Lab Assessment

**Experiment -1**

**Aim :** To Implement and demonstrate the FIND-S algorithm for finding the most specific hypothesis

based on a given set of training data samples. Read the training data from a .CSV file.

**Algorithm :**

The find-S algorithm is a basic concept learning algorithm in machine learning. The find-S algorithm finds the most specific hypothesis that fits all the positive examples. We have to note here that the algorithm considers only those positive training example. The find-S algorithm starts with the most specific hypothesis and generalizes this hypothesis each time it fails to classify an observed positive training data. Hence, the Find-S algorithm moves from the most specific hypothesis to the most general hypothesis.

1. Start with the most specific hypothesis.   
   **h = {ϕ, ϕ, ϕ, ϕ, ϕ, ϕ}**
2. Take the next example and if it is negative, then no changes occur to the hypothesis.
3. If the example is positive and we find that our initial hypothesis is too specific then we update our current hypothesis to a general condition.
4. Keep repeating the above steps till all the training examples are complete.
5. After we have completed all the training examples we will have the final hypothesis when can use to classify the new examples.

**Code :**

**#importing**

**import** csv

hypo **=** ['%','%','%','%','%','%'];

**with** open('trainingdata.csv') **as** csv\_file:

readcsv **=** csv**.**reader(csv\_file, delimiter**=**',')

print(readcsv)

data **=** []

print("\nThe given training examples are:")

**for** row **in** readcsv:

print(row)

**if** row[len(row)**-**1]**.**upper() **==** "YES":

data**.**append(row)

**#printting**

print("\nThe positive examples are:");

**for** x **in** data:

print(x);

print("\n");

**#finding Hypothesis**

TotalExamples **=** len(data);

i**=**0;

j**=**0;

k**=**0;

print("The steps of the Find-s algorithm are :\n",hypo);

list **=** [];

p**=**0;

d**=**len(data[p])**-**1;

**for** j **in** range(d):

list**.**append(data[i][j]);

hypo**=**list;

i**=**1;

**for** i **in** range(TotalExamples):

**for** k **in** range(d):

**if** hypo[k]**!=**data[i][k]:

hypo[k]**=**'?';

k**=**k**+**1;

**else**:

hypo[k];

print(hypo);

i**=**i**+**1;

#printing the final Hyposthesis

print("\nThe maximally specific Find-s hypothesis for the given training examples is :");

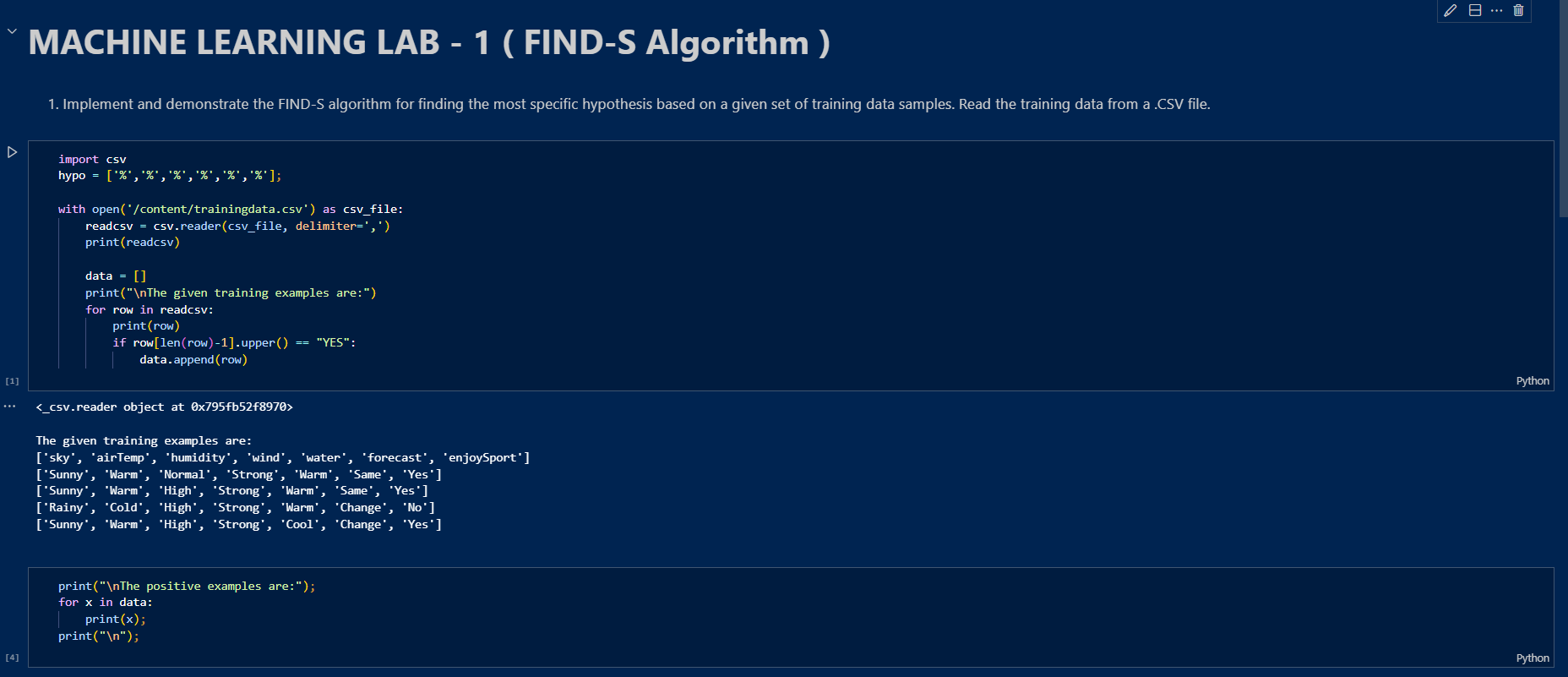
list**=**[];

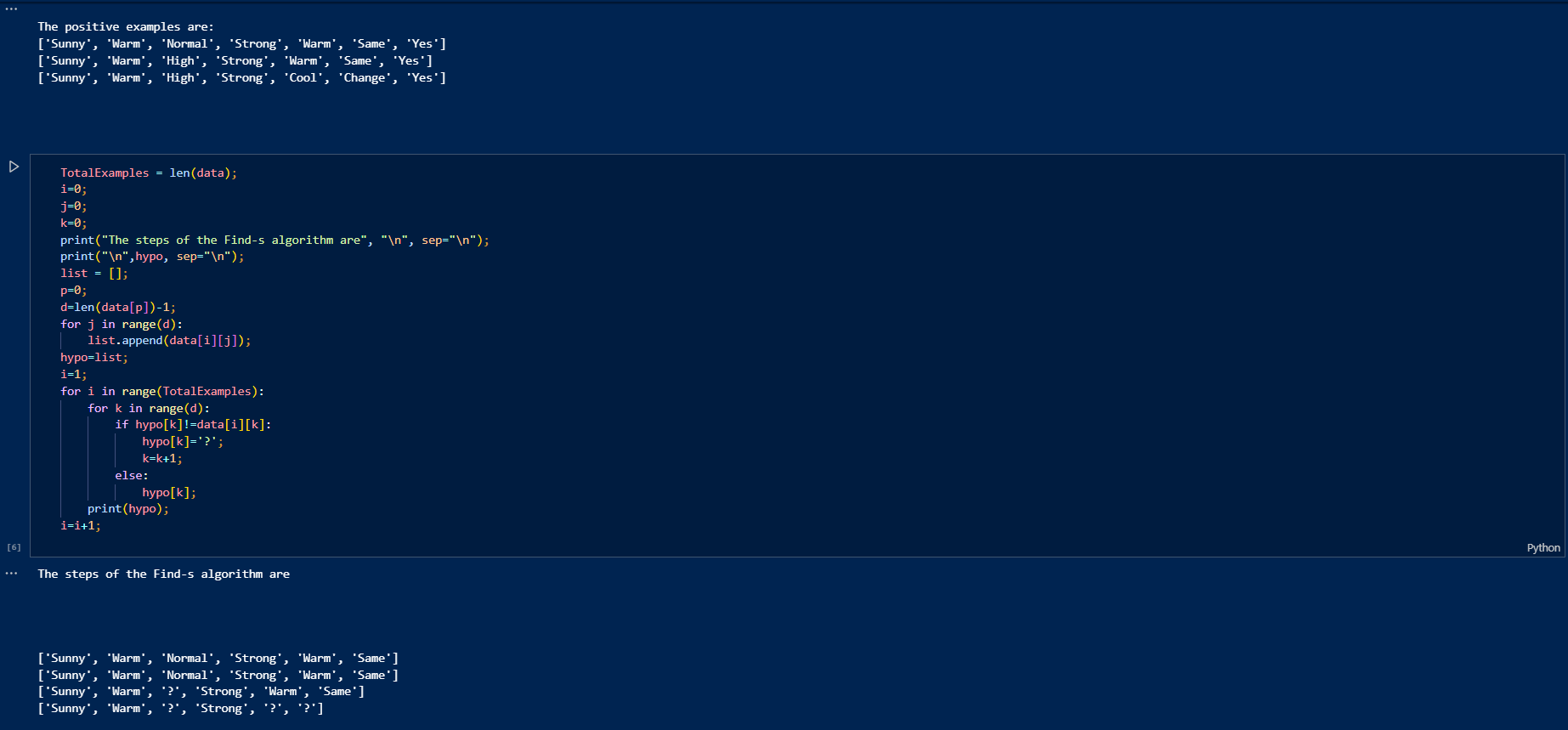
**for** i **in** range(d):

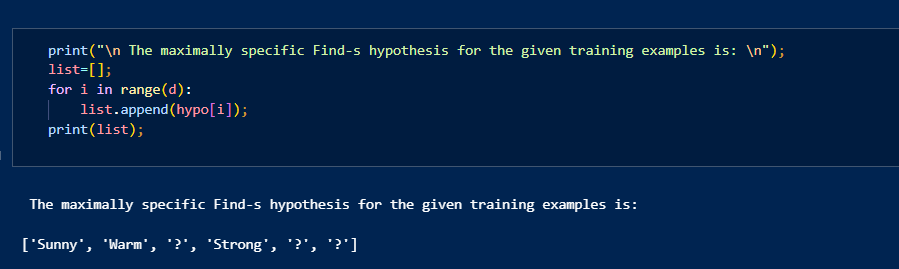
list**.**append(hypo[i]);

print(list);

**Output :**







**Conclusion :**

**In this experiment, we successfully implemented the Find-S algorithm to identify the most specific hypothesis that fits all positive training examples. By iteratively updating the hypothesis to generalize only when necessary, the algorithm efficiently converges on a specific set of conditions that describe the positive examples. This process highlights the foundational principles of machine learning, particularly in concept learning and hypothesis formation. The final hypothesis obtained can be used to classify new instances, demonstrating the practical application of the Find-S algorithm in understanding and categorizing data.**

**Experiment -2**

**Aim :** To implement For a given set of training data examples stored in a .CSV file, implement and demonstrate the Candidate Elimination algorithm to output a description of the set of all hypotheses consistent with the training examples.

