|  |  |
| --- | --- |
| **Name:** | Bodhisatya Ghosh |
| **Branch:** | CSE – Data Science |
| **Batch:** | B |
| **Course:** | Soft Computing |
| **Experiment no:** | 1 |

**Aim:** To implement activation functions

**Theory:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Name** | **Graph** | **Formula** | **Significance** | **Description** | **Usage** |
| Identity | Figure B.7: Plot of the Linear / Identity activation function. For a... |  Download Scientific Diagram | f(x) = x | A linear function | The identity function is a special case of an activation function where the output signal is equal to the input signal. In other words, the identity function simply passes the input signal through unchanged. | Used when there is no need for any activation function but consistency is required across all layers |
| Binary Step | Getting to know Activation Functions in Neural Networks. | by Hasara Samson  | Towards Data Science | f(x) = 1,x>=0  = 0, x<0 | Most common activation function in neural networks | Binary step function is one of the simplest activation functions. The function produces binary output and thus the name binary step funtion. The function produces 1 (or true) when input passes a threshold limit whereas it produces 0 (or false) when input does not pass threshold. | Used in single-layer nets to convert the net input to an output that is a binary (1 or 0) |
| Bipolar Step | Activation Functions – Machine Learning Geek |  |  | In the **Bipolar Step Function**, if the value of Y is above a certain value known as the threshold, the output is +1 and if it’s less than the threshold then the output is -1. | It has bipolar outputs (+1 to -1). It can be utilized in single-layer networks. |
| Binary Sigmoid | Binary sigmoid activation function The limited numeric response range,... |  Download Scientific Diagram | f(x) = 1/(1+e^(-x)) | It is differentiable, non-linear, and produces non-binary activations But the problem with Sigmoid is the vanishing gradients. | Binary Sigmoid Function or **Sigmoid function** is a logistic function where the output values are either binary or vary from 0 to 1. | The sigmoid function extracts a bounded absolute value from the model's output. Can be used in logistic problems. |
| Bipolar Sigmoid | 3: Bipolar sigmoid function. | Download Scientific Diagram |  |  | This is the function from where the Hyperbolic Tan Function was derived from. Here (lambda) represents the steepness factor. The range of this function is between -1 and 1. For the hyperbolic tangent function, the value of the steepness factor is 2. | If the network uses the binary data, then it is better to convert it to bipolar form and use the bipolar sigmoidal activation function or hyperbolic tangent function. |
| Ramp | Ramp function - Wikipedia |  |  | 0 for negative inputs, output equals input for non-negative inputs |  |
| ReLu | 6: Graph of ReLu activation function | Download Scientific Diagram | f(x) = max(0,x) |  | It is a piecewise linear function that is defined to be 0 for all negative values of x and equal to a × x otherwise, where a is a learnable parameter. |  |

**Program:**

from math import exp as e

import numpy as np

def identity(x):

    return x

def binary\_step(x):

    if(x>=0):

        return 1

    return 0

def bipolar\_step(x):

    if(x>=0):

        return 1

    return -1

def binary\_sigmoid(x):

    val = 1/(1+(e(x\*-1)))

    return val

def bipolar\_sigmoid(x):

    val = (1-e(x \* -1))/(1+e(x \* -1))

    return val

def ramp(x):

    if(x >= 1):

        return 1

    elif (0 <= x <= 1):

        return x

    return 0

def relu(x):

    return max(0,x)

def main():

    n = int(input("Enter number of input: "))

    x = []

    w = []

    yin = 0

    for i  in range(n):

        xn = float(input("Enter value of x{}: ".format(i+1)))

        wn = float(input("Enter weight of x{}: ".format(i+1)))

        x.append(xn)

        w.append(wn)

    x = np.array(x)

    w = np.array(w)

    for i in range(len(x)):

        yin = yin + (x[i]\*w[i])

    print("Identity: {}".format(identity(yin)))

    print("Binary step: {}".format(binary\_step(yin)))

    print("Bipolar step: {}".format(bipolar\_step(yin)))

    print("Binary sigmoid: {}".format(binary\_sigmoid(yin)))

    print("Bipolar sigmoid: {}".format(bipolar\_sigmoid(yin)))

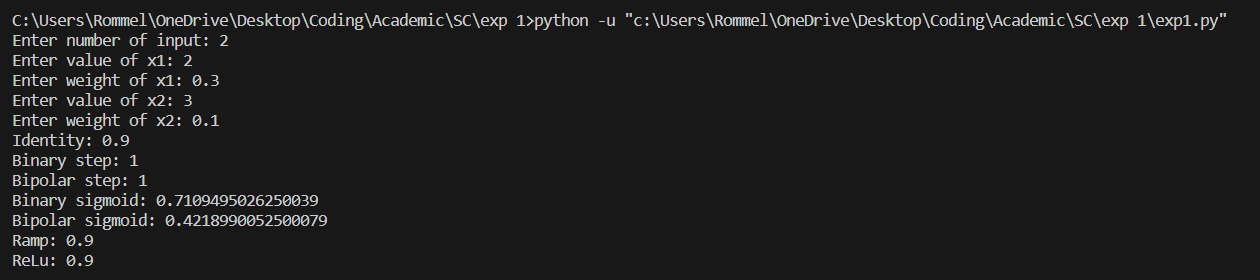
    print("Ramp: {}".format(ramp(yin)))

    print("ReLu: {}".format(relu(yin)))

if \_\_name\_\_ == '\_\_main\_\_':

    main()

**Results:**



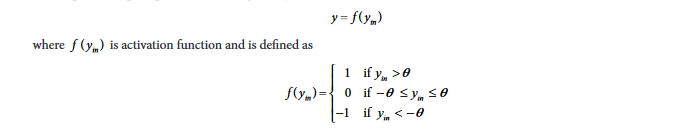
**Conclusion:** In this experiment we have learnt how to implement activation functions

|  |  |
| --- | --- |
| **Name:** | Bodhisatya Ghosh |
| **Branch:** | CSE – Data Science |
| **Batch:** | B |
| **Course:** | Soft Computing |
| **Experiment no:** | 2 |

**Aim:** To implement a perceptron for a given problem statement and design ANN for the same using Joone Editor.

**Theory:** Perceptron networks come under single-layer feed-forward networks and are also called simple perceptrons. The perceptron network consists of three units, namely, sensory unit (input unit), associator unit (hidden unit), response unit (output unit). The sensory units are connected to associator units with fixed weights having values 1, 0 or -1, which are assigned at random. The binary step with fixed threshold q is used as activation for associator. The output of the perceptron network is given by:



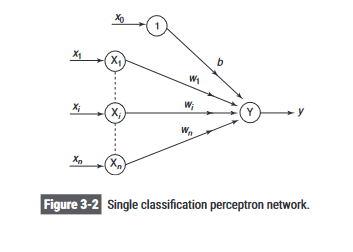


The perceptron learning rule is used in the weight updation between the associator unit and the response unit. For each training input, the net will calculate the response and it will determine whether or not an error has occurred. The error calculation is based on the comparison of the values of targets with those of the calculated outputs. The weights on the connections from the units that send the nonzero signal will get adjusted suitably. The weights will be adjusted on the basis of the learning rule if an error has occurred for a particular training pattern, i.e.



The entire loop of the training process continues until the training input pair is presented to the network. The training (weight updation) is done on the basis of the comparison between the calculated and desired output. The loop is terminated if there is no change in weight.

Single classification perceptron network:

****

**Program:**

#include <stdio.h>

#include <stdlib.h>

#include <math.h>

int activation\_function(float yin,float theta){

    if(yin > theta)

    {

        return 1;

    }

    else if((yin >= -1\*theta) && (yin <= theta))

    {

        return 0;

    }

    else if (yin < theta)

    {

        return -1;

    }

}

float \* step\_calc(int step\_num,int\*\* vector,int\* targets,int bias,float theta,int learning\_rate,float\* weights,int n)

{

    float yin = weights[0] \* bias;

    for (int i = 1; i < n+1; i++)

    {

        yin = yin + (vector[step\_num-1][i-1] \* weights[i]);

    }

    int y = activation\_function(yin,theta);

    // printf("Y after activation: %d\n",y);

    if(targets[step\_num-1] == y)

    {

        return weights;

    }

    // printf("Y after activation: %d\n",y);

    weights[0] = weights[0] + (learning\_rate\*targets[step\_num-1]);

    for (int i = 1; i < n+1; i++)

    {

        weights[i] = weights[i] + (learning\_rate \* targets[step\_num-1] \* vector[step\_num-1][i-1]);

    }

    return weights;

}

int main()

{

    int n,vec,epoch,bias,learning\_rate,threshhold;

    printf("Enter number of vectors: ");

    scanf("%d",&vec);

    printf("Enter number of elements per vector: ");

    scanf("%d",&n);

    int\*\* vector = malloc(vec \* sizeof(int \*));

    for (int i = 0; i < vec; i++)

    {

        vector[i] = (int\*)malloc(n \* sizeof(int));

    }

    int\* targets = malloc(vec \* sizeof(int));

    float\* weights = malloc((n+1) \* sizeof(float));

    for (int i = 0; i < vec; i++)

    {

        printf("\nFor vector %d\n",i+1);

        for(int j = 0; j < n; j++)

        {

            printf("Enter element %d: ",j+1);

            scanf("%d",&vector[i][j]);

        }

        printf("Enter target for vector %d: ",i+1);

        scanf("%d",&targets[i]);

    }

    for (int i = 0; i < n+1; i++)

    {

        printf("Enter initial weight %d: ",i);

        scanf("%d",&weights[i]);

    }

    printf("Enter threshhold: ");

    scanf("%f",&threshhold);

    printf("Enter bias: ");

    scanf("%d",&bias);

    printf("Enter learning rate: ");

    scanf("%d",&learning\_rate);

    printf("Enter number of epochs: ");

    scanf("%d",&epoch);

    for(int ep = 0;ep < epoch; ep++)

    {

        printf("\nEpoch %d\n",ep+1);

        for(int step = 1; step <= vec; step++)

        {

            weights = step\_calc(step,vector,targets,bias,0.2,learning\_rate,weights,n);

        }

        for (int i = 0; i < n+1; i++)

