1. **Aim :** **Implementing K-Nearest Neighbors Algorithm**

**Description :**

**K-Nearest Neighbors (KNN)** is a simple, intuitive, and widely used supervised machine learning algorithm used for classification and regression tasks. It works by finding the most similar (or "nearest") data points in the feature space and making predictions based on their labels or values.

For a given test data point, find the 'k' closest points (neighbors) from the training set based on a distance metric (typically Euclidean distance).

* **Classification**: The test point is assigned the most common class label among the k neighbors.
* **Regression**: The test point’s predicted value is the average (or weighted average) of the values of its k nearest neighbors.

**Procedure :**

1. **Choose the number of neighbors (k)**: The parameter k represents the number of nearest neighbors to consider for making a prediction. Typically, k is a positive integer, and common values are 3, 5, or 7.
2. **Distance metric**: A distance metric (e.g., Euclidean distance, Manhattan distance, or Minkowski distance) is used to measure the similarity between data points. For classification problems, we typically use Euclidean distance:

Euclidean distance =

where x(i) and y(i)are the coordinates of two points in the feature space, and nnn is the number of features.

1. **Identify k nearest neighbors**: For each test instance, the algorithm computes the distance to all points in the training dataset and selects the k closest ones.
2. **Make predictions**:
   * **Classification**: Assign the most frequent class label among the k neighbors to the test point.
   * **Regression**: Calculate the average (or weighted average) of the values from the k neighbors.

Source Code :

#%%

# Importing libraries

import numpy as nm

import matplotlib.pyplot as plt

import pandas as pd

#importing datasets

data\_set= pd.read\_csv('User\_Data.csv')

#Extracting Independent and dependent Variable

x= data\_set.iloc[:, [2,3]].values

y= data\_set.iloc[:, 4].values

# Splitting the dataset into training and test set.

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

x\_train, x\_test, y\_train, y\_test= train\_test\_split(x, y, test\_size= 0.25, random\_state=0)  # Holdout cross-validation

#feature Scaling

from sklearn.preprocessing import StandardScaler

st\_x= StandardScaler()

x\_train= st\_x.fit\_transform(x\_train)

x\_test= st\_x.transform(x\_test)

# %%

# Fitting K-NN classifier to the training set

from sklearn.neighbors import KNeighborsClassifier

classifier= KNeighborsClassifier(n\_neighbors=5, metric='minkowski', p=2 )

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

#Predicting the test set result

y\_pred= classifier.predict(x\_test)

#Creating the Confusion matrix

from sklearn.metrics import confusion\_matrix

cm= confusion\_matrix(y\_test, y\_pred)

# %%

# Visulaizing the trianing set result

import matplotlib.pyplot as plt

from matplotlib.colors import ListedColormap

# import matplotlib.colors as mcolors

x\_set, y\_set = x\_train, y\_train

x1, x2 = nm.meshgrid(nm.arange(start = x\_set[:, 0].min() - 1, stop = x\_set[:, 0].max() + 1, step  =0.01),

nm.arange(start = x\_set[:, 1].min() - 1, stop = x\_set[:, 1].max() + 1, step = 0.01))

plt.contourf(x1, x2, classifier.predict(nm.array([x1.ravel(), x2.ravel()]).T).reshape(x1.shape),

alpha = 0.75, cmap = ListedColormap(['red','green' ]))

plt.xlim(x1.min(), x1.max())

plt.ylim(x2.min(), x2.max())

for i, j in enumerate(nm.unique(y\_set)):

    plt.scatter(x\_set[y\_set == j, 0], x\_set[y\_set == j, 1],

        color = ListedColormap(['red', 'green'])(i), label = j)

plt.title('K-NN Algorithm (Training set)')

plt.xlabel('Age')

plt.ylabel('Estimated Salary')

plt.legend()

plt.show()