**Algorithm :**

The candidate elimination algorithm incrementally builds the version space given a hypothesis space H and a set E of examples. The examples are added one by one; each example possibly shrinks the version space by removing the hypotheses that are inconsistent with the example. The candidate elimination algorithm does this by updating the general and specific boundary for each new example.

* You can consider this as an extended form of the Find-S algorithm.
* Consider both positive and negative examples.
* Actually, positive examples are used here as the Find-S algorithm (Basically they are generalizing from the specification).
* While the negative example is specified in the generalizing form.

**Step1:** Load Data set

**Step2:** Initialize General Hypothesis and Specific Hypothesis.

**Step3:** For each training example

**Step4:** If example is positive example

if attribute\_value == hypothesis\_value:

Do nothing

else:

replace attribute value with '?' (Basically generalizing it)

**Step5:** If example is Negative example

Make generalize hypothesis more specific.

**Code :**

**#importing libraries**

**import numpy as np**

**import pandas as pd**

**#loading dataset**

**data = pd.DataFrame(data=pd.read\_csv('/content/trainingdata.csv'))**

**print(data)**

**#seperating concept feature from target**

**concepts = np.array(data.iloc[:,0:-1])**

**print(concepts)**

**#** *Isolating target into a separate DataFrame*

*# copying last column to target array*

**target = np.array(data.iloc[:,-1])**

**print(target)**

**#initialising instance for concept**

**def learn(concepts, target):**

**specific\_h = concepts[0].copy()**

**print("initialization of specific\_h and general\_h")**

**print(specific\_h)**

**general\_h = [["?" for i in range(len(specific\_h))] for i in range(len(specific\_h))]**

**print(general\_h)**

**for i, h in enumerate(concepts):**

**if target[i] == "yes":**

**for x in range(len(specific\_h)):**

**if h[x] != specific\_h[x]:**

**specific\_h[x] = '?'**

**general\_h[x][x] = '?'**

**if target[i] == "no":**

**for x in range(len(specific\_h)):**

**if h[x] != specific\_h[x]:**

**general\_h[x][x] = specific\_h[x]**

**else:**

**general\_h[x][x] = '?'**

**print("steps of candidate elimination algorithm",i+1)**

**print(specific\_h)**

**print(general\_h)**

**indices = [i for i, val in enumerate(general\_h) if val == ['?', '?', '?', '?', '?', '?']]**

**for i in indices:**

**general\_h.remove(['?', '?', '?', '?', '?', '?'])**

**return specific\_h, general\_h**

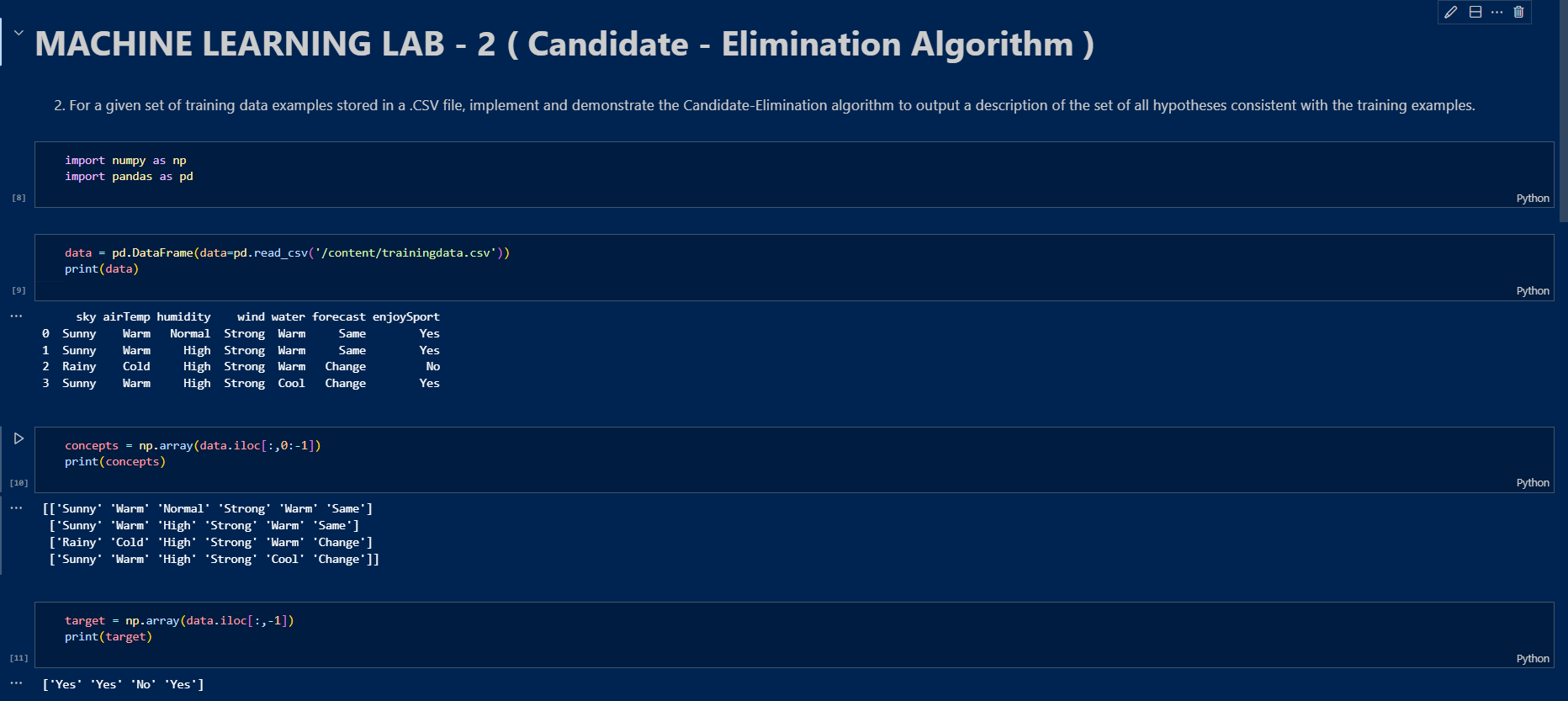
**#printing the final outcome**

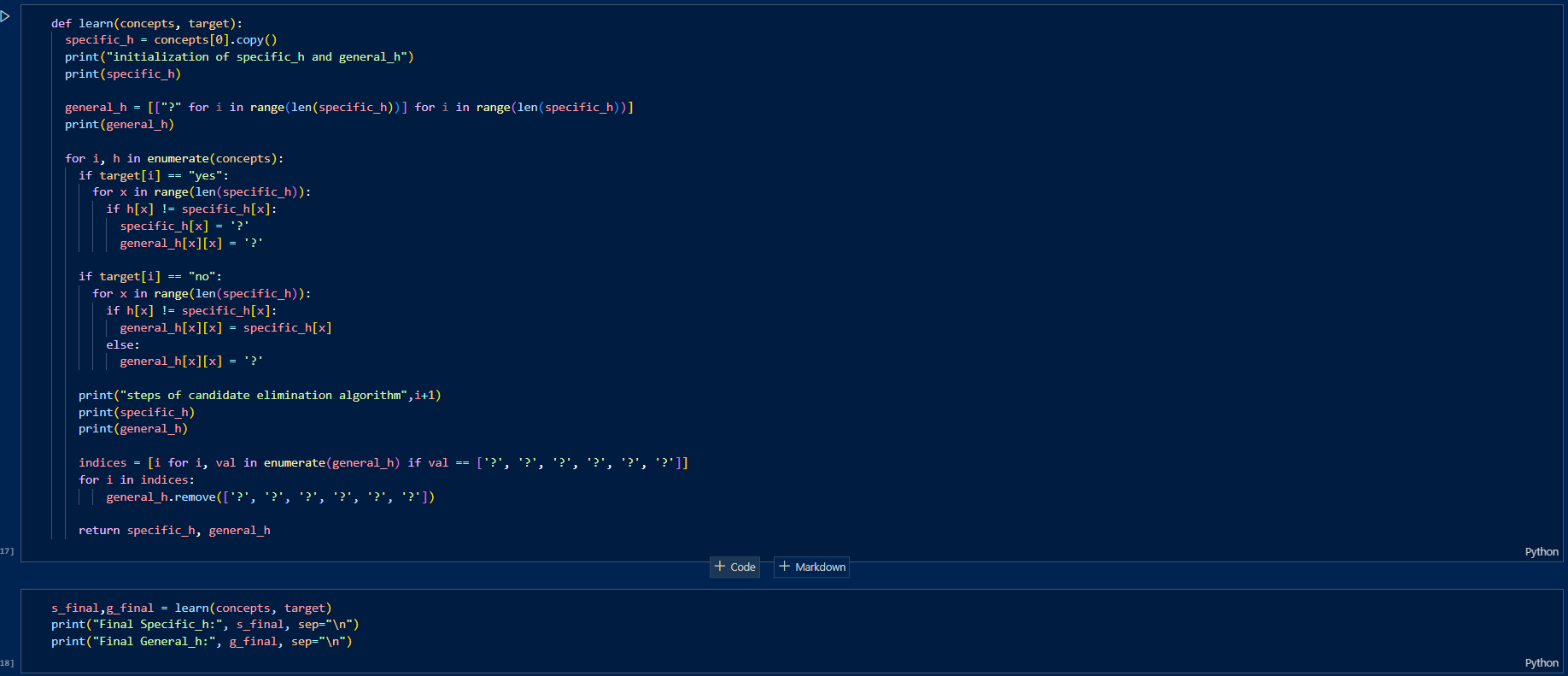
s\_final,g\_final = learn(concepts, target)