        {

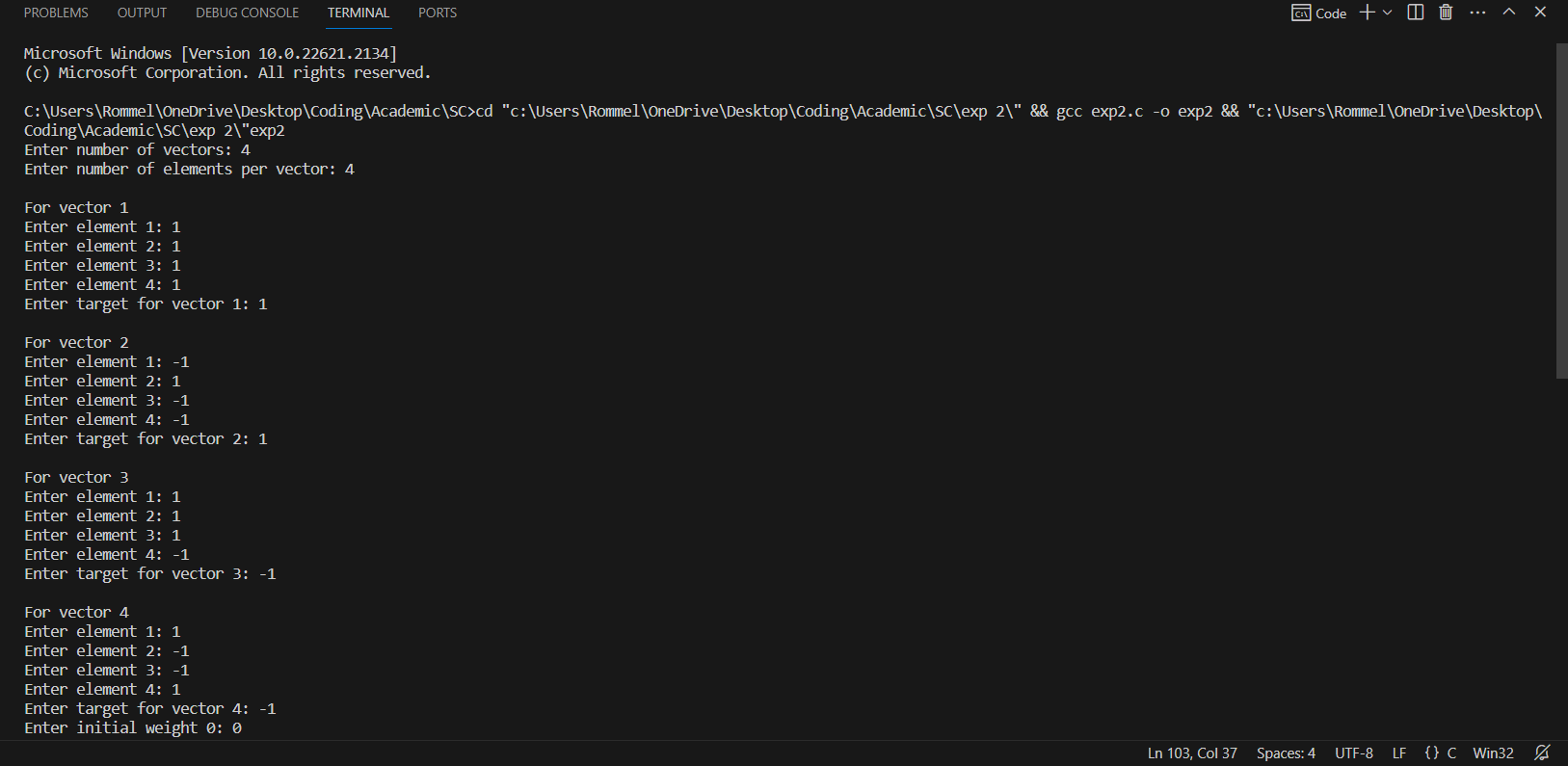
            printf("Weight %d after epoch %d: %f\n",i,ep+1,weights[i]);

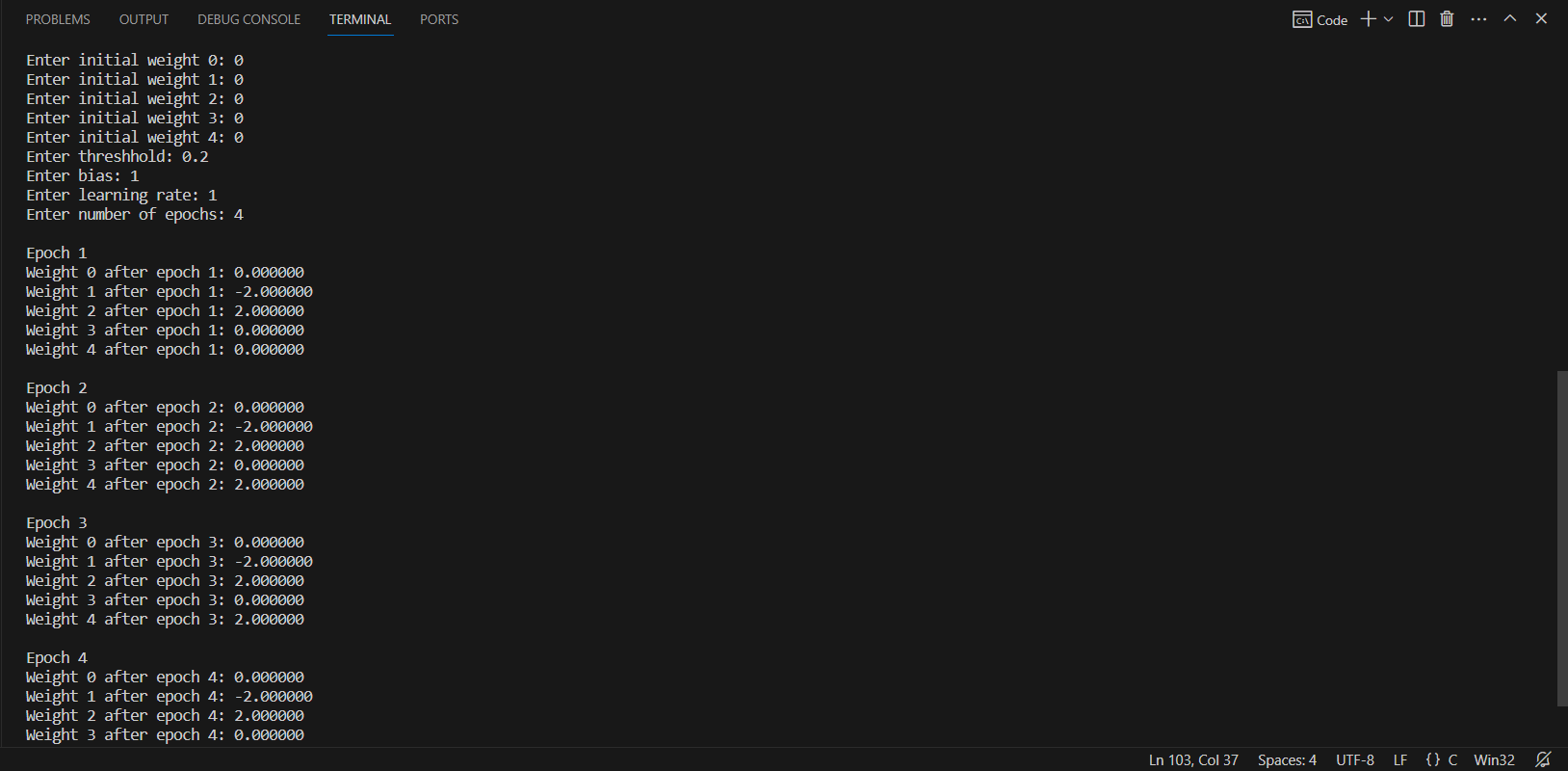
        }

    }

}

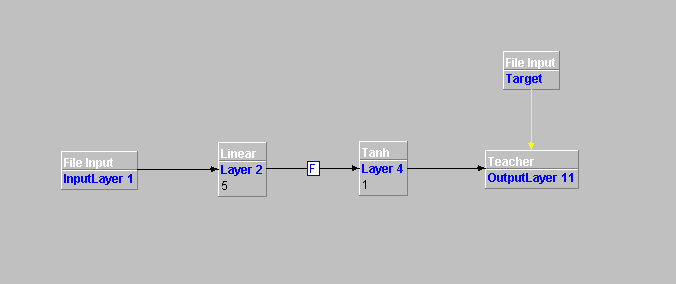
**Results:**



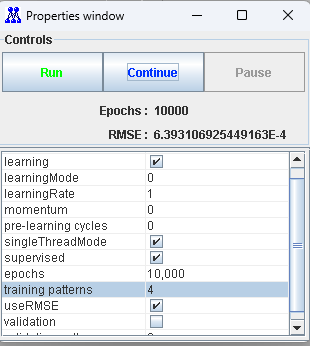
****

**Joone Editor:-**

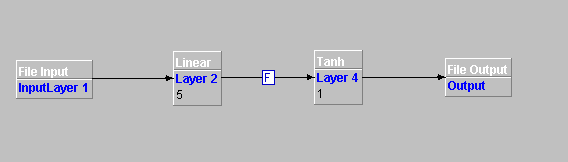
Step 1: Draw a network with a teacher in it.



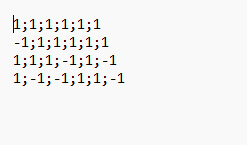
Step 2: Go to the control panel and do the required changes.



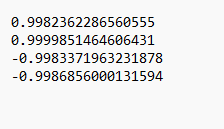
Step 3: After training the layers we test it by forming the model.



Step 4: Give the input file.



Step 5: Make changes in the control panel run the model and then store the output in the file.



Here we observe that our Joone model is correct and works perfectly. It adjusted its weights so as to meet the target values.

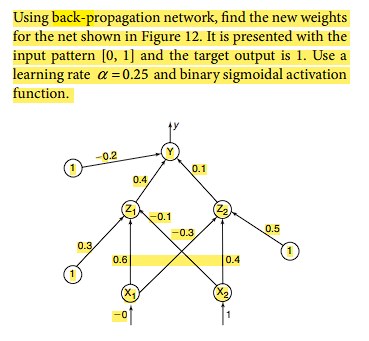
**CONCLUSION: -** In this experiment we studied about the perceptron model and joone editor and how to use it.

|  |  |
| --- | --- |
| **Name:** | Bodhisatya Ghosh |
| **Branch:** | CSE – Data Science |
| **Batch:** | B |
| **Course:** | Soft Computing |
| **Experiment no:** | 3 |

**Aim:** To implement Multilayer Perceptron Learning algorithm (EBPTA).

**Theory:** This learning algorithm is applied to multilayer feed-forward networks consisting of processing elements with continuous differentiable activation functions. The networks associated with back-propagation learning algorithm are also called back-propagation networks (BPNs). For a given set of training input-output pair, this algorithm provides a procedure for changing the weights in a BPN to classify the given input patterns correctly. The basic concept for this weight update algorithm is simply the gradient-descent method as used in the case of simple perceptron networks with differentiable units.

This is a method where the error is propagated back to the hidden unit. The aim of the neural network is to train the net to achieve a balance between the net’s ability to respond (memorization) and its ability to give reasonable responses to the input that is similar but not identical to the one that is used in training (generalization)

****

**Program:**

#include <stdio.h>

#include <stdlib.h>

#include <math.h>

void print\_neurons(float\* arr,int num,char\* c){

    printf("%s\n",c);

    for(int i = 0; i < num ; i++){

        printf("%f ",arr[i]);

    }

    printf("\n\n");

}

void print\_weights(float\*\* arr,int row, int col){

    printf("Weights are :\n");

    for(int i = 0; i < row ; i++){

        for(int j = 0; j < col; j++){

            printf("%f ",arr[i][j]);

        }

        printf("\n");

    }

    printf("\n");

}

void print\_bias(float\* bias,int neuron\_cnt,char\* c){

    printf("%s\n",c);

    for(int i = 0; i < neuron\_cnt; i++){

        printf("%f ",bias[i]);

    }

    printf("\n\n");

}

float\* allocate\_neuron(float\* arr,int size,char\* desc){

    printf("Allocating %d %s\n",size,desc);

    arr = (float\*)malloc(size \* sizeof(float));

    return arr;

}

float\*\* fill\_weights(float\*\* weights,int i,int j,char\* desc){

    weights = (float\*\*)malloc(i \* sizeof(float\*));

    for(int x = 0; x<j ; x++){

        weights[x] = (float\*)malloc(j \* sizeof(float));

    }

    printf("Weights from %s\n",desc);

    for(int a = 0; a < i; a++){

        for(int b = 0; b < j; b++){

            printf("Enter weight for neuron %d to neuron %d: ",a+1,b+1);

            scanf("%f",&weights[a][b]);

        }

    }

    printf("\n");

    return weights;

}

float activation\_function(float x){

    float ans = (float)(1/(1+exp(-x)));

    return ans;

}

float diff\_activation\_function(float x){

    float ans = activation\_function(x)\*(1-activation\_function(x));

    return ans;

}

void feedforward(float\* layer1,float\* layer2,float\* layer2\_in,int layer1\_size,int layer2\_size, float\* bias, float\*\* weights){

    printf("Feeding forward\n");

    for(int i = 0 ; i < layer2\_size; i++){

        float zin = bias[i];

        for(int j = 0; j < layer1\_size; j++){

            zin += layer1[j]\*weights[j][i];

        }

        layer2[i] = activation\_function(zin);

        layer2\_in[i] = zin;

        printf("Finished one neuron to neuron\n\n");

    }

}

void backpropagate\_final(float\*y, float\* yin,int neuron\_count,float\* error\_sigma,float target){

    for(int i = 0; i < neuron\_count; i ++){

        error\_sigma[i] = (target - y[i]) \* diff\_activation\_function(yin[i]);