# %%

# Visualizing the test set result

from matplotlib.colors import ListedColormap

x\_set, y\_set = x\_test, y\_test

x1, x2 = nm.meshgrid(nm.arange(start = x\_set[:, 0].min() - 1, stop = x\_set[:, 0].max() + 1, step  =0.01),

nm.arange(start = x\_set[:, 1].min() - 1, stop = x\_set[:, 1].max() + 1, step = 0.01))

plt.contourf(x1, x2, classifier.predict(nm.array([x1.ravel(), x2.ravel()]).T).reshape(x1.shape),

alpha = 0.75, cmap = ListedColormap(['red','green']))

plt.xlim(x1.min(), x1.max())

plt.ylim(x2.min(), x2.max())

for i, j in enumerate(nm.unique(y\_set)):

    plt.scatter(x\_set[y\_set == j, 0], x\_set[y\_set == j, 1],

        color = ListedColormap(['red', 'green'])(i), label = j)

plt.title('K-NN algorithm(Test set)')

plt.xlabel('Age')

plt.ylabel('Estimated Salary')

plt.legend()

plt.show()

**Conclusion :**

**1. Confusion Matrix:**

The confusion matrix (cm) will display how well the K-Nearest Neighbors (KNN) classifier has performed in terms of true positives, false positives, true negatives, and false negatives. This will help you understand how many of the test set instances were correctly or incorrectly classified.

For example, a confusion matrix might look like this:

plaintext

Copy code

[[50, 3],

[ 4, 43]]

Where:

* **50**: True positives (correctly predicted "class 0" in the test set).
* **43**: True negatives (correctly predicted "class 1").
* **4**: False positives (incorrectly predicted "class 1" instead of "class 0").
* **3**: False negatives (incorrectly predicted "class 0" instead of "class 1").

**2. Training Set Visualization:**

The first plot shows the classification results for the **training set**. This will display:

* **Red points**: Instances classified as class 0.
* **Green points**: Instances classified as class 1.
* The decision boundary is created by the KNN classifier, indicating the region that separates the two classes. Points on either side of this boundary are classified accordingly.

The plot will be a 2D graph where:

* The **x-axis** represents the "Age" feature.
* The **y-axis** represents the "Estimated Salary" feature.

**3. Test Set Visualization:**

The second plot shows the **test set** results, with a similar approach to the training set visualization. This plot will:

* Display **red and green points** representing the predicted classes.
* Show the decision boundary for the KNN classifier applied to the test data.

The test set plot will visually reflect how well the KNN classifier generalizes to unseen data. If the test points are scattered near the correct class and the decision boundary is well-defined, it means the classifier is performing well.

1. **Aim :** Implementing support vector machine

**Description :**

**Support Vector Machine (SVM)** is a powerful and versatile supervised machine learning algorithm used primarily for classification tasks, but can also be applied to regression problems. SVM is particularly effective for high-dimensional spaces and datasets where the classes are not linearly separable.

**Procedure :**

1. **Load and Prepare the Data**:
   * Load the dataset (e.g., Iris dataset, or your custom dataset).
   * Split the data into independent variables (features) and dependent variables (labels).
   * Split the dataset into **training** and **testing** sets, typically using an 80-20 or 70-30 split.
2. **Feature Scaling**:
   * SVM is sensitive to the scale of the input features. It's important to standardize or normalize the features to ensure that they all contribute equally to the model.
   * The **StandardScaler** from scikit-learn is often used to scale the data.
3. **Create the SVM Model**:
   * Import the **SVC** class from sklearn.svm and initialize the SVM classifier. The most commonly used kernel is the **RBF** (Radial Basis Function) kernel, but you can also experiment with linear or polynomial kernels.
   * Specify important parameters:
     + **kernel**: Specifies the kernel type (e.g., ‘linear’, ‘rbf’).
     + **C**: Regularization parameter.
     + **gamma**: Defines the influence of a single training example (only used for ‘rbf’, ‘poly’, and ‘sigmoid’ kernels).
4. **Training the Model**:
   * Train the model using the **fit()** method, which will optimize the hyperplane to separate the classes in the training data.
5. **Prediction**:
   * Use the **predict()** method to make predictions on the test set after the model has been trained.
6. **Evaluation**:
   * The performance of the model can be assessed using metrics such as **accuracy**, **confusion matrix**, **precision**, **recall**, and **F1 score**.
   * For visualizing results, especially in 2D data, you can plot the decision boundaries to understand how the SVM model is classifying the data.