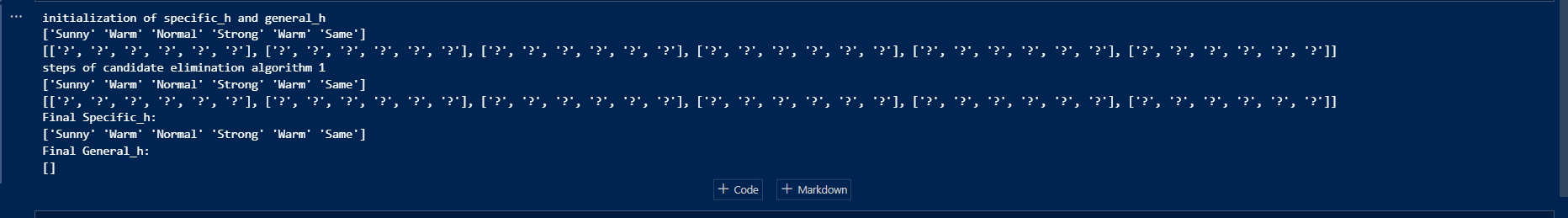
print("Final Specific\_h:", s\_final, sep="\n")

print("Final General\_h:", g\_final, sep="\n")

**Output :**

****

****

****

**Conclusion :**

**In this experiment, we successfully implemented the Candidate Elimination algorithm, which provides a systematic approach to identifying the set of all hypotheses consistent with a given set of training data. Unlike the Find-S algorithm, Candidate Elimination considers both positive and negative examples to refine the hypothesis space. The algorithm effectively narrows down the version space by updating the general and specific boundaries based on the examples encountered. This ensures that the final set of hypotheses is both as specific as necessary and as general as possible, thus encapsulating all consistent hypotheses. This experiment highlights the robustness of the Candidate Elimination algorithm in learning from data, even in the presence of conflicting examples.**

**Experiment - 3**

**Aim :** To implement a program to demonstrate the working of the decision tree based ID3 algorithm. Use an appropriate data set for building the decision tree and apply this knowledge to classify a new sample.

**Algorithm :**

The ID3 algorithm is a popular decision tree algorithm used in machine learning. It aims to build a decision tree by iteratively selecting the best attribute to split the data based on information gain. Each node represents a test on an attribute, and each branch represents a possible outcome of the test. The leaf nodes of the tree represent the final classifications

The ID3 algorithm works by building a decision tree, which is a hierarchical structure that classifies data points into different categories and splits the dataset into smaller subsets based on the values of the features in the dataset. The ID3 algorithm then selects the feature that provides the most information about the target variable. The decision tree is built top-down, starting with the root node, which represents the entire dataset. At each node, the ID3 algorithm selects the attribute that provides the most information gain about the target variable. The attribute with the highest information gain is the one that best separates the data points into different categories.

*H*(*S*)=Σ−(*Pi*​∗*log*2​(*Pi*​))

* where, 𝑃𝑖*Pi*​ represents the fraction of the sample within a particular node.
* S – The current dataset.
* i – Set of classes in S

Steps fo ID3 :

1. **Determine entropy** for the overall the dataset using class distribution.
2. For each feature.
   * Calculate **Entropy for Categorical Values**.
   * Assess **information gain** for each unique categorical value of the feature.
3. Choose the feature that generates **highest information gain**.
4. Iteratively apply all above steps to build the decision tree structure.

**Code:**

**#importing Libraries**

import numpy as np

import pandas as pd

from sklearn import tree

from sklearn.preprocessing import LabelEncoder

from sklearn.tree import DecisionTreeClassifier

**#reading the dataset**

data = pd.read\_csv('/content/tennisdata.csv')

print(data.head())

**#splitting of X and Y**

x = data.iloc[:,:-1]

print(x.head())

y = data.iloc[:,-1]

print(y.head())

**#label encoding for converting them to numerical values**

le\_outlook = LabelEncoder()

x.Outlook = le\_outlook.fit\_transform(x.Outlook)

le\_temperature = LabelEncoder()

x.Temperature = le\_temperature.fit\_transform(x.Temperature)

le\_humidity = LabelEncoder()

x.Humidity = le\_humidity.fit\_transform(x.Humidity)

le\_windy = LabelEncoder()

x.Windy = le\_windy.fit\_transform(x.Windy)

print(x.head())

le\_PlayTennis = LabelEncoder()

y = le\_PlayTennis.fit\_transform(y)

print(y)

**#training the Decsision tree classifier model with x and y and predicting the values**

from sklearn.preprocessing import LabelEncoder

from sklearn.tree import DecisionTreeClassifier

# Assuming the label encoders and classifier have already been defined and trained

classifier = DecisionTreeClassifier()

classifier.fit(x, y)

# Function to encode input

def labelEncoderForInput(list1):

    list1[0] = le\_outlook.transform([list1[0]])[0]

    list1[1] = le\_temperature.transform([list1[1]])[0]

    list1[2] = le\_humidity.transform([list1[2]])[0]

    list1[3] = le\_windy.transform([list1[3]])[0]

    return [list1]

# Predict for an input

inp = ["Rainy", "Mild", "High", "False"]

inp1 = ["Rainy", "Cool", "High", "False"]

pred1 = labelEncoderForInput(inp1)

y\_pred = classifier.predict(pred1)