    }

}

void backpropagate\_hidden(float\* hidden\_layer\_error\_sigma,float\* output\_layer\_error\_sigma,float\* hidden\_layer\_input,float\*\* weight,int hidden\_neurons,int output\_neurons){

    for(int i = 0 ;i < output\_neurons;i++){

        for(int j = 0; j < hidden\_neurons; j++){                        // As error sigma was not initilaized, this will not work for nets with

            hidden\_layer\_error\_sigma[j] = output\_layer\_error\_sigma[i] \* weight[j][i];      //more than one output neuron

        }

    }

    for(int i = 0; i<hidden\_neurons; i++){

        hidden\_layer\_error\_sigma[i] = hidden\_layer\_error\_sigma[i] \* diff\_activation\_function(hidden\_layer\_input[i]);

    }

}

void update\_weights(float\*\* weight,float\* input,int row,int col,float\* error\_sigma,float learning\_rate,float\* bias){

    for(int i = 0; i < col; i++){

        for(int j = 0; j < row; j++){

            weight[j][i] = weight[j][i] + (learning\_rate \* error\_sigma[i] \* input[j]);

        }

        bias[i] = bias[i] + (error\_sigma[i] \* learning\_rate);

    }

}

void main()

{

    float\* x;       //input

    float\*\* v;      //input to hidden weights

    float\* z;       //hidden

    float\* zin;       //hidden

    float\*\* w;      //hidden to output

    float\* y;       //output

    float\* yin;       //output

    float\* output\_error;    //backpropagating error from output neuron(s)

    float\* hidden\_error;    //backpropagating error from hidden neuron(s)

    float bias\_hidden[] = {0.3,0.5};  //biases

    float bias\_output[] = {-0.2};

    int n,p,m;      // neuron numbers

    float learning\_rate,target;     //learning rate and target

    printf("Enter number of input neurons: ");

    scanf("%d",&n);

    printf("Enter number of hidden layer neurons: ");

    scanf("%d",&p);

    printf("Enter number of output neurons: ");

    scanf("%d",&m);

    printf("Enter learning rate: ");

    scanf("%f",&learning\_rate);

    printf("Enter target: ");

    scanf("%f",&target);

    printf("\n");

    x = allocate\_neuron(x,n,"input layer neurons");

    zin = allocate\_neuron(zin,p,"hidden layer neurons inputs");

    z = allocate\_neuron(z,p,"hidden layer neurons");

    yin = allocate\_neuron(yin,m,"output layer neurons inputs");

    y = allocate\_neuron(y,m,"output layer neurons");

    hidden\_error = allocate\_neuron(hidden\_error,p,"error propagation for hidden layer");

    output\_error = allocate\_neuron(output\_error,m,"error propagation for output layer");

    for(int i = 0; i < n; i++){

        printf("Enter input neuron %d: ",i+1);

        scanf("%f",&x[i]);

    }

    print\_neurons(x,n,"Input neurons");

    v = fill\_weights(v,n,p,"input layer to hidden layer");

    w = fill\_weights(w,p,m,"hidden layer to output layer");

    feedforward(x,z,zin,n,p,bias\_hidden,v);

    print\_neurons(z,p,"Hidden neurons");

    feedforward(z,y,yin,p,m,bias\_output,w);

    print\_neurons(y,m,"Output neurons");

    backpropagate\_final(y,yin,m,output\_error,target);

    backpropagate\_hidden(hidden\_error,output\_error,zin,v,p,m);

    update\_weights(w,z,p,m,output\_error,learning\_rate,bias\_output);

    update\_weights(v,x,n,p,hidden\_error,learning\_rate,bias\_hidden);

    printf("V ");

    print\_weights(v,n,p);

    print\_bias(bias\_hidden,p,"Hidden layer bias");

    printf("W: ");

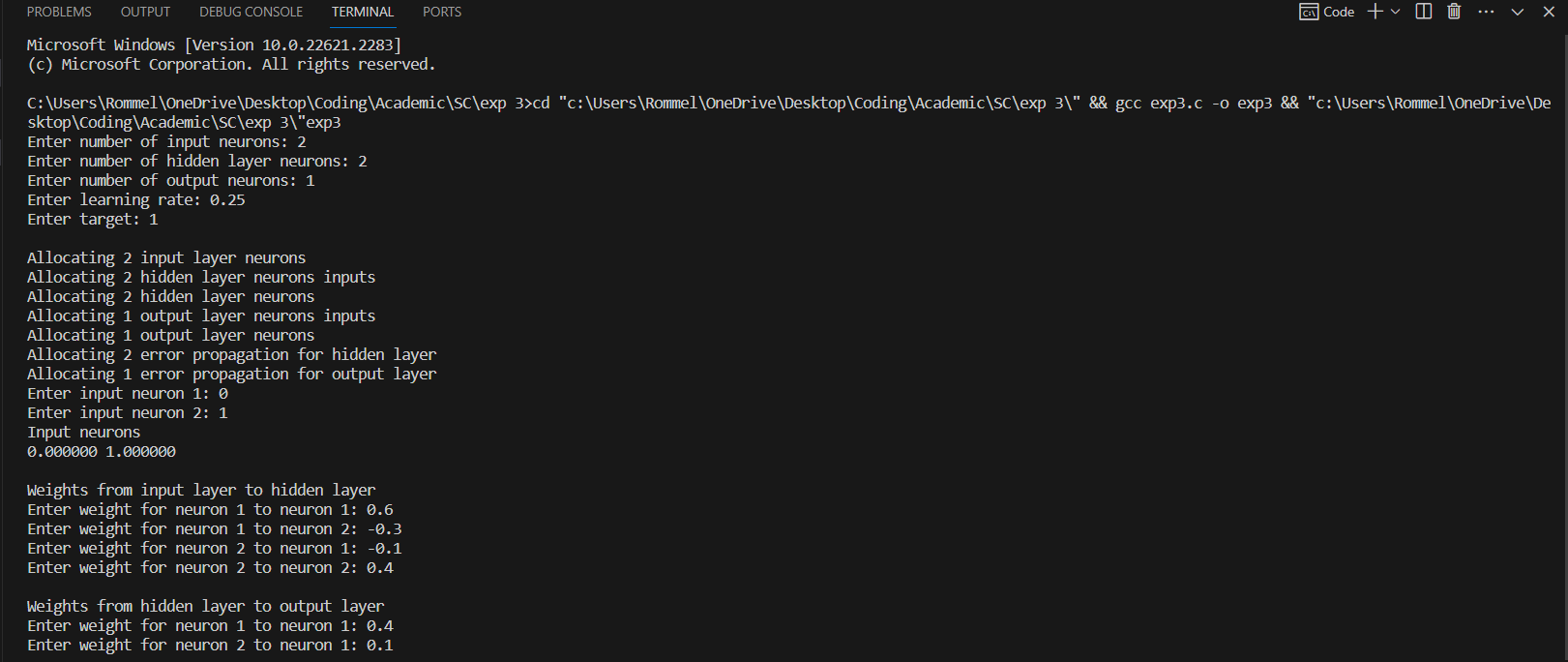
    print\_weights(w,p,m);

    print\_bias(bias\_output,m,"Output layer bias");

}

//2 2 1 0.25 1 0 1 0.6 -0.3 -0.1 0.4 0.4 0.1

**Results:**

****



**CONCLUSION: -** In this experiment we studied about multilayer perceptron learning algorithm with back-propagation.

|  |  |
| --- | --- |
| **Name:** | Bodhisatya Ghosh |
| **Branch:** | CSE – Data Science |
| **Batch:** | B |
| **Course:** | Soft Computing |
| **Experiment no:** | 4 |

**Aim:** To implement KSOFM for a given pattern classification problem.

**Theory:** Feature mapping is a process which converts the patterns of arbitrary dimensionality into a response of one- or twodimensional arrays of neurons, i.e., it converts a wide pattern space into a typical feature space. The network performing such a mapping is called feature map. Apart from its capability to reduce the higher dimensionality, it has to preserve the neighbourhood relations of the input patterns, i.e., it has to obtain a topology preserving map. For obtaining such feature maps, it is required to find a self-organizing neural array which consists of neurons arranged in a one-dimensional array or a two-dimensional array. At the time of self-organization, the weight vector of the cluster unit which matches the input pattern very closely is chosen as the winner unit. The closeness of weight vector of cluster unit to the input pattern may be based on the square of the minimum Euclidean distance. The weights are updated for the winning unit and its neighboring units. It should be noted that the weight vectors of the neighboring units are not close to the input pattern and the connective weights do not multiply the signal sent from the input units to the cluster units until dot product measure of similarity is being used.

**Program:**

#include <stdio.h>

#include <stdlib.h>

#include <math.h>

void print\_array(float\* arr,int num,char\* c){

    printf("%s",c);

    for(int i = 0; i < num ; i++){

        printf("%f ",arr[i]);

    }

    printf("\n");

}

void print\_array\_int(int\* arr,int num,char\* c){

    printf("%s",c);

    for(int i = 0; i < num ; i++){

        printf("%d ",arr[i]);

    }

    printf("\n");

}

void print\_weights(float\*\* arr,int row, int col){

    for(int i = 0; i < row ; i++){

        for(int j = 0; j < col; j++){

            printf("%f ",arr[i][j]);

        }

        printf("\n");

    }

    printf("\n");

}

float\* allocate\_1Dmem(float\* arr,int size,char\* desc){

    printf("Allocating memory for %s\n",desc);

    arr = (float\*)malloc(size \* sizeof(float));

    return arr;

}

int\* allocate\_1Dmem\_int(int\* arr,int size,char\* desc){

    printf("Allocating memory for %s\n",desc);

    arr = (int\*)malloc(size \* sizeof(int));

    return arr;

}

float\*\* fill\_weights(float\*\* weights,int i,int j,char\* desc){

    printf("Allocating %s\n",desc);

    weights = (float\*\*)malloc(i \* sizeof(float\*));

    for(int x = 0; x<j ; x++){

        weights[x] = (float\*)malloc(j \* sizeof(float));

    }

    for(int a = 0; a < i; a++){

        for(int b = 0; b < j; b++){

            printf("Enter weight for cluster %d to vector %d: ",a+1,b+1);

            scanf("%f",&weights[a][b]);

        }

    }

    printf("\n");

    return weights;

}

float\*\* allocate\_2Dmem(float\*\* vector,int i,int j,char\* desc){

    printf("Allocating memory for %s\n",desc);

    vector = (float\*\*)malloc(i \* sizeof(float\*));

    for(int x = 0; x<j ; x++){

        vector[x] = (float\*)malloc(j \* sizeof(float));

    }

    for(int a = 0; a < i; a++){

        printf("For vector %d\n",a+1);

        for(int b = 0; b < j; b++){

            printf("Enter value %d: ",b+1);

            scanf("%f",&vector[a][b]);

        }

    }

    printf("\n");

    return vector;

}

int calc\_winning\_cluster(float\* vector, float\*\* weights, int clusters, int vect\_cnt){

    float dist[] = {0,0};

    for(int i = 0; i < clusters; i++){

        for(int j = 0; j < vect\_cnt; j++){

            dist[i] = dist[i] + pow((weights[i][j]-vector[j]),2);

        }

    }

    if(dist[0] >= dist[1]){

        return 1;

    }

    else{

        return 0;

    }

}

float\* update\_weights(float\* weights, float\* vector,float learning\_rate, int vector\_len){

    print\_array(vector,vector\_len,"Vector: ");

    for(int i = 0; i < vector\_len; i++){

        weights[i] = weights[i] + learning\_rate \* (vector[i] - weights[i]);

        // printf("Weight from vector %d is %f:",i,weights[i]);

        // printf("\n");

    }

    print\_array(weights,vector\_len,"Updated cluster weight: ");

    return weights;

}

float\*\* KSOFM(float\*\* vectors, float\*\* weights, int\* winning\_cluster, int num\_of\_vectors, int num\_of\_clusters, float learning\_rate, int vect\_len){

    for (int i = 0; i < num\_of\_vectors; i++)