**Source code :**

# Importing libraries

import numpy as np

import matplotlib.pyplot as plt

import pandas as pd

from sklearn import datasets

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

from sklearn.preprocessing import StandardScaler

from sklearn.svm import SVC

from sklearn.metrics import confusion\_matrix, accuracy\_score

# Load the iris dataset

data = datasets.load\_iris()

X = data.data # Features

y = data.target # Labels

# Split the dataset into training and test sets (80% train, 20% test)

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

# Feature scaling: SVM performs better with scaled data

scaler = StandardScaler()

X\_train = scaler.fit\_transform(X\_train)

X\_test = scaler.transform(X\_test)

# Initialize the Support Vector Machine classifier (using the RBF kernel)

classifier = SVC(kernel='rbf', random\_state=42)

# Fit the model to the training data

classifier.fit(X\_train, y\_train)

# Predicting the test set results

y\_pred = classifier.predict(X\_test)

# Create a confusion matrix

cm = confusion\_matrix(y\_test, y\_pred)

# Output the accuracy

accuracy = accuracy\_score(y\_test, y\_pred)

print(f"Accuracy of the SVM classifier: {accuracy \* 100:.2f}%")

# Display the confusion matrix

print("Confusion Matrix:")

print(cm)

# Optional: Visualize the results (for 2D data only, here we use only the first two features)

# This is mainly for educational purposes and works well with 2D datasets

# Plot the training set results

plt.figure(figsize=(8, 6))

plt.scatter(X\_train[:, 0], X\_train[:, 1], c=y\_train, cmap='winter', marker='o', label='Training set')

plt.title('SVM Training Set Results')

plt.xlabel('Feature 1')

plt.ylabel('Feature 2')

plt.legend(loc='upper left')

plt.show()

# Plot the test set results

plt.figure(figsize=(8, 6))

plt.scatter(X\_test[:, 0], X\_test[:, 1], c=y\_test, cmap='cool', marker='s', label='Test set')

plt.title('SVM Test Set Results')

plt.xlabel('Feature 1')

plt.ylabel('Feature 2')

plt.legend(loc='upper left')

plt.show()

**Conclusion :**

The output of the provided **Support Vector Machine (SVM)** code will include the following:

**1. Accuracy of the SVM classifier:**

This value indicates the proportion of correct predictions made by the model on the test set. The output might look something like this:

plaintext

Copy code

Accuracy of the SVM classifier: 100.00%

This means that the SVM classifier correctly classified all the test instances. However, keep in mind that this result might vary depending on the dataset split and other factors.

**2. Confusion Matrix:**

The confusion matrix will display the true versus predicted labels, which helps assess the classifier's performance. For the Iris dataset, the confusion matrix might look something like this:

plaintext

Copy code

Confusion Matrix:

[[13 0 0]

[ 0 12 1]

[ 0 0 14]]

Here:

* **13**: Instances of Class 0 (Setosa) correctly classified as Setosa.
* **12**: Instances of Class 1 (Versicolor) correctly classified as Versicolor.
* **14**: Instances of Class 2 (Virginica) correctly classified as Virginica.
* **1**: One instance of Class 1 (Versicolor) misclassified as Class 2 (Virginica).