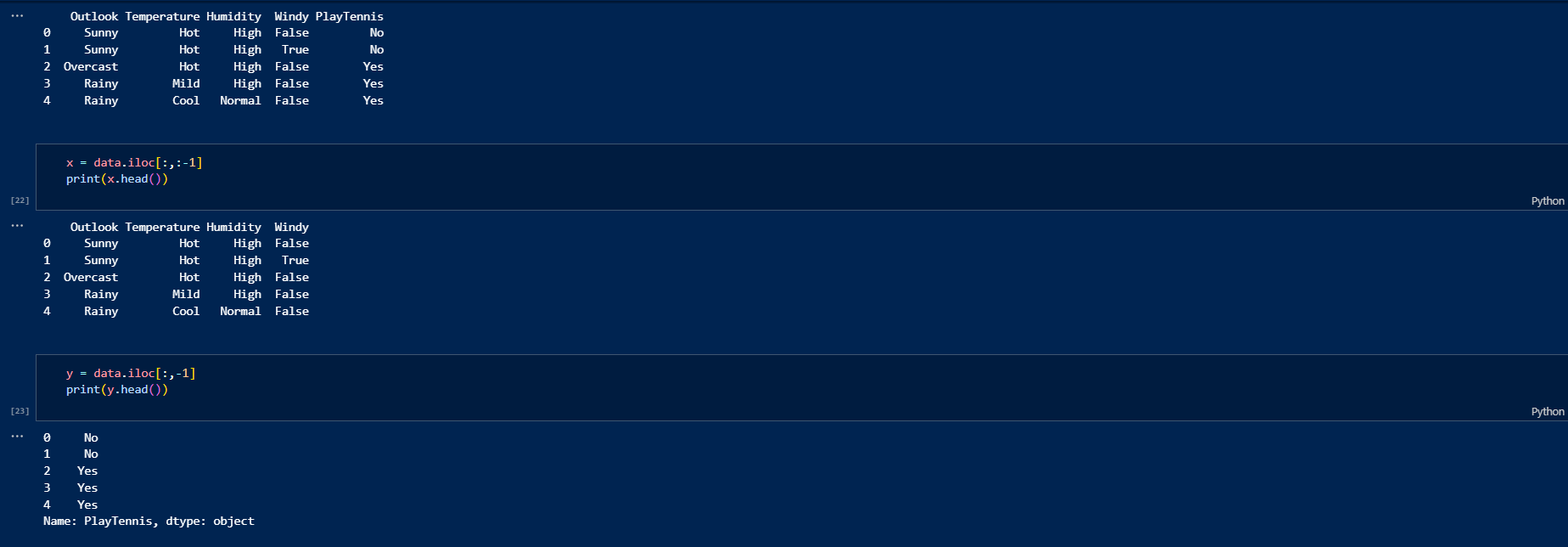
y\_pred

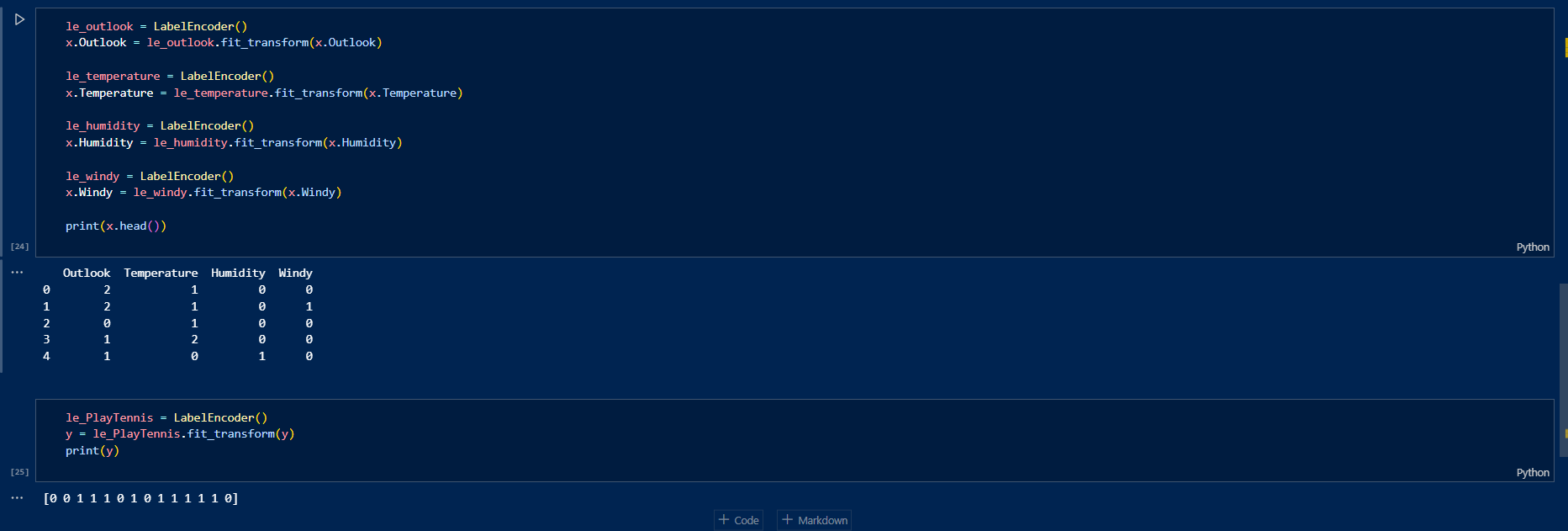
# Print the result

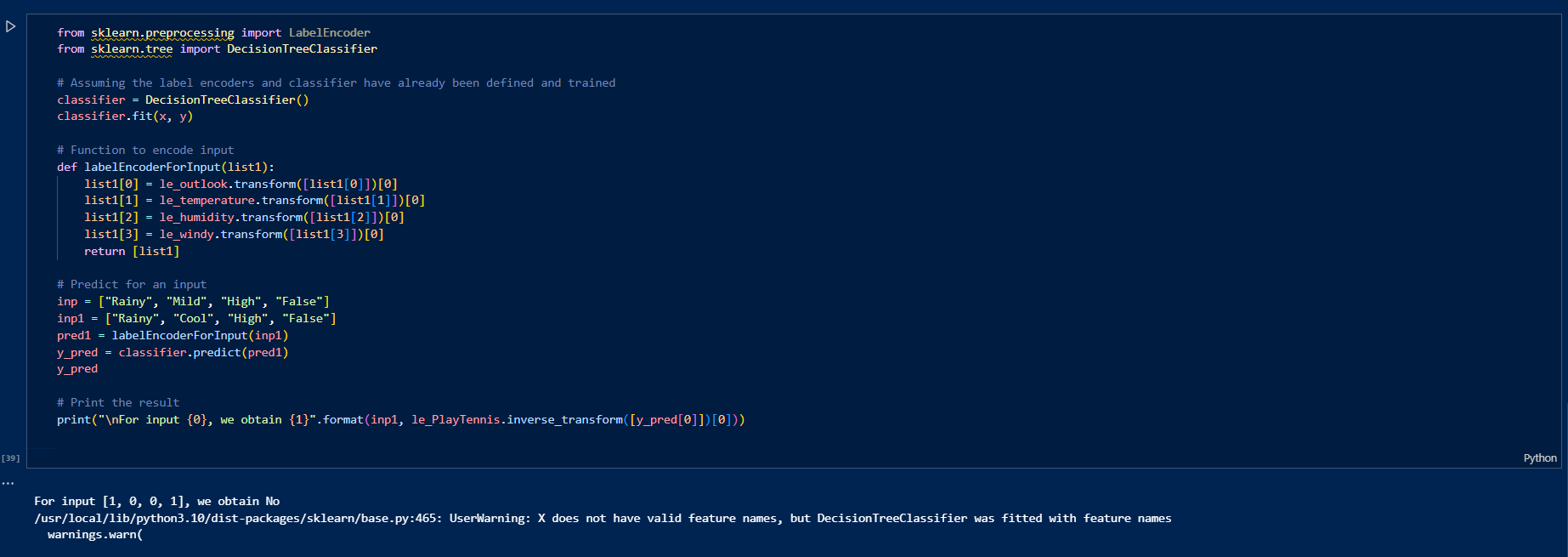
print("\nFor input {0}, we obtain {1}".format(inp1, le\_PlayTennis.inverse\_transform([y\_pred[0]])[0]))

**Output:**

****

****

****

****

**Conclusion:**

**In this experiment, we successfully implemented the ID3 algorithm to build a decision tree for classifying new samples based on the given dataset. By using the concept of information gain, we were able to iteratively select the best attributes to split the data and construct a decision tree. The decision tree classifier effectively categorized new samples by testing attributes and making decisions based on the learned structure. This demonstrates the power and efficiency of decision trees in handling categorical data and making accurate predictions. The experiment highlighted the practical application of the ID3 algorithm in machine learning and its ability to provide clear, interpretable models for decision-making.**

**Experiment - 4**

**Aim :** To implement and Build an Artificial Neural Network by implementing the Back propagation algorithm and test the same using appropriate data sets.

**Algorithm :**

Backpropagation is an iterative algorithm, that helps to minimize the cost function by determining which weights and biases should be adjusted. During every epoch, the model learns by adapting the weights and biases to minimize the loss by moving down toward the gradient of the error. Thus, it involves the two most popular optimization algorithms, such as [gradient descent](https://www.geeksforgeeks.org/gradient-descent-algorithm-and-its-variants/) or [stochastic gradient descent](https://www.geeksforgeeks.org/ml-stochastic-gradient-descent-sgd/).

**Implementation of Back Propagation algorithm in Python :**

1. **Neural Network Initialization**: The NeuralNetwork class is initialized with parameters for the input size, hidden layer size, and output size. It also initializes the weights and biases with random values.
2. **Sigmoid Activation Function**: The sigmoid method implements the sigmoid activation function, which squashes the input to a value between 0 and 1.
3. **Sigmoid Derivative**: The sigmoid\_derivative method calculates the derivative of the sigmoid function. It computes the gradients of the loss function with respect to weights.
4. **Feedforward Pass**: The feedforward method calculates the activations of the hidden and output layers based on the input data and current weights and biases. It uses matrix multiplication to propagate the inputs through the network.
5. **Backpropagation**: The backward method performs the backpropagation algorithm. It calculates the error at the output layer and propagates it back through the network to update the weights and biases using gradient descent.
6. **Training the Neural Network**: The train method trains the neural network using the specified number of epochs and learning rate. It iterates through the training data, performs the feedforward and backward passes, and updates the weights and biases accordingly.
7. **XOR Dataset**: The XOR dataset (X) is defined, which contains input pairs that represent the XOR operation, where the output is 1 if exactly one of the inputs is 1, and 0 otherwise.
8. **Testing the Trained Model**: After training, the neural network is tested on the XOR dataset (X) to see how well it has learned the XOR function. The predicted outputs are printed to the console, showing the neural network’s predictions for each input pair.

**Code:**

**#importing libraries**

**Import numpy as np**

x = np.array([[0.66666667, 1.],

              [0.33333333, 0.55555556],

              [1., 0.66666667]])

y = np.array([[0.92],

              [0.86],

              [0.89]])

class Neural\_Network:

    def \_\_init\_\_(self):

        self.inputSize = 2

        self.outputSize = 1

        self.hiddenSize = 3

        self.W1 = np.random.randn(self.inputSize, self.hiddenSize)

        self.W2 = np.random.randn(self.hiddenSize, self.outputSize)

    def forward(self, X):

        self.z = np.dot(X, self.W1)

        self.z2 = self.sigmoid(self.z)

        self.z3 = np.dot(self.z2, self.W2)

        o = self.sigmoid(self.z3)

        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

        self.o\_delta = self.o\_error \* self.sigmoidPrime(o)

        self.z2\_error = self.o\_delta.dot(self.W2.T)

        self.z2\_delta = self.z2\_error \* self.sigmoidPrime(self.z2)

        self.W1 += X.T.dot(self.z2\_delta)

        self.W2 += self.z2.T.dot(self.o\_delta)

    def train(self, X, y):

        o = self.forward(X)

        self.backward(X, y, o)

NN = Neural\_Network()

for i in range(1000):

    print("\nInput: \n" + str(x))