    {

        printf("\nTraining on vector %d\n",i+1);

        winning\_cluster[i] = calc\_winning\_cluster(vectors[i], weights, num\_of\_clusters, num\_of\_vectors);

        weights[winning\_cluster[i]] = update\_weights(weights[winning\_cluster[i]],vectors[i],learning\_rate,vect\_len);

    }

    return weights;

}

void main()

{

    int\* winning\_cluster;     //winning cluster info

    float\*\* vectors;       //input vectors

    float\*\* weights;      //vector to cluster weight

    int vect\_len,input,clusters;      // vector len, vector num, cluster num

    float learning\_rate;     //learning rate and target

    printf("Enter number of vectors: ");

    scanf("%d",&input);

    printf("Enter number of elements per vector: ");

    scanf("%d",&vect\_len);

    vectors = allocate\_2Dmem(vectors,input,vect\_len,"vectors");

    printf("Enter number of clusters: ");

    scanf("%d",&clusters);

    winning\_cluster = allocate\_1Dmem\_int(winning\_cluster,input,"winning cluster info");

    weights = fill\_weights(weights,clusters,input,"weights");

    printf("Enter learning rate: ");

    scanf("%f",&learning\_rate);

    printf("\n");

    weights = KSOFM(vectors, weights, winning\_cluster, input, clusters, learning\_rate, vect\_len);

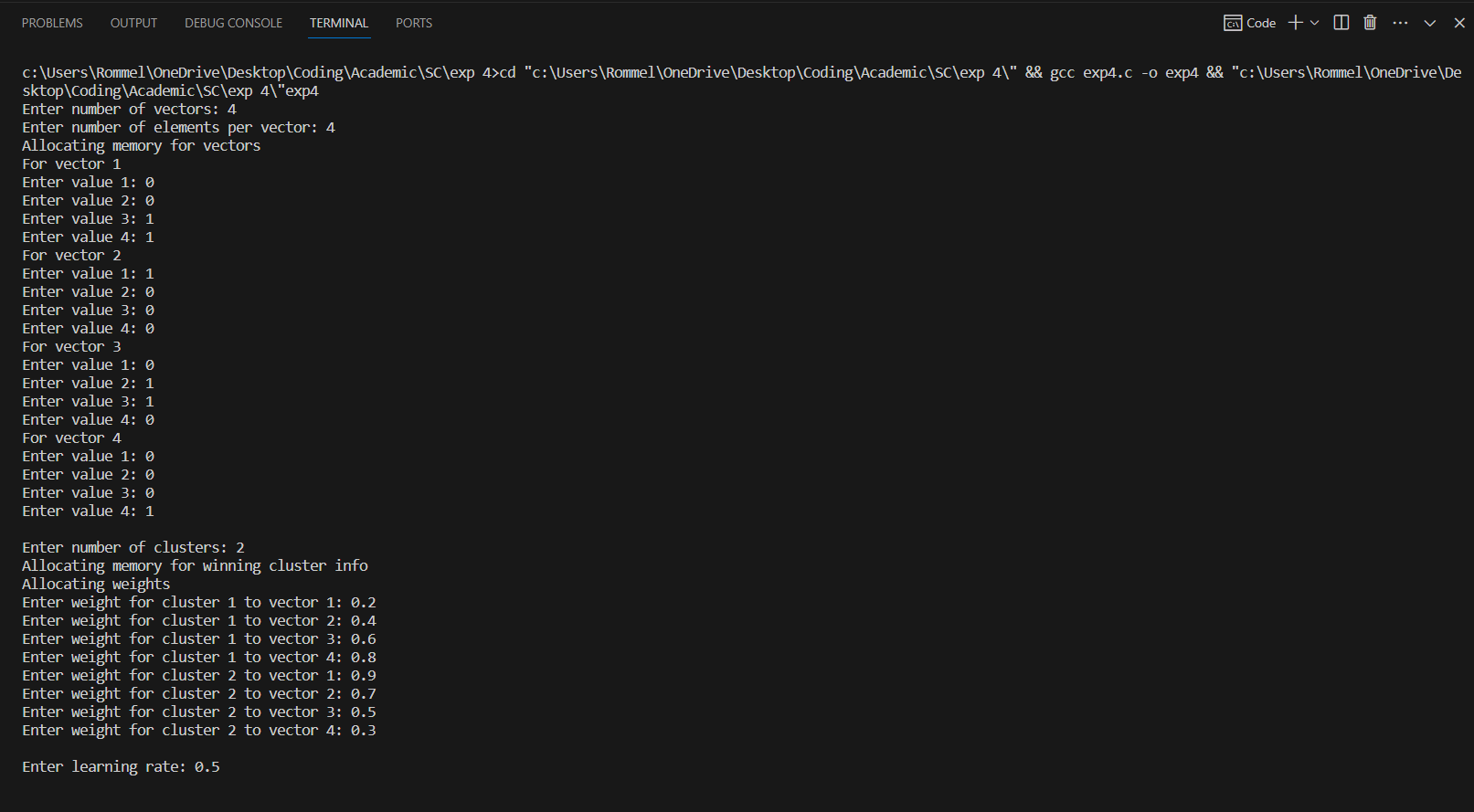
    printf("\nFinal weights are : \n");

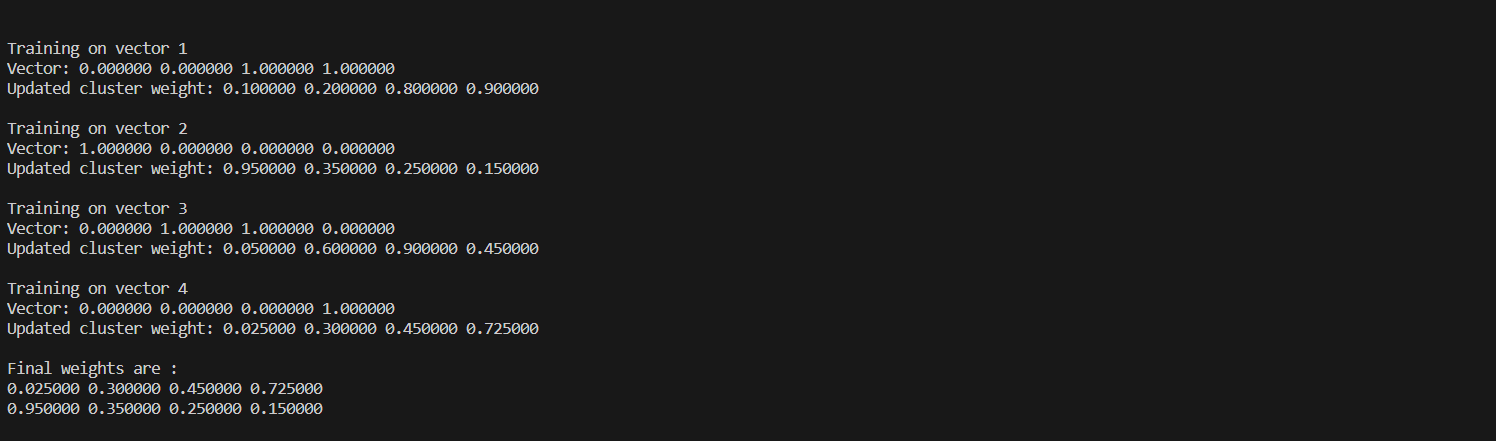
    print\_weights(weights,clusters,input);

}

//4 4 0 0 1 1 1 0 0 0 0 1 1 0 0 0 0 1 2 0.2 0.4 0.6 0.8 0.9 0.7 0.5 0.3 0.5

**Results:**





**CONCLUSION: -** In this experiment we studied about unsupervised learning using Kohonen Self-Organising Feature Maps

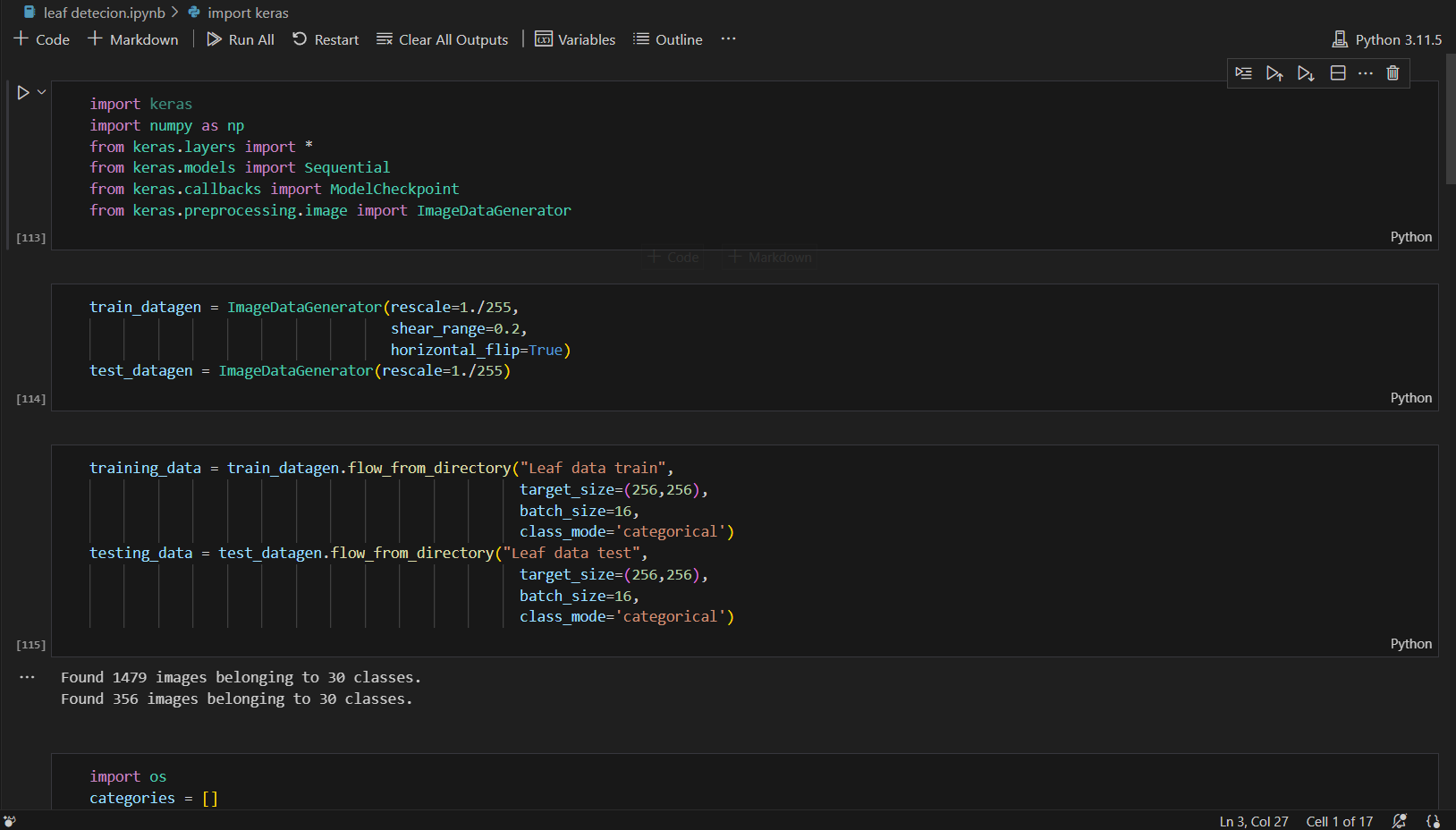
|  |  |
| --- | --- |
| **Name:** | Bodhisatya Ghosh |
| **Branch:** | CSE – Data Science |
| **Batch:** | B |
| **Course:** | Soft Computing |
| **Experiment no:** | 5 |

