tip['medium'])
rule3 = ctrl.Rule(service['good'] | quality['good'], tip['high'])
rule1.view()

tipping_ctrl = ctrl.ControlSystem([rule1, rule2, rule3])

```

```

tipping = ctrl.ControlSystemSimulation(tipping_ctrl)

# Pass inputs to the ControlSystem using Antecedent labels with Pythonic API
# Note: if you like passing many inputs all at once, use .inputs(dict_of_data)

tipping.input['quality'] = 6.5
tipping.input['service'] = 9.8

# Crunch the numbers

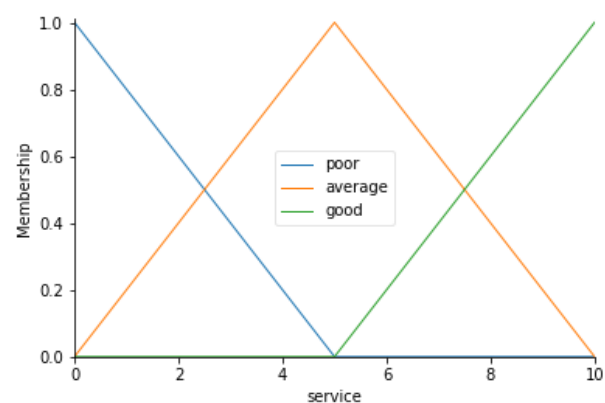
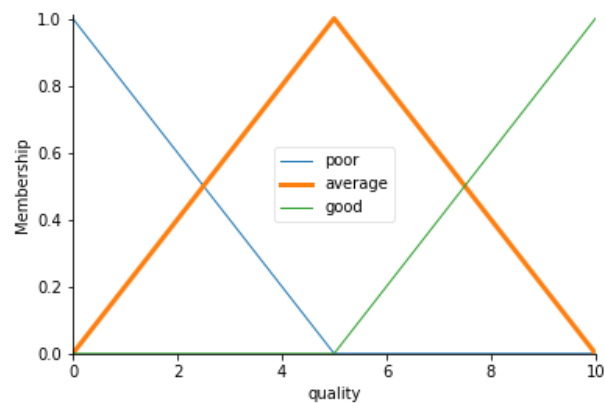
tipping.compute()

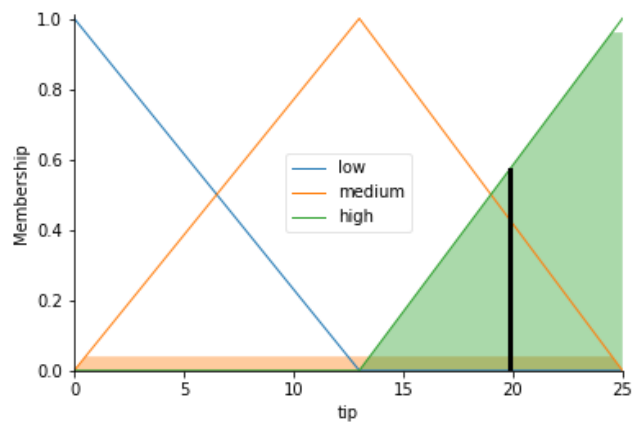
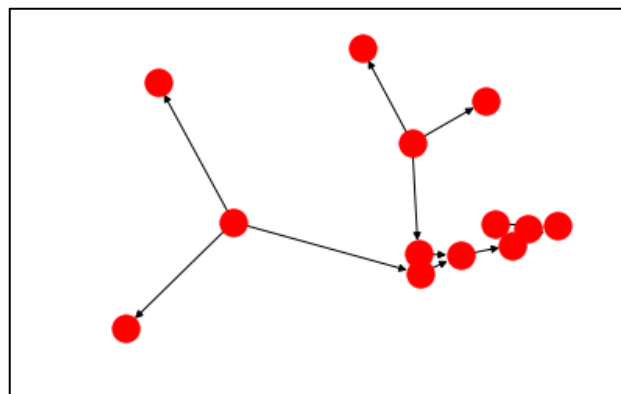
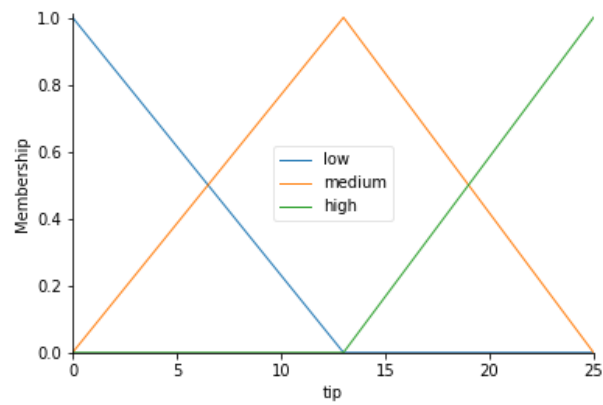
print(tipping.output['tip'])

tip.view(sim=tipping)

```

Output:





The resulting suggested tip is 20.24%.

Result:

Thus, the Python program for developing a fuzzy controller for the Tipping problem has been executed successfully and the output is verified.