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

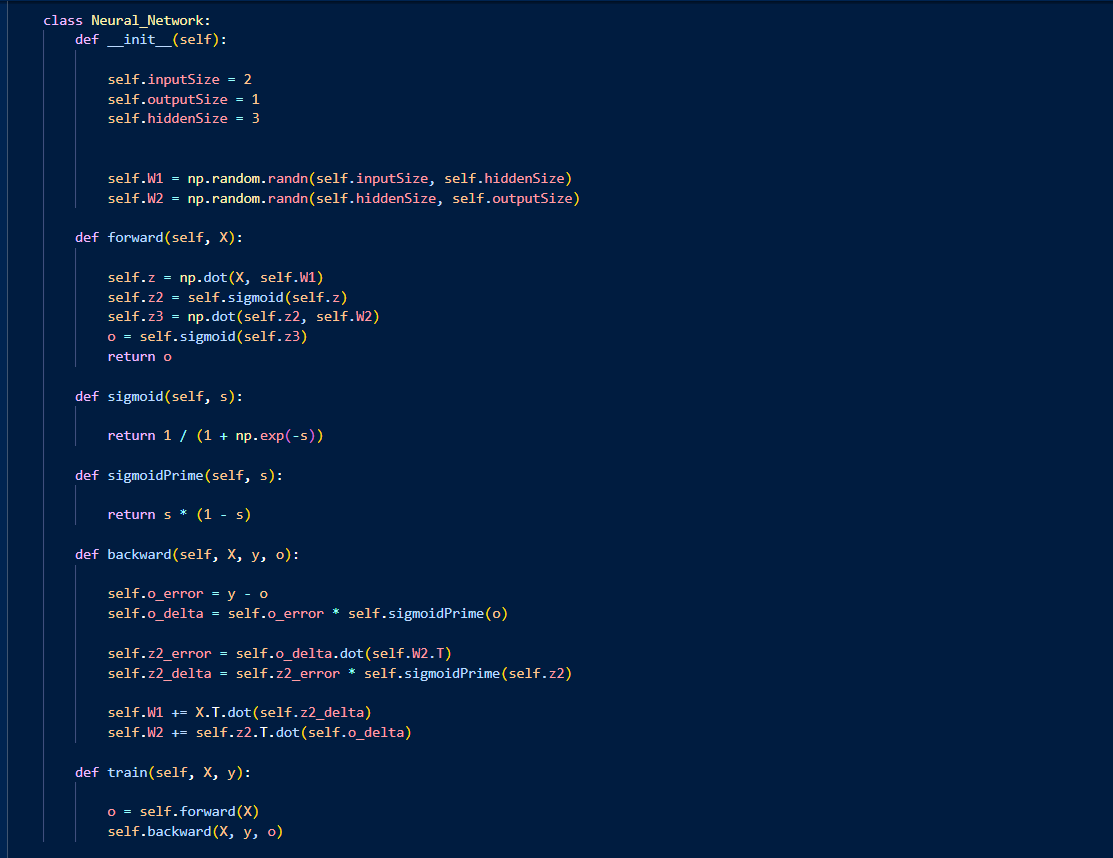
    print("\nPredicted Output: \n" + str(NN.forward(x)))

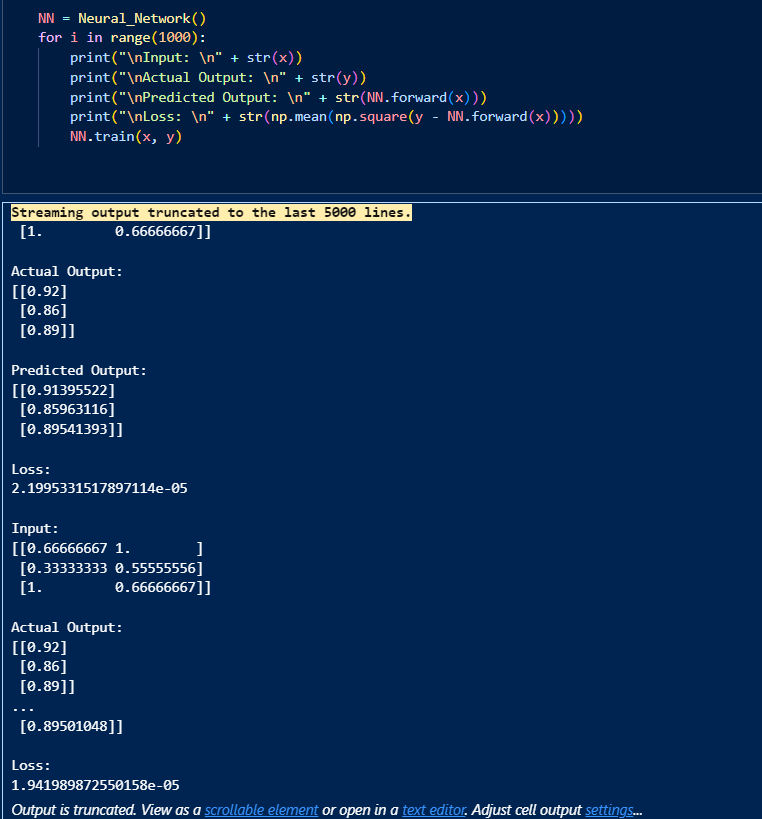
    print("\nLoss: \n" + str(np.mean(np.square(y - NN.forward(x)))))

    NN.train(x, y)

**Output:**

****

****

****

**Conclusion:**

**In this experiment, we successfully implemented and tested an Artificial Neural Network (ANN) using the Backpropagation algorithm. The network was trained on a simple dataset to learn the XOR function. Through iterative training, the weights and biases of the network were adjusted to minimize the error between the predicted outputs and the actual outputs. By the end of the training process, the network demonstrated its ability to accurately predict the XOR outputs, showcasing the effectiveness of the Backpropagation algorithm in training neural networks. This experiment highlights the fundamental concepts of neural network training and the importance of backpropagation in optimizing model performance.**

**Experiment - 5**

**Aim :** To implement and Apply EM algorithm to cluster a set of data stored in a .CSV file. Use the same data set for clustering using k-Means algorithm. Compare the results of these two algorithms and comment on the quality of clustering. You can add Java/Python ML library classes/API in the program.

**Algorithm :**

[K-Means Clustering](https://www.geeksforgeeks.org/k-means-clustering-introduction/) is an[Unsupervised Machine Learning](https://www.geeksforgeeks.org/ml-types-learning-part-2/) algorithm, which groups the unlabeled dataset into different clusters.

The algorithm works as follows:

1. First, we randomly initialize k points, called means or cluster centroids.
2. We categorize each item to its closest mean, and we update the mean’s coordinates, which are the averages of the items categorized in that cluster so far.
3. We repeat the process for a given number of iterations and at the end, we have our clusters.

The Pseudocode code of the above algorithm is :

Initialize k means with random values  
--> For a given number of iterations:  
   
 --> Iterate through items:  
   
 --> Find the mean closest to the item by calculating   
 the euclidean distance of the item with each of the means  
   
 --> Assign item to mean  
   
 --> Update mean by shifting it to the average of the items in that cluster

**Code:**

**#importing libraries**

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

from sklearn.cluster import KMeans

from sklearn import preprocessing

from sklearn.mixture import GaussianMixture

from sklearn.datasets import load\_iris

import sklearn.metrics as sm

**#reading the dataset**

dataset = load\_iris()

print(dataset)

x = pd.DataFrame(dataset.data)

x.columns = ['Sepal\_Length','Sepal\_Width','Petal\_Length','Petal\_Width']

y = pd.DataFrame(dataset.target)

y.columns = ['Targets']

print(x)

print(y)

**#Real Plot**

plt**.**figure(figsize**=**(14,7))

colormap**=**np**.**array(['red','lime','black'])

plt**.**subplot(1,3,1)

plt**.**scatter(X**.**Petal\_Length,X**.**Petal\_Width,c**=**colormap[y**.**Targets],s**=**40)

plt**.**title('Real')

#K-Means Plot

plt.subplot(1,3,2)

model = KMeans(n\_clusters=3)

model.fit(x)

predY = np.choose(model.labels\_,[0,1,2]).astype(np.int64)

plt.scatter(x.Petal\_Length,x.Petal\_Width,c=colormap[predY],s=40)

plt.title('K-Means')

#GMM plot

scaler = preprocessing.StandardScaler()

scaler.fit(x)

xsa = scaler.transform(x)

xs = pd.DataFrame(xsa,columns=x.columns)

gmm = GaussianMixture(n\_components = 3)

gmm.fit(xs)

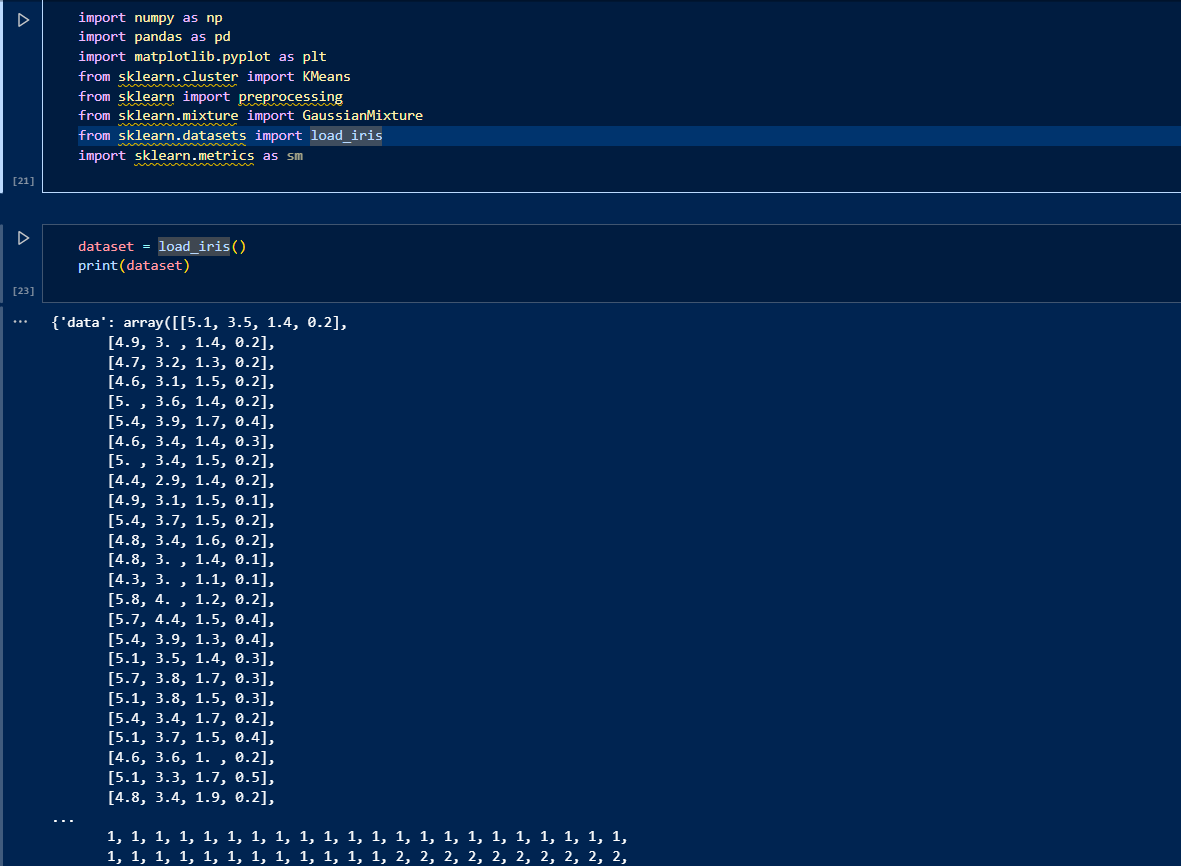
y\_cluster\_gmm = gmm.predict(xs)