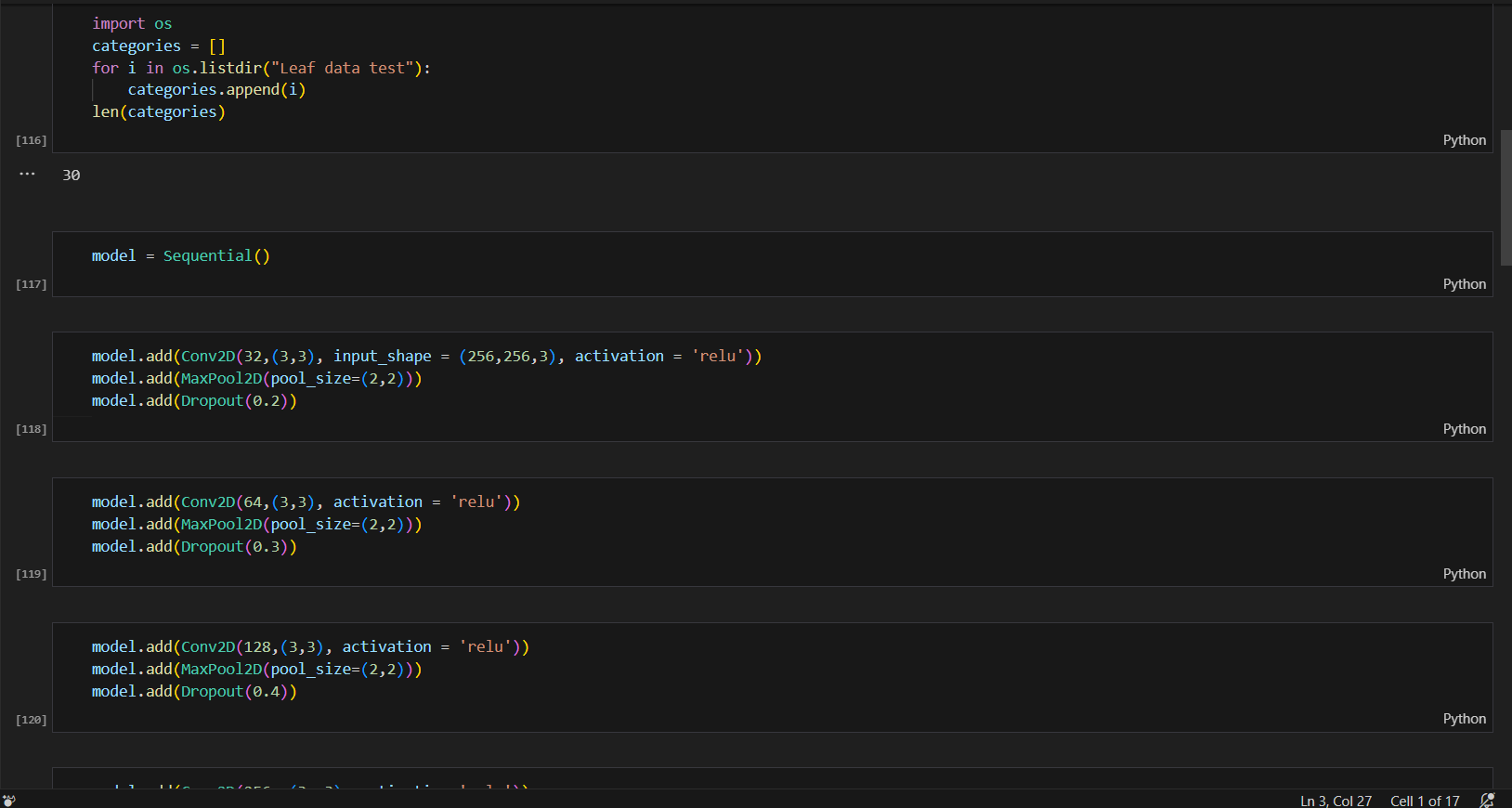
**Aim:** To implement a CNN for a given problem statement.

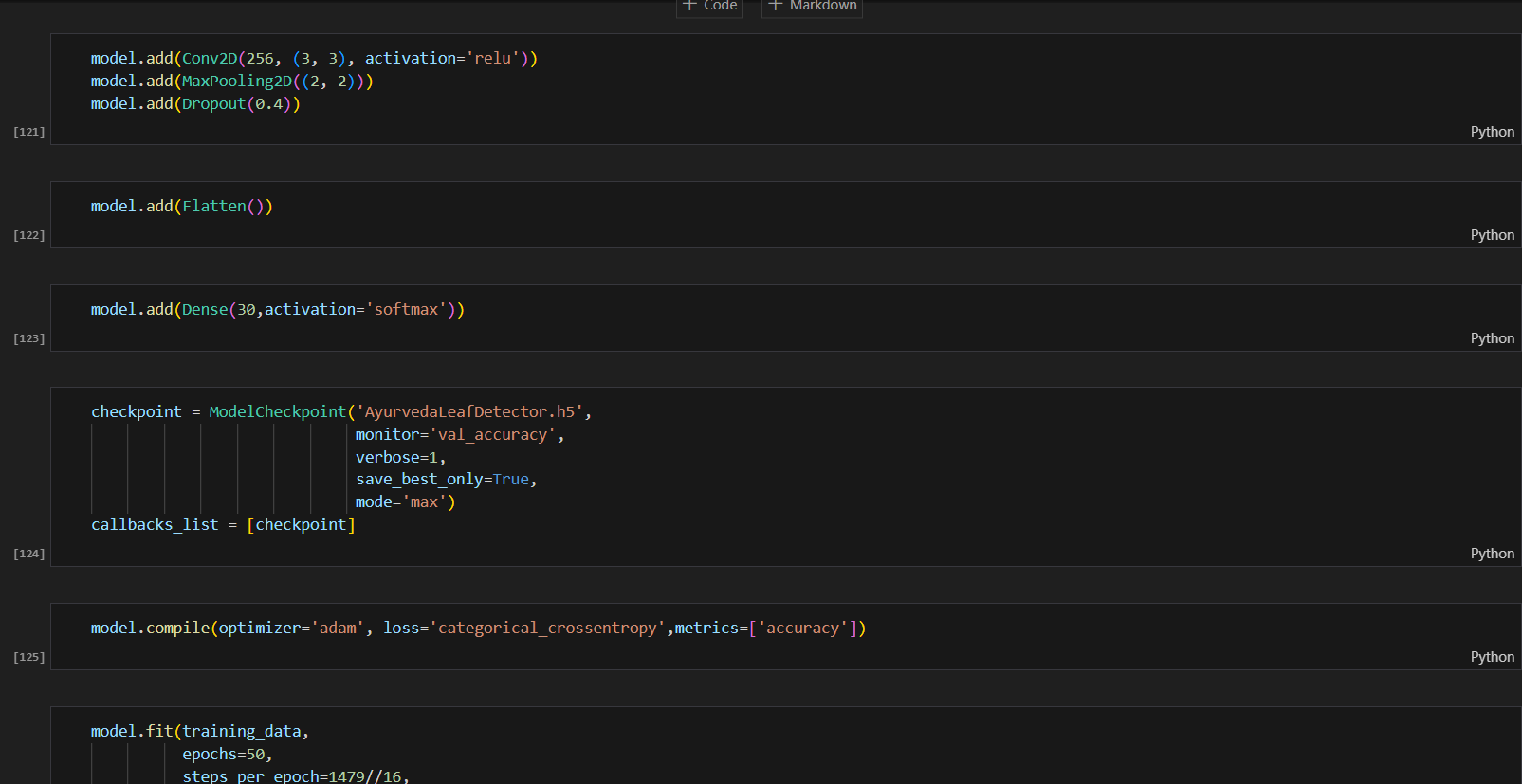
**Theory:** A **Convolutional Neural Network (CNN)** is a type of Deep Learning neural network architecture commonly used in Computer Vision. Computer vision is a field of Artificial Intelligence that enables a computer to understand and interpret the image or visual data. Convolutional Neural Network (CNN) is the extended version of [artificial neural networks (ANN)](https://www.geeksforgeeks.org/artificial-neural-networks-and-its-applications/) which is predominantly used to extract the feature from the grid-like matrix dataset. For example visual datasets like images or videos where data patterns play an extensive role.