**3. Visualization:**

The code also includes plotting for the training and test set results, which will display the following:

**Training Set Plot:**

* This plot shows the points in the training set, color-coded based on the class labels.
* Each point will be a circle ('o' marker) and will be plotted with Feature 1 on the x-axis and Feature 2 on the y-axis.

**Test Set Plot:**

* Similarly, the test set will be plotted with square markers ('s') for each data point.
* The points will be color-coded according to their actual class labels.

1. **Aim** : Implementiing The Naïve Bayes’ Classifier

**Description :**

The **Naïve Bayes** classifier is a family of probabilistic algorithms based on **Bayes' Theorem** with the assumption of **independence** between the features. Despite the "naïve" assumption of independence (which is rarely true in real-world data), Naïve Bayes often performs surprisingly well, especially for text classification tasks.

**Bayes' Theorem:**

Bayes' Theorem provides a way to calculate the posterior probability of a class given the observed

* P(A∣B) is the **posterior probability** of class CCC given the features B.
* P(B∣A) is the **likelihood**, which is the probability of observing the features B given class A.
* P(A) is the **prior probability** of class A, i.e., how likely the class is before observing the features.
* P(B) is the **evidence** or the probability of observing the features B across all classes.

**Procedure :**

**1. Preprocessing the Data:**

* **Data Collection**: First, gather the dataset that contains the features and the class labels.
* **Data Cleaning**: Handle missing values, outliers, or any irrelevant features in the dataset.
* **Feature Selection**: Select the relevant features for classification.
* **Feature Encoding**: If the dataset has categorical features, encode them using techniques like **One-Hot Encoding** or **Label Encoding**.

**2. Splitting the Data into Training and Testing Sets:**

* **Split the Data**: Divide the dataset into a training set (typically 70-80% of the data) and a test set (typically 20-30% of the data) to evaluate the model's performance.
* **Feature Scaling (Optional)**: For certain types of Naïve Bayes classifiers, such as **Gaussian Naïve Bayes**, you may need to scale features to ensure they're in a similar range. For others, like **Multinomial Naïve Bayes**, this is generally not required.

**3. Calculate Prior Probabilities:**

* **Prior Probability**: Calculate the probability of each class in the dataset. This is done by counting how many instances belong to each class and dividing by the total number of instances.

P(Ck)=Number of instances in class CkTotal number of instancesP(C\_k) = \frac{\text{Number of instances in class } C\_k}{\text{Total number of instances}}P(Ck​)=Total number of instancesNumber of instances in class Ck​​

This gives us an idea of how frequent each class is in the dataset.

**4. Calculate Likelihood (Conditional Probability):**

* **Likelihood**: For each feature and class, calculate the conditional probability that the feature value occurs given the class. This is done differently based on the type of data and the variant of Naïve Bayes you are using:

**5. Apply Bayes' Theorem to Calculate Posterior Probabilities:**

* **Posterior Probability**: Use Bayes' Theorem to calculate the probability of each class given the observed features. For a given data points

**6. Make Predictions:**

* **Classification**: For a new data point, calculate the posterior probability for each class . The class with the highest posterior probability is assigned to the data points.

**7. Model Evaluation:**

* **Accuracy**: Check how well the Naïve Bayes classifier performs on the test set by calculating the **accuracy**, which is the percentage of correct predictions.
* **Confusion Matrix**: The confusion matrix shows the counts of true positives, true negatives, false positives, and false negatives, providing insights into the performance of the classifier.