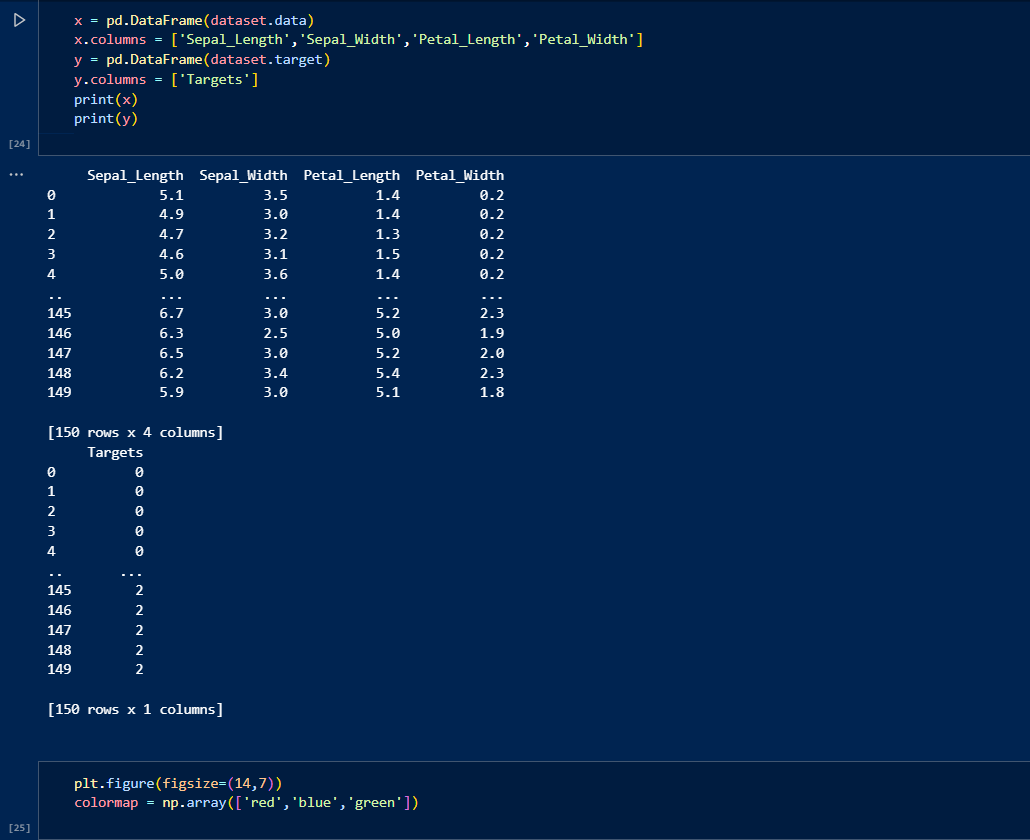
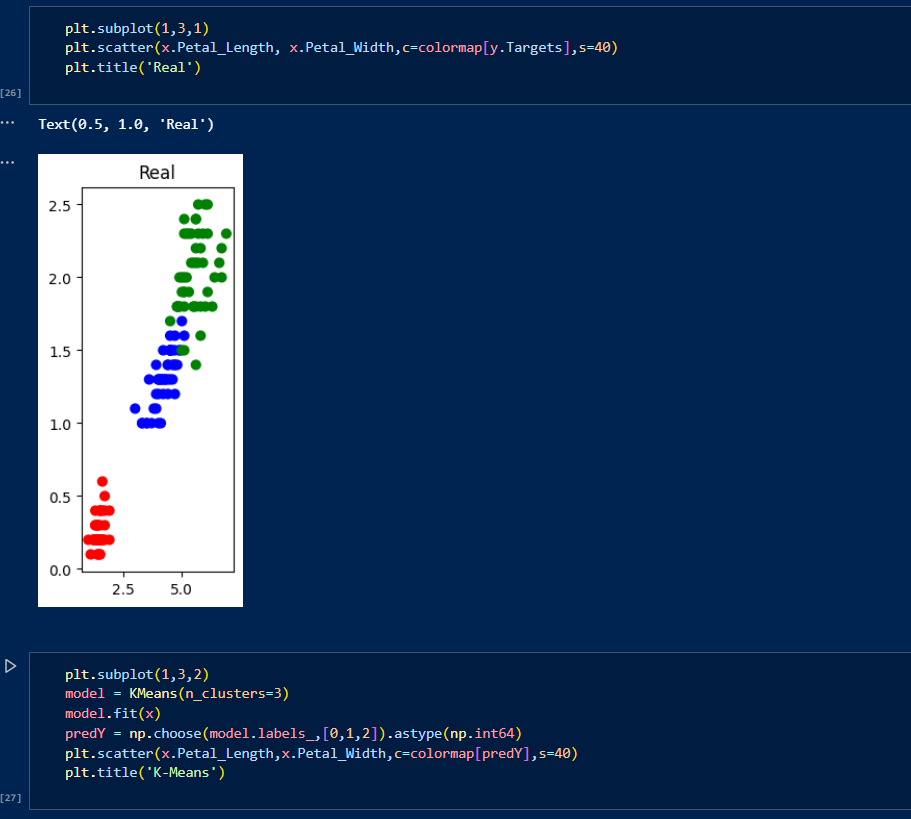
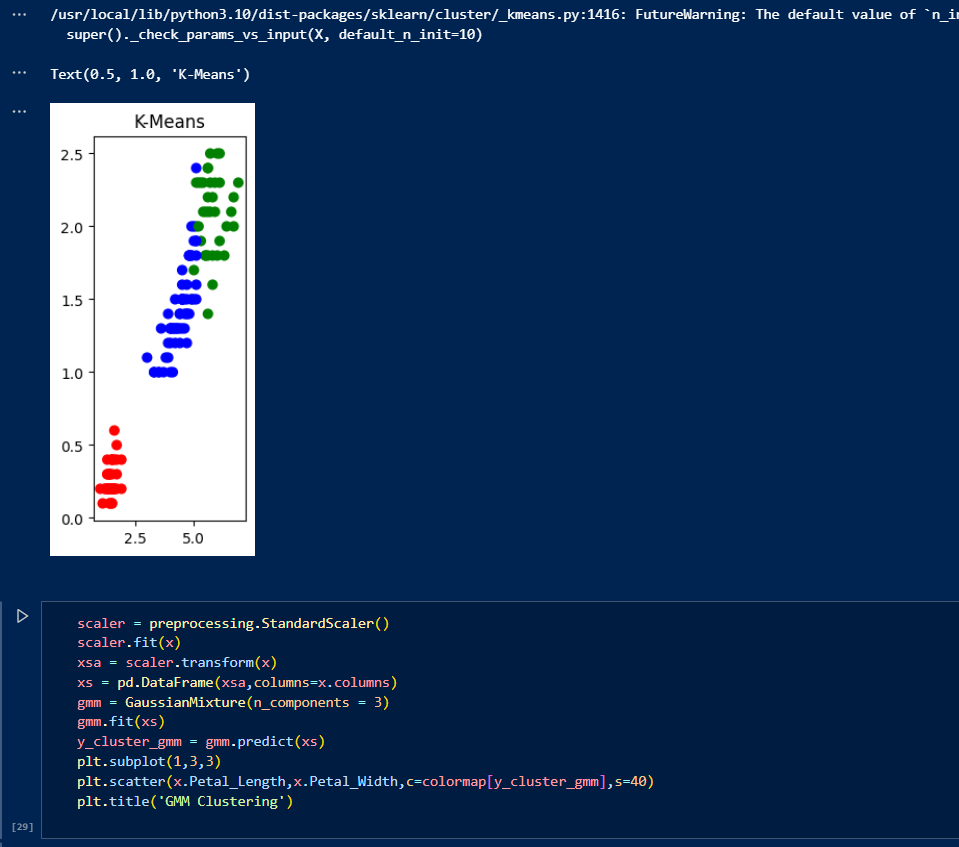
plt.subplot(1,3,3)

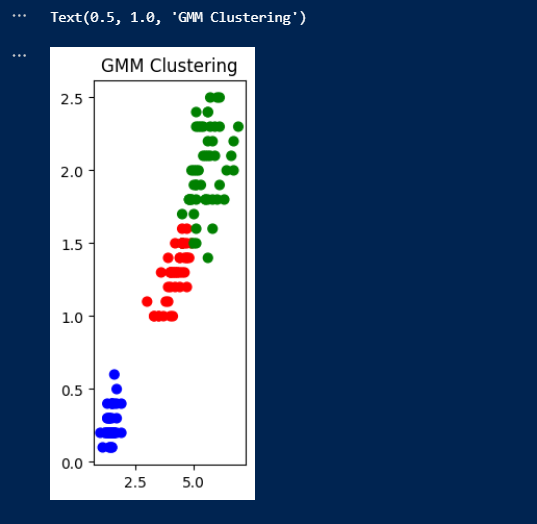
plt.scatter(x.Petal\_Length,x.Petal\_Width,c=colormap[y\_cluster\_gmm],s=40)

plt.title('GMM Clustering')

**Output:**

****

**** **** 

****

**Conclusion:**

**In this algorithm we applied both the algorithms K-means and Expected Maximization plotting K-mean and GMM in which we say that t**he K-Means clustering resulted in distinct clusters but may have difficulty capturing the true underlying distribution of the data due to its reliance on spherical clusters.GMM provided a more flexible clustering, capturing the variances and covariances of the data more accurately, resulting in clusters that better represent the true distribution of the Iris dataset.