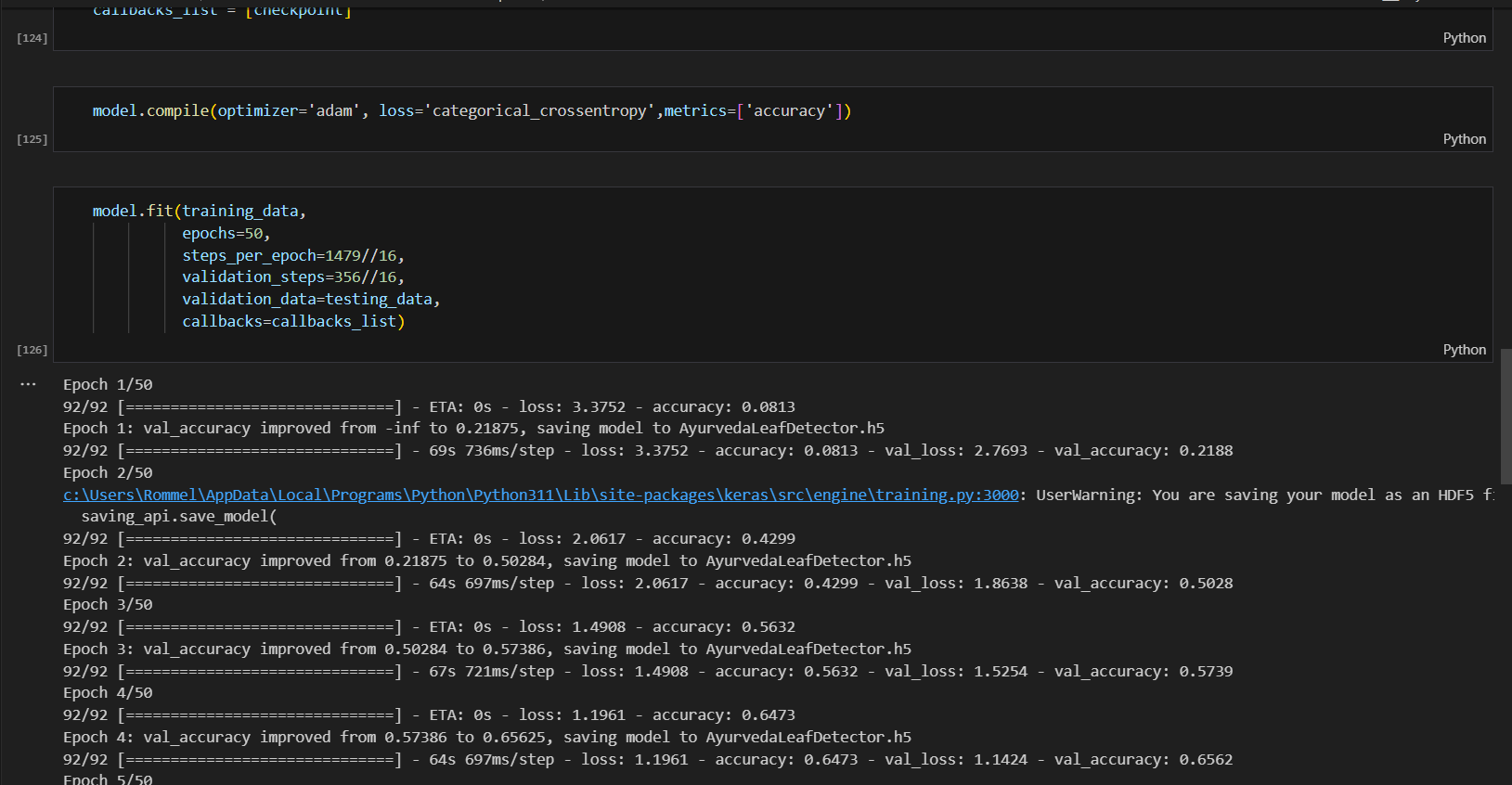


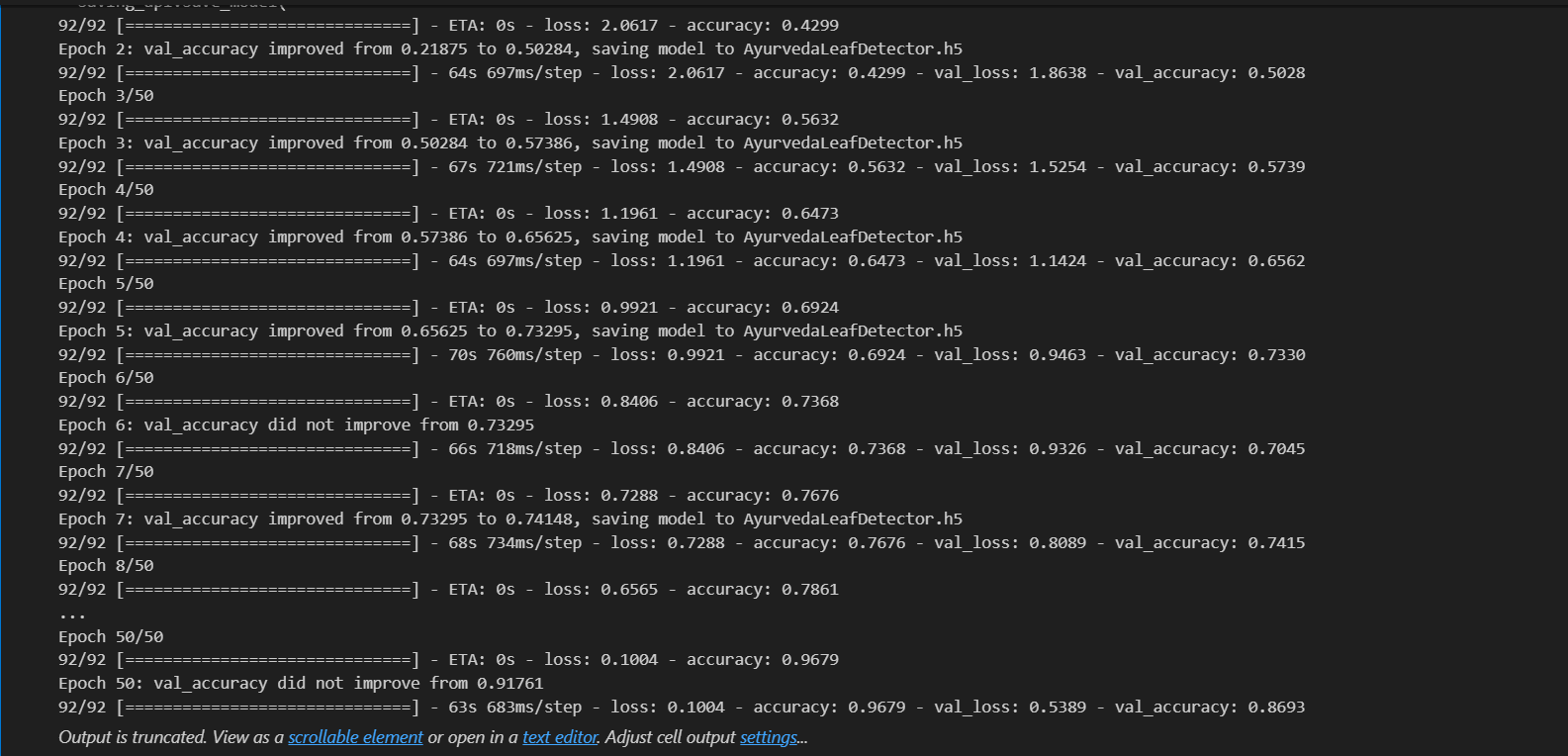
The Convolutional layer applies filters to the input image to extract features, the Pooling layer down samples the image to reduce computation, and the fully connected layer makes the final prediction. The network learns the optimal filters through backpropagation and gradient descent.

**Program:** 

****

****

****

****

**Results:**

****

**CONCLUSION: -** In this experiment we studied about unsupervised learning using Kohonen Self-Organising Feature Maps

|  |  |
| --- | --- |
| **Name:** | Bodhisatya Ghosh |
| **Branch:** | CSE – Data Science |
| **Batch:** | B |
| **Course:** | Soft Computing |
| **Experiment no:** | 6 |

**Aim:** To calculate standard deviation, root mean square, mean absolute error etc for measuring the fitness of a model.

**Theory:** Linear regression analysis is used to predict the value of a variable based on the value of another variable. The variable you want to predict is called the dependent variable. The variable you are using to predict the other variable's value is called the independent variable.

This form of analysis estimates the coefficients of the linear equation, involving one or more independent variables that best predict the value of the dependent variable. Linear regression fits a straight line or surface that minimizes the discrepancies between predicted and actual output values. There are simple linear regression calculators that use a “least squares” method to discover the best-fit line for a set of paired data. You then estimate the value of X (dependent variable) from Y (independent variable).

**Program:**

# %%

import numpy as np

import pandas as pd

import seaborn as sns

# %%

data = pd.read\_csv('car price.csv')

data.drop(columns=['Unnamed: 0'],inplace=True)

data.head()

# %%

data.info()

# %%

data.describe()

# %%

data.isna().sum()

# %%

x = data

y = data['Price']

categorical = pd.DataFrame()

# %%

from sklearn.preprocessing import LabelEncoder

# %%

label = LabelEncoder()

for i in x.columns:

    if(x[i].dtype == 'object'):

        x[i] = label.fit\_transform(x[i])

        categorical[i] = x[i]

categorical

# %%

x.head()

# %%

sns.heatmap(x.corr())

# %%

from scipy import stats

for i in categorical.columns:

    stats.pointbiserialr(categorical[i], y)

    print(i +' '+ str(stats.pointbiserialr(categorical[i], y)[0]))

# %%

x.drop(columns=['Company Name','Model Name','Location','Engine Type','Color'],inplace=True)

x.head()

# %%

sns.heatmap(x.corr())

# %%

x.drop(columns=['Price'],inplace = True)

# %%

x.head()

# %%

from sklearn.preprocessing import StandardScaler

scale = StandardScaler()

linearVar = [['Mileage','Engine Capacity']]

x[['Mileage','Engine Capacity']] = scale.fit\_transform(x[['Mileage','Engine Capacity']])

# %%

from sklearn.preprocessing import PolynomialFeatures

poly = PolynomialFeatures(3)

x = poly.fit\_transform(x)

# %%

x

# %%

from sklearn.model\_selection import train\_test\_split

xtrain,xtest,ytrain,ytest = train\_test\_split(x,y,test\_size=0.4,random\_state=100)

# %%

from sklearn.linear\_model import LinearRegression

model = LinearRegression()

model.fit(xtrain,ytrain)

result = model.fit(xtrain,ytrain)

# %%

ypred = model.predict(xtest)

# %%

ax1 = sns.kdeplot(ytest, color = 'r', label = 'Actual Value')

sns.kdeplot(ypred, color = 'b', label = 'Fitted Value', ax = ax1)

# %%

from sklearn.metrics import mean\_squared\_error,mean\_absolute\_error,r2\_score

resid = ytest - ypred

std\_dev = np.std(resid)

rmse = np.sqrt(mean\_squared\_error(ytest,ypred))

mae = mean\_absolute\_error(ytest,ypred)

r2 = r2\_score(ytest,ypred)

# %%

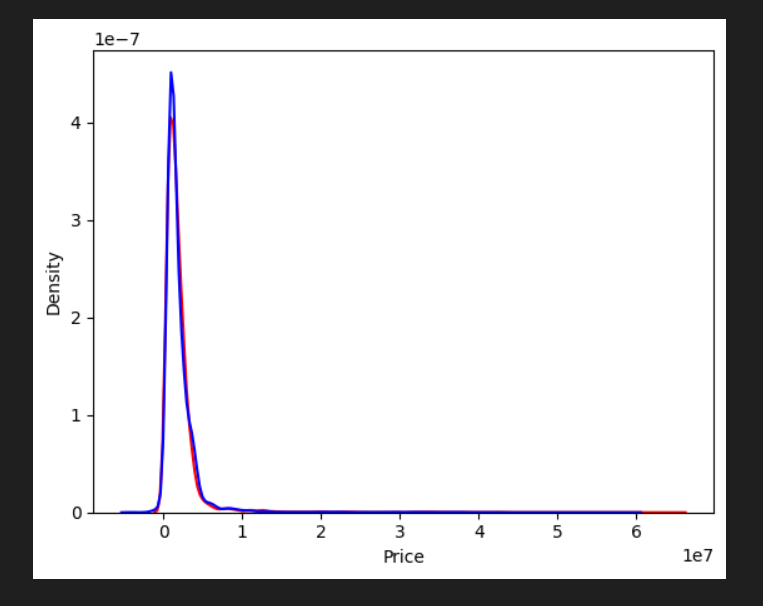
print('Standard Deviation: ' + str(std\_dev))

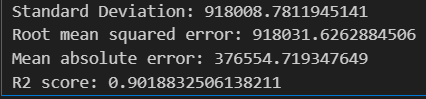
print('Root mean squared error: ' + str(rmse))

print('Mean absolute error: ' + str(mae))

print('R2 score: ' + str(r2))

**Results:**





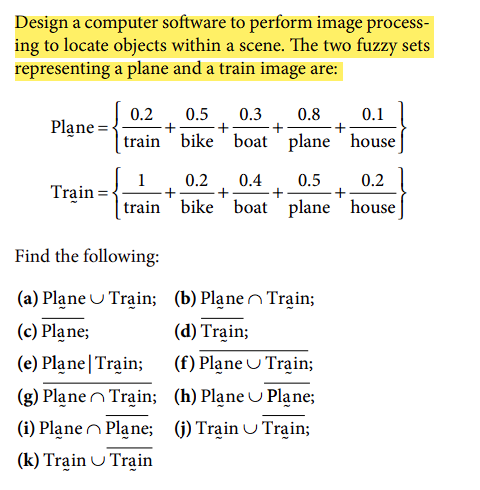
**CONCLUSION: -** In this experiment we studied about linear regression and have measured its metrics

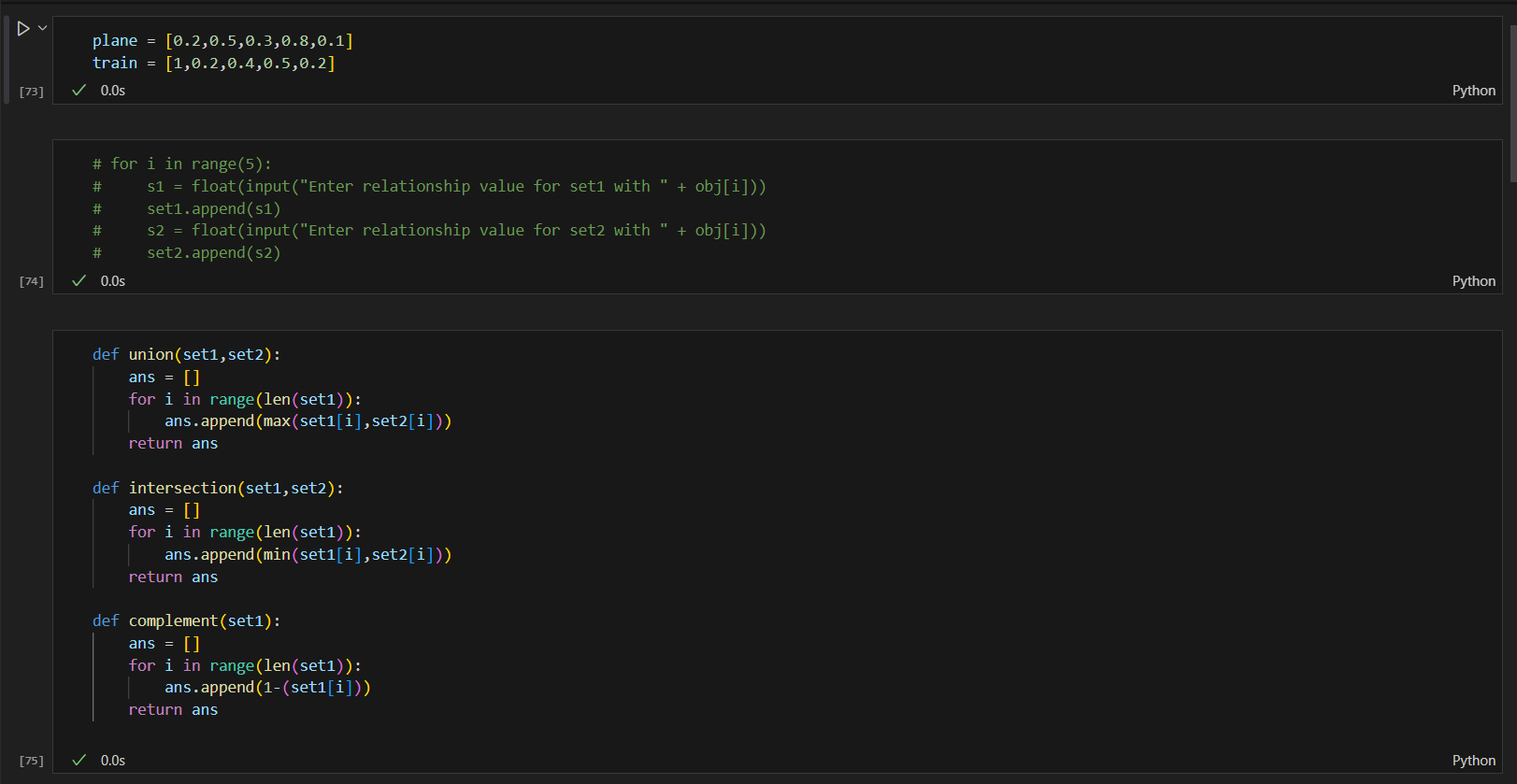
|  |  |
| --- | --- |
| **Name:** | Bodhisatya Ghosh |
| **Branch:** | CSE – Data Science |
| **Batch:** | B |
| **Course:** | Soft Computing |
| **Experiment no:** | 7 |