Overall, GMM clustering tends to perform better in scenarios where the data has varying cluster shapes and densities, while K-Means is faster and simpler to implement for spherical clusters. The choice between the two algorithms depends on the specific characteristics of the dataset and the desired clustering outcome.

**Experiment - 6**

**Aim :** Write a program to implement k-Nearest Neighbour algorithm to classify the iris data set. Print both correct and wrong predictions. Java/Python ML library classes can be used for this problem.

**Algorithm :**

KNN is one of the most basic yet essential classification algorithms in machine learning. It belongs to the [supervised learning](https://www.geeksforgeeks.org/supervised-unsupervised-learning) domain and finds intense application in pattern recognition, [data mining](https://www.geeksforgeeks.org/data-mining), and intrusion detection.

It is widely disposable in real-life scenarios since it is non-parametric, meaning it does not make any underlying assumptions about the distribution of data (as opposed to other algorithms such as GMM, which assume a [Gaussian distribution](https://www.geeksforgeeks.org/mathematics-probability-distributions-set-3-normal-distribution) of the given data). We are given some prior data (also called training data), which classifies coordinates into groups identified by an attribute.

**Algorithm for KNN:**

**Step 1: Selecting the optimal value of K**

* K represents the number of nearest neighbors that needs to be considered while making prediction.

**Step 2: Calculating distance**

* To measure the similarity between target and training data points, Euclidean distance is used. Distance is calculated between each of the data points in the dataset and target point.

**Step 3: Finding Nearest Neighbors**

* The k data points with the smallest distances to the target point are the nearest neighbors.

**Step 4: Voting for Classification or Taking Average for Regression**

* In the classification problem, the class labels of K-nearest neighbors are determined by performing majority voting. The class with the most occurrences among the neighbors becomes the predicted class for the target data point.
* In the regression problem, the class label is calculated by taking average of the target values of K nearest neighbors. The calculated average value becomes the predicted output for the target data point.

Let X be the training dataset with n data points, where each data point is represented by a d-dimensional feature vector 𝑋𝑖*Xi*​ and Y be the corresponding labels or values for each data point in X. Given a new data point x, the algorithm calculates the distance between x and each data point 𝑋𝑖*Xi*​ in X using a distance metric, such as Euclidean distance:distance(𝑥,𝑋𝑖)=∑𝑗=1𝑑(𝑥𝑗–𝑋𝑖𝑗)2]distance(*x*,*Xi*​)=∑*j*=1*d*​(*xj*​–*Xij*​​)2​]

The algorithm selects the K data points from X that have the shortest distances to x. For classification tasks, the algorithm assigns the label y that is most frequent among the K nearest neighbors to x. For regression tasks, the algorithm calculates the average or weighted average of the values y of the K nearest neighbors and assigns it as the predicted value for x.

**Code:**

**#importing libraries**

import numpy as np

from sklearn.datasets import load\_iris

from sklearn.neighbors import KNeighborsClassifier

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

**#reading the dataset**

dataset = load\_iris()

x\_train,x\_test ,y\_train,y\_test = train\_test\_split(dataset["data"],dataset["target"],random\_state=0)

**#implementing K Nearest Neighbors**

kn = KNeighborsClassifier(n\_neighbors=1)

kn.fit(x\_train,y\_train)

**#predicting the output**

for i in range(len(x\_test)):

  x=x\_test[i]

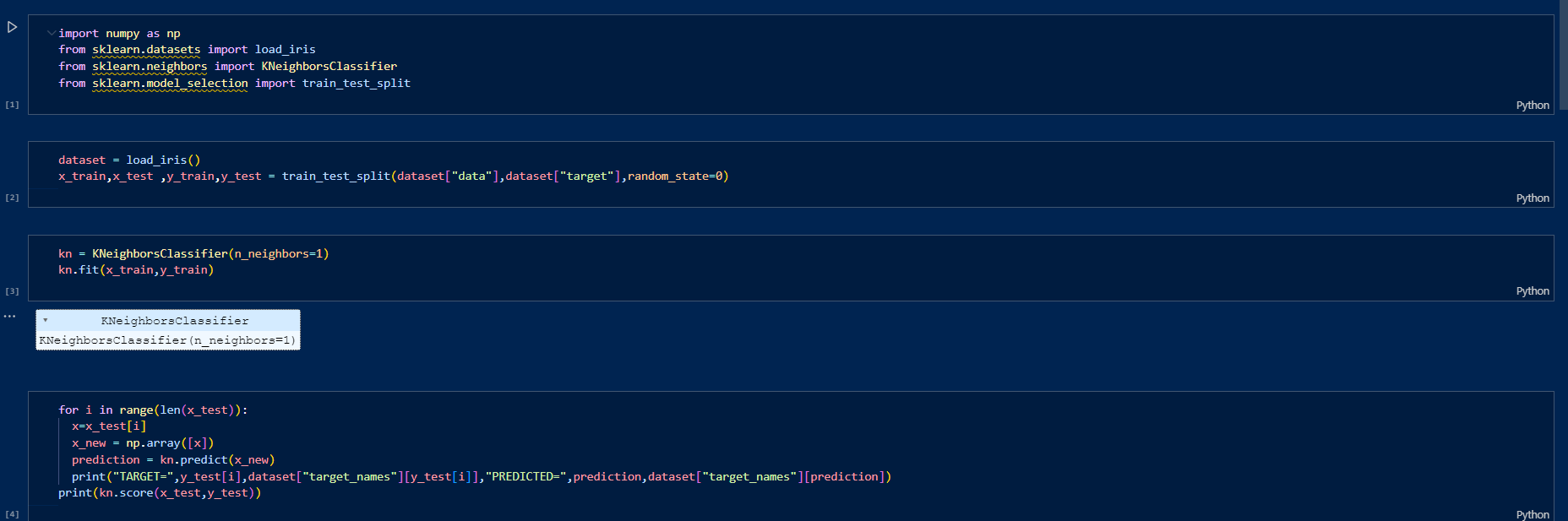
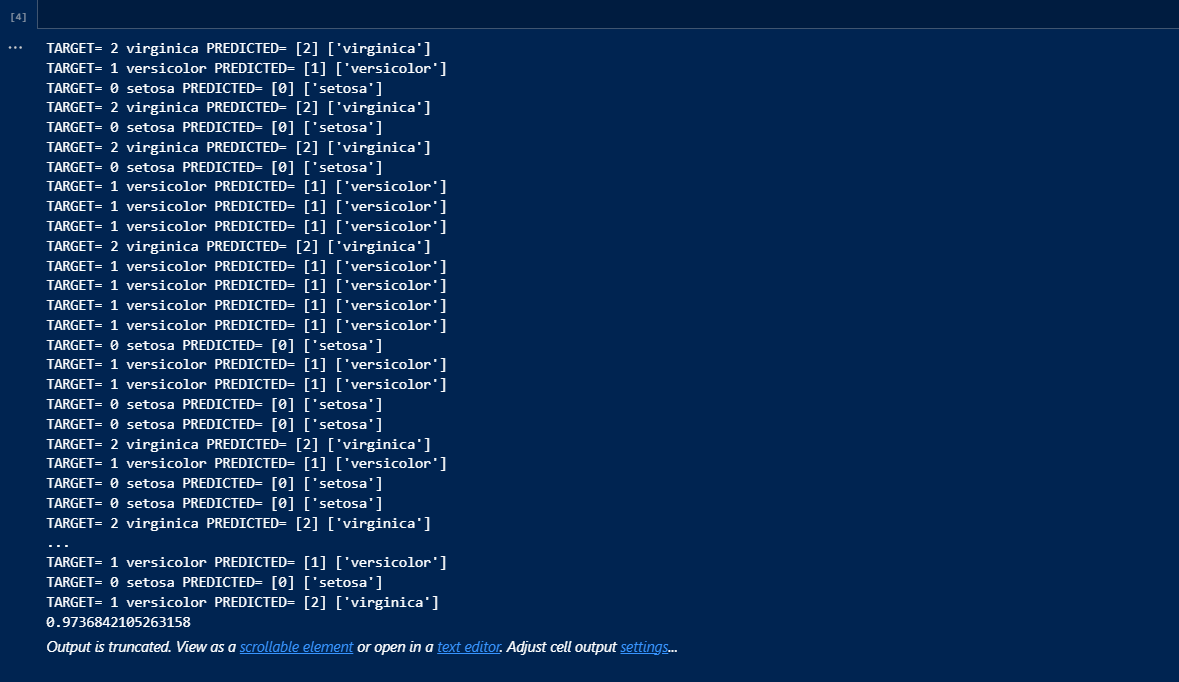
  x\_new = np.array([x])

  prediction = kn.predict(x\_new)

  print("TARGET=",y\_test[i],dataset["target\_names"][y\_test[i]],"PREDICTED=",prediction,dataset["target\_names"][prediction])

print(kn.score(x\_test,y\_test))