**Aim:** To implement fuzzy set and fuzzy relations for a given problem.

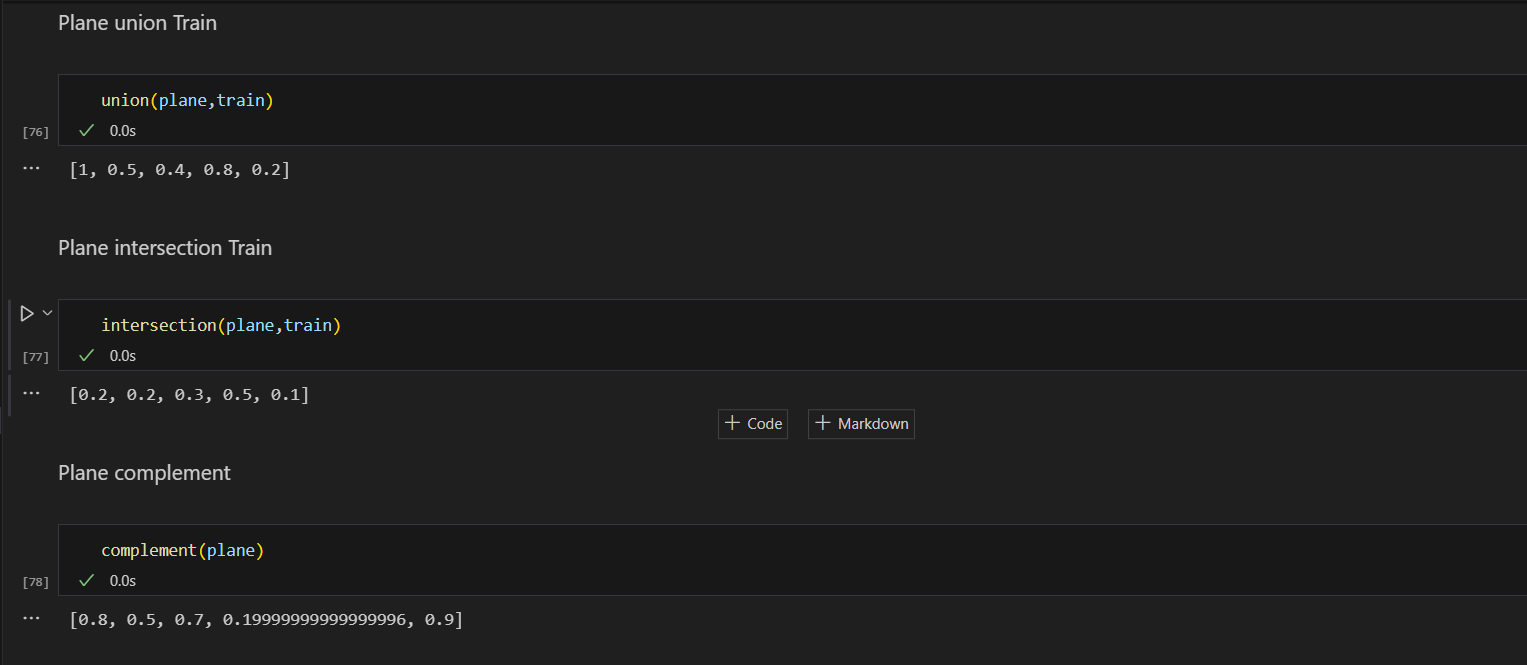
**Theory:** Linear regression analysis is used to predict the value of a variable based on the value of another variable. The variable you want to predict is called the dependent variable. The variable you are using to predict the other variable's value is called the independent variable.

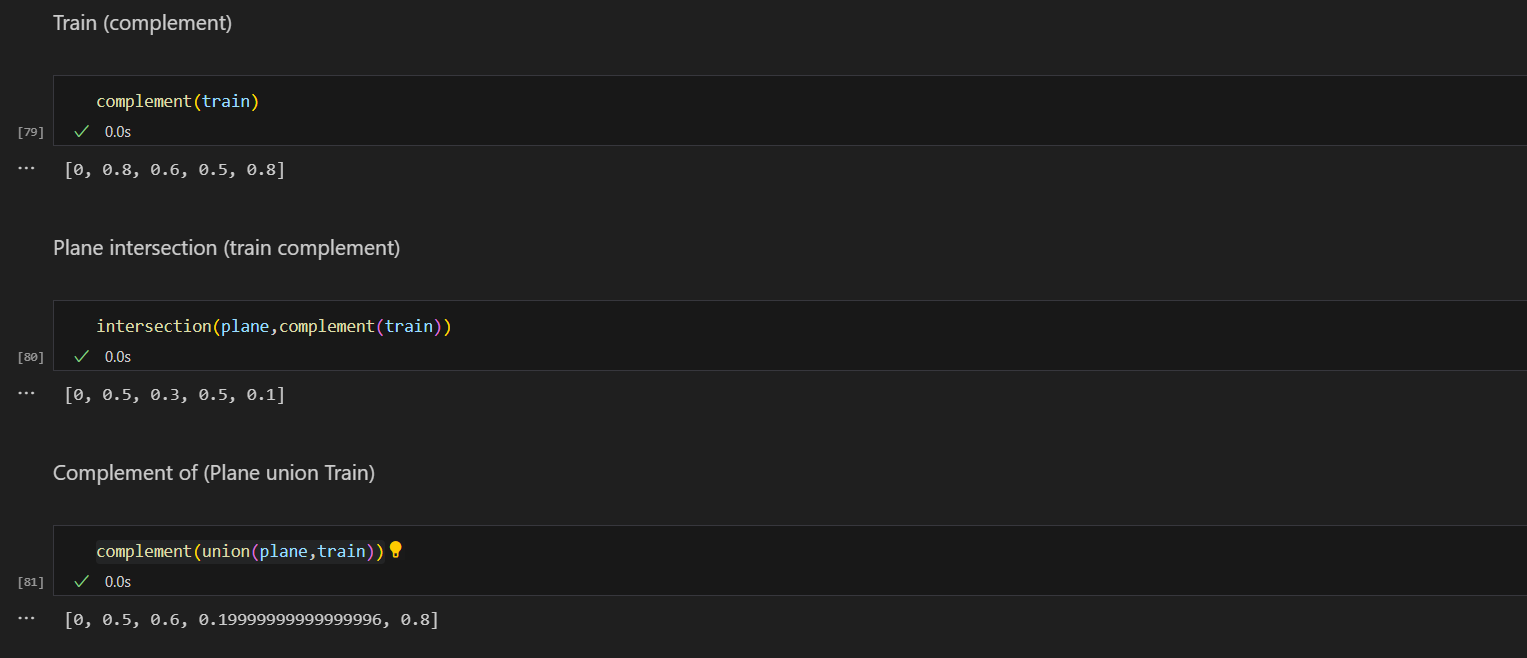
This form of analysis estimates the coefficients of the linear equation, involving one or more independent variables that best predict the value of the dependent variable. Linear regression fits a straight line or surface that minimizes the discrepancies between predicted and actual output values. There are simple linear regression calculators that use a “least squares” method to discover the best-fit line for a set of paired data. You then estimate the value of X (dependent variable) from Y (independent variable).

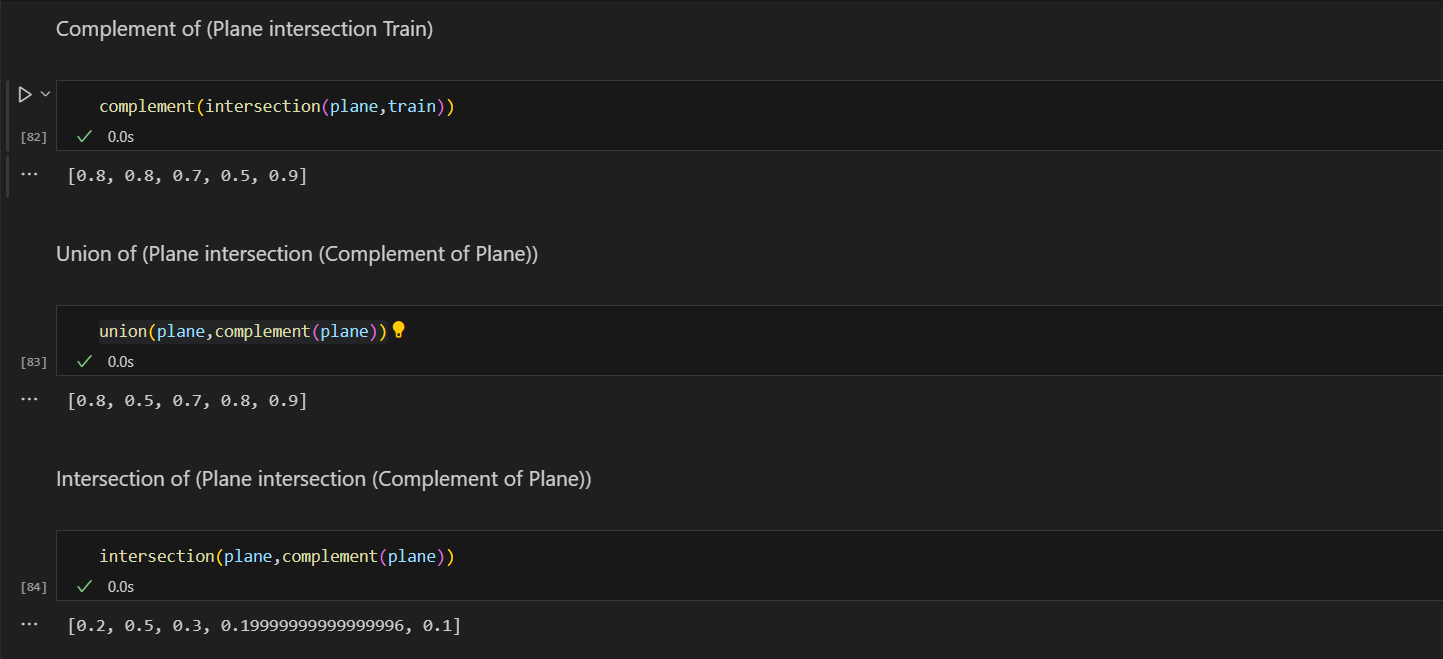


**Program:** ****

**Results:**









**CONCLUSION: -** In this experiment we studied about fuzzy sets and fuzzy relations

|  |  |
| --- | --- |
| **Name:** | Bodhisatya Ghosh |
| **Branch:** | CSE – Data Science |
| **Batch:** | B |
| **Course:** | Soft Computing |
| **Experiment no:** | 8 |

**Aim:** To design and implement fuzzy controller for a given problem.

**Theory:** A fuzzy controller is a type of control system that utilizes fuzzy logic to emulate human decision-making. Fuzzy logic is an extension of classical (or Boolean) logic that allows for a more nuanced and flexible approach to reasoning and decision-making in the presence of uncertainty. Fuzzy controllers are widely used in various applications, including industrial control systems, consumer electronics, and automated systems. Fuzzy controllers are used in a variety of applications, such as temperature control systems, speed control in vehicles, decision support systems, and more.

**REAL TIME APPLICATION USED:**

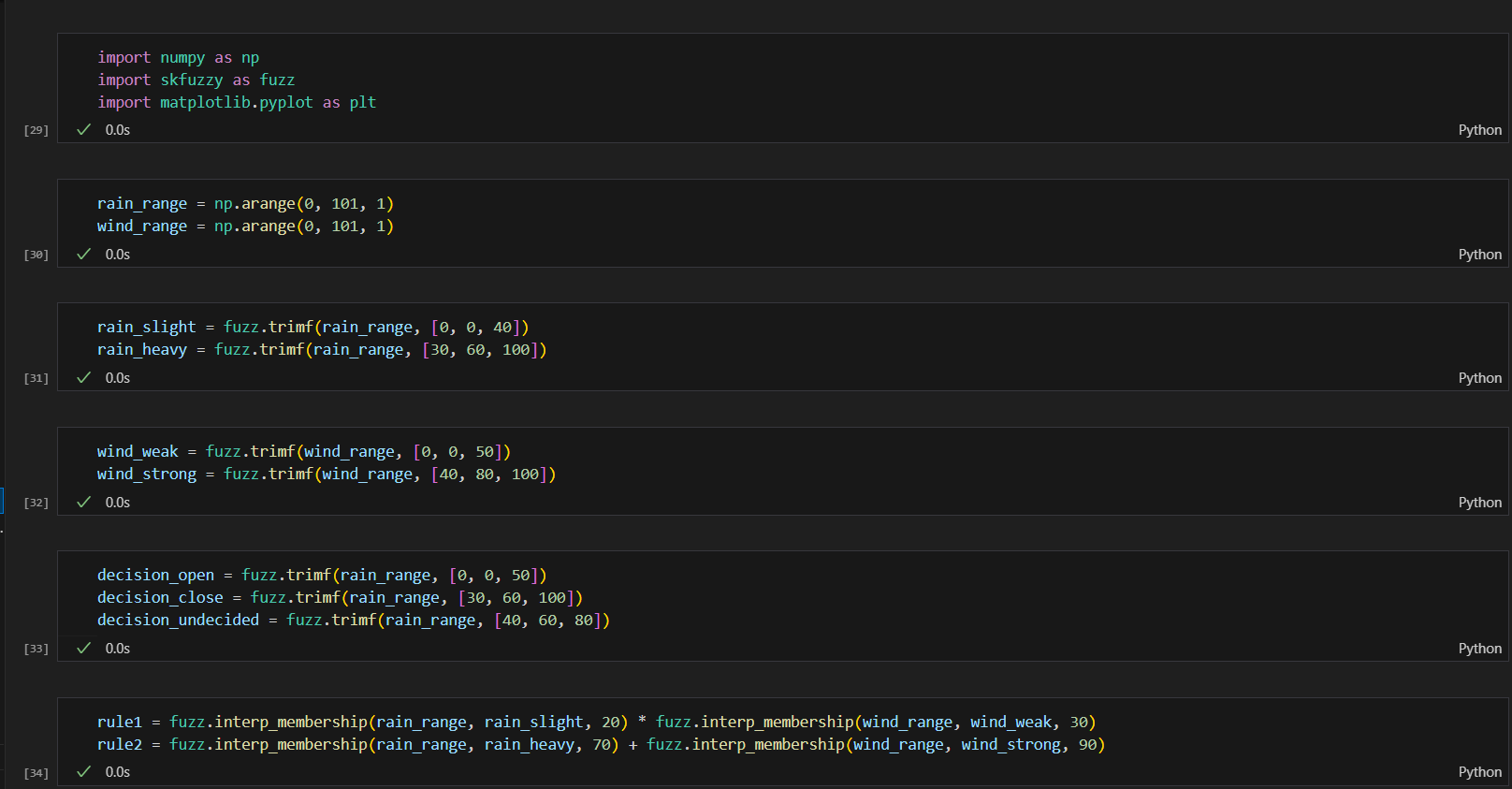
I have implemented a basic fuzzy logic system for deciding whether to open, close, or remain undecided about opening an umbrella based on rain and wind conditions.

1. Fuzzy Sets:

* You define fuzzy sets for "Rain" and "Wind," each with different degrees of membership. For example, "Slight Rain" and "Heavy Rain" are fuzzy sets for rain, and "Weak Wind" and "Strong Wind" are fuzzy sets for wind. These sets are defined using triangular membership functions.

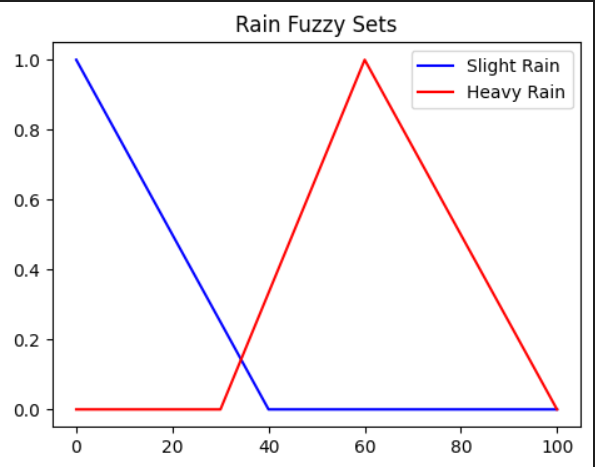
1. Fuzzification:
   * Fuzzification is the process of converting crisp inputs (rain and wind values) into fuzzy sets. The fuzz.trimf function is used to create triangular membership functions for rain and wind conditions.
2. Fuzzy Rules:
   * Two fuzzy rules are defined based on rain and wind conditions:
     + Rule 1: If rain is slight and wind is weak, then there is a preference to open the umbrella.
     + Rule 2: If rain is heavy or wind is strong, then there is a preference to close the umbrella.
3. Fuzzy Inference:
   * Fuzzy inference is the process of determining the degree to which each fuzzy rule contributes to the overall decision. This is done using the fuzz.interp\_membership function, which calculates the membership values for the input conditions in each rule.
4. Rule Activation:
   * The membership values obtained from fuzzification are used to activate each rule. These values represent the degree to which the conditions specified in each rule are satisfied.
5. Rule Combination:
   * The results of the two rules are combined using the np.fmax function, which takes the maximum degree of membership for each output value. This step determines the final fuzzy output for each possible decision.
6. Defuzzification:
   * The final fuzzy output is then defuzzified to obtain a crisp decision. The fuzz.defuzz function is used with the 'centroid' method, which calculates the center of mass of the combined fuzzy set.
7. Decision:
   * Based on the defuzzified result, a decision is made on whether to open the umbrella, close it, or remain undecided.

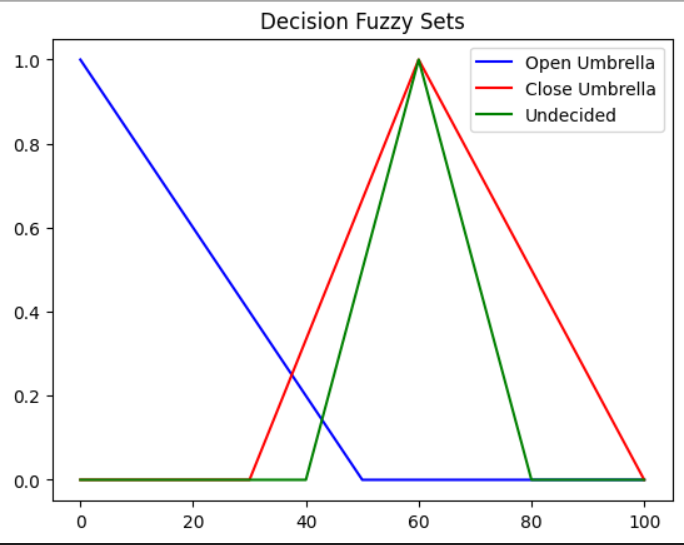
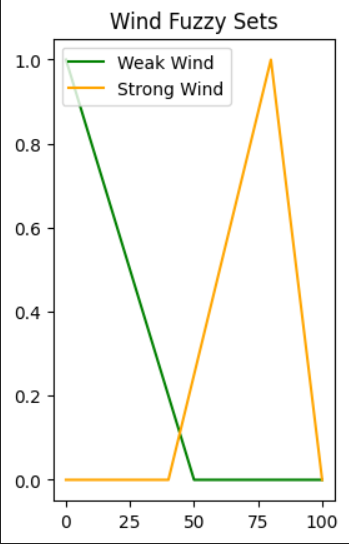
**Program:**

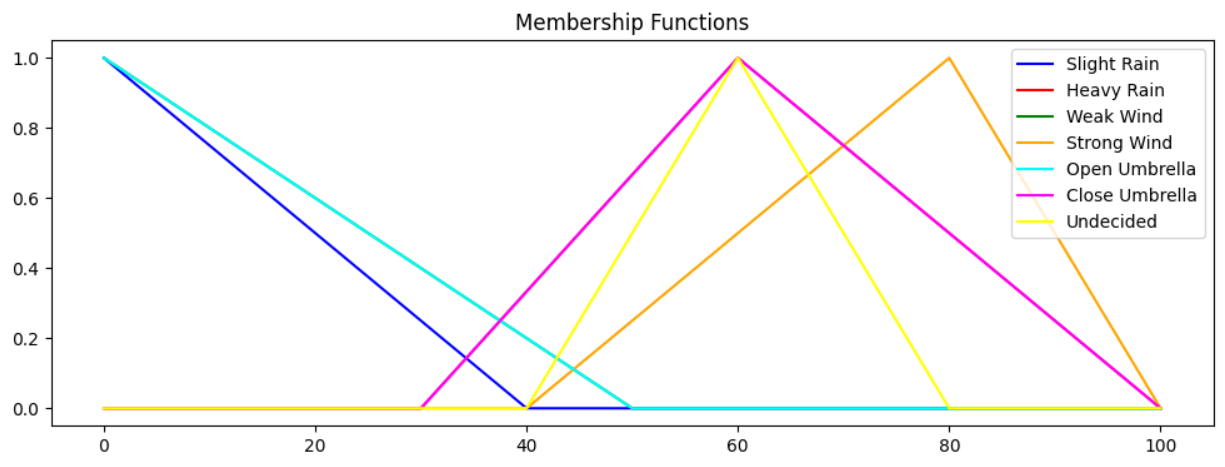


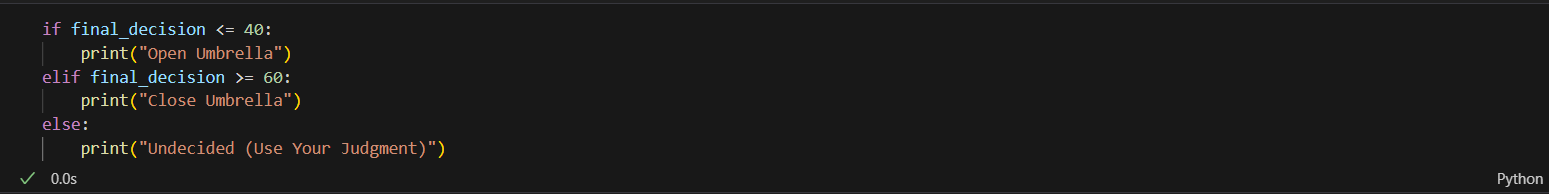


**Results:**









**CONCLUSION: -** In this experiment we have successfully implemented a Fuzzy Logic Controllers