**Output:**

**** ****

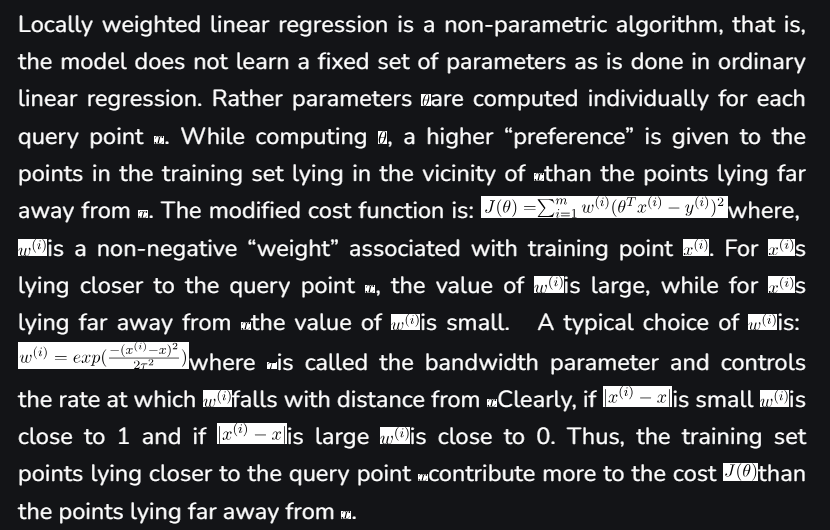
**Conclusion:**

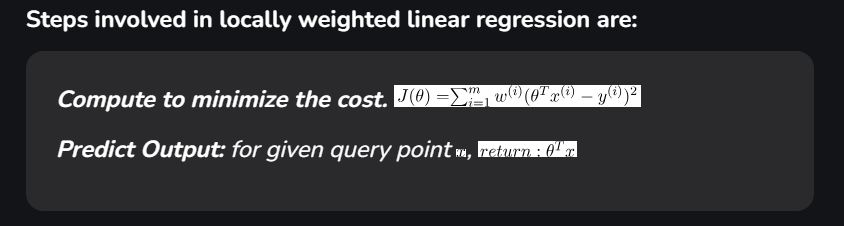
In this experiment, we implemented the k-Nearest Neighbour (k-NN) algorithm to classify the Iris dataset. The Iris dataset contains measurements of iris flowers and their corresponding species. We used the k-NN algorithm with k=1k = 1k=1 to predict the species of the flowers in the test set based on their measurements. We trained the k-NN model on the training set and used it to predict the species of the flowers in the test set. The output includes the actual species (target) and the predicted species for each test data point, allowing us to see both correct and incorrect predictions. The accuracy score of the model on the test set is printed, indicating how well the model performed.

**Experiment - 7**

**Aim :** Implement the non-parametric Locally Weighted Regression algorithm in order to fit data points. Select appropriate data set for your experiment and draw graphs.

**Algorithm :**

****

****

**Code:**

**#importing libraries**

from math import ceil

import numpy as np

from scipy import linalg

def lowess(x,y,f,iterations):

  n = len(x)

  r = int(ceil(f\*n))

  h = [np.sort(np.abs(x - x[i]))[r] for i in range(n)]

  w = np.clip(np.abs((x[:,None] - x[None,:])/h),0.0,1.0)

  w = (1 - w\*\*3)\*\*3

  yest = np.zeros(n)

  delta = np.ones(n)

  for iteration in range(iterations):

    for i in range(n):

      weights = delta \* w[:,i]

      b = np.array([np.sum(weights\*y), np.sum(weights\*y\*x)])

      A = np.array([[np.sum(weights),np.sum(weights \* x)],[np.sum(weights \* x),np.sum(weights \* x\*\*2)]])

      beta = linalg.solve(A,b)

      yest[i] = beta[0] + beta[1] \* x[i]

    residuals = y - yest

    s = np.median(np.abs(residuals))

    delta = np.clip(residuals/(6.0 \* s),-1,1)

    delta = (1-delta\*\*2)\*\*2

  return yest

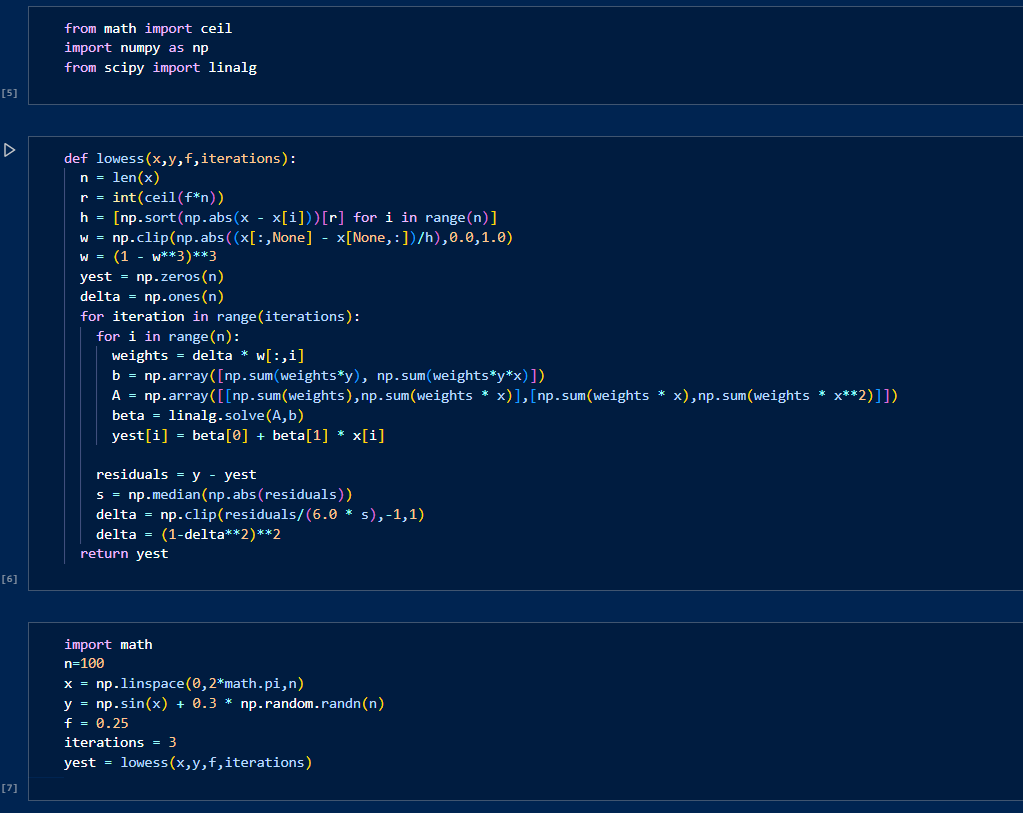
**#plotting**

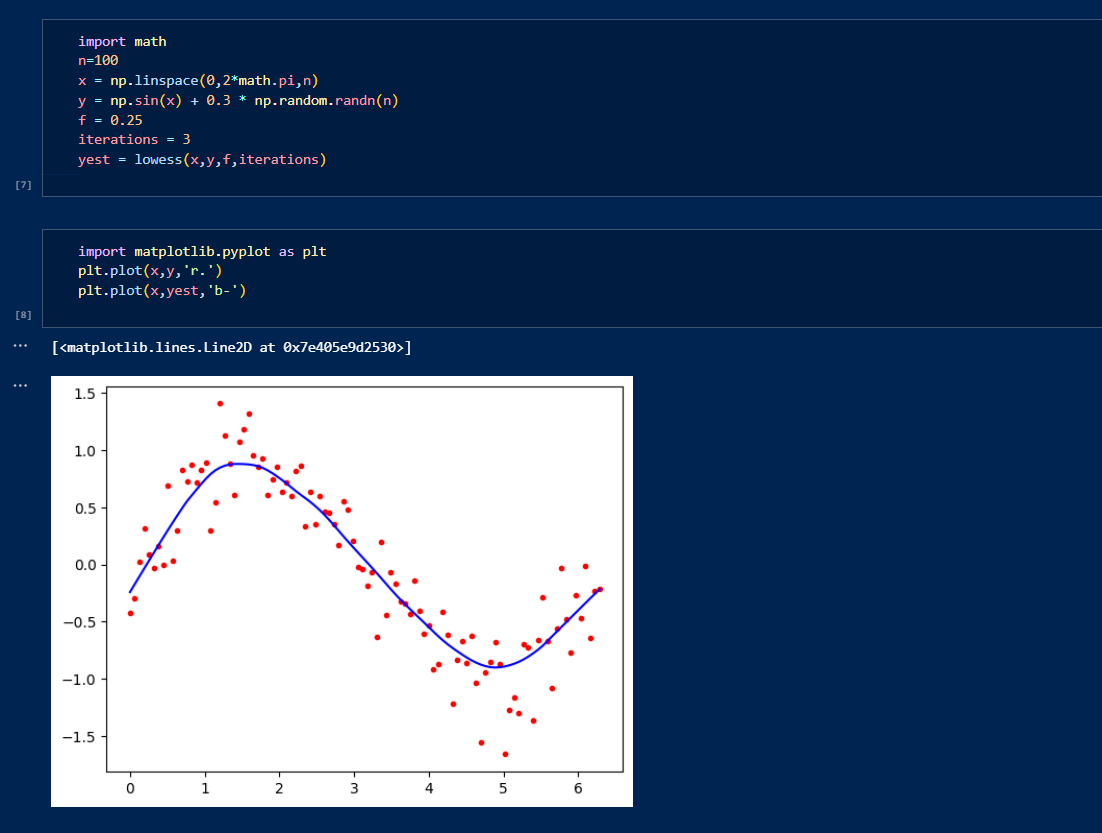
import matplotlib.pyplot as plt

plt.plot(x,y,'r.')

plt.plot(x,yest,'b-')

**Output:**

****

****

**Conclusion:**

In this experiment, we implemented the non-parametric Locally Weighted Regression (Lowess) algorithm to fit a set of data points. The Lowess algorithm is a powerful technique for smoothing scatterplots and capturing local trends in data without assuming a global functional form. The implementation fits a smooth curve to the given data points, as shown in the plot.

The red dots represent the original data points, while the blue line represents the fitted curve obtained using the Lowess algorithm.

**Quality of Fitting:**

* The Lowess algorithm effectively captures local trends in the data, providing a smooth fit that adapts to variations in the dataset.
* It is particularly useful for datasets where the relationship between variables is not well-represented by a single global model.
* The degree of smoothing can be adjusted by tuning the parameter fff, allowing for flexible modeling of different types of data.

Overall, Locally Weighted Regression is a versatile and robust method for smoothing and fitting data, making it a valuable tool for exploratory data analysis and visualizing complex relationships in datasets.