**Source Code :**

from sklearn.preprocessing import LabelEncoder

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

from sklearn.naive\_bayes import GaussianNB

from sklearn import preprocessing

from sklearn.metrics import classification\_report, confusion\_matrix

CGPA = ['g9', 'g8', 'g9', 'l8', 'g8', 'g9', 'l8', 'g9', 'g8', 'g8']

Inter = ['Y', 'N', 'N', 'N', 'Y', 'Y', 'Y', 'N', 'Y', 'Y']

PK = ['+++', '+', '==', '==', '+', '+', '+', '+++', '+', '==']

CS = ['G', 'M', 'P', 'G', 'M', 'M', 'P', 'G', 'G', 'G']

Job = ['Y', 'Y', 'N', 'N', 'Y', 'Y', 'N', 'Y', 'Y', 'Y']

# Creating LabelEncoder

le = preprocessing.LabelEncoder()

# Converting string labels into numbers

CGPA\_encoded = le.fit\_transform(CGPA)

Inter\_encoded = le.fit\_transform(Inter)

PK\_encoded = le.fit\_transform(PK)

CS\_encoded = le.fit\_transform(CS)

label = le.fit\_transform(Job)

# Print encoded labels

print("CGPA:", CGPA\_encoded)

print("Inter:", Inter\_encoded)

print("PK:", PK\_encoded)

print("CS:", CS\_encoded)

print("Job:", label)

# Prepare the features set

features = []

for i in range(len(CGPA\_encoded)):

features.append([CGPA\_encoded[i], Inter\_encoded[i], PK\_encoded[i], CS\_encoded[i]])

# Split data into training and testing sets

X\_train, X\_test, y\_train, y\_test = train\_test\_split(features, label, test\_size=0.30, random\_state=2)

# Create a Gaussian Naive Bayes classifier

model = GaussianNB()

# Train the model using the training sets

model.fit(X\_train, y\_train)

# Predict the output for the test set

y\_pred = model.predict(X\_test)

# Print classification report and confusion matrix

print(classification\_report(y\_test, y\_pred))

print(confusion\_matrix(y\_test, y\_pred))

# Predict if a new example gets the job or not

example = [[2, 0, 2, 0]] # New example for prediction

prediction = model.predict(example)

# Print the prediction result

if prediction == 1:

print("Predicted Value: Got JOB")

else:

print("Predicted Value: Didn't get JOB")

**Output :**

First 5 rows of scaled X\_train:

-----------------------------

[[-8.27308450e-01 -6.69702151e-01 -2.55977897e-01 -2.56851610e-01

-9.06779945e-01 9.78449295e-01 8.39209057e-01 -3.79916697e-01

-8.87195260e-01 7.05945857e-01 -1.58324589e+00 4.07316938e-02

1.59410455e+00 1.23000969e-01 1.03169828e+00 1.29602901e+00

2.08805272e-03 1.48588856e-03 -1.26472066e+00 -1.86270651e-01

-5.32188383e-01 -2.79190595e-01]

[ 1.59320521e-02 9.73285936e-01 -4.24880418e-01 3.60281784e+00

1.65492948e-03 -6.63743887e-04 1.05238881e+00 -1.51922385e-04

4.41798440e-01 2.66238660e-01 -3.38979557e-01 -1.10494808e+00

1.54125442e+00 2.07218285e+00 -3.47330564e-04 -4.52391042e-04

1.56682642e+00 1.66208665e+00 3.11735768e-01 -4.63144929e-01

1.87903388e+00 3.94142333e+00]

[-1.60027891e+00 5.97745802e-01 -1.57451427e-01 -2.80824712e-01

4.28617118e-02 1.34445276e+00 1.05238881e+00 -1.51922385e-04

4.41798440e-01 -1.27273653e+00 1.13480928e-01 1.64468337e+00

-6.78450849e-01 4.64107798e-01 -2.31921592e-02 -2.04216230e-01

-1.06829266e+00 -1.66212271e+00 5.43567595e-01 -9.88366682e-02

-5.32188383e-01 -2.79190595e-01]

[ 8.59172554e-01 -1.10783231e+00 -4.95256468e-01 -2.80824712e-01

1.65492948e-03 -6.63743887e-04 -1.50576826e+00 -1.06564763e+00

-8.87195260e-01 -8.33029331e-01 -7.91440041e-01 -1.33408403e+00

1.38270405e+00 -1.69376312e-01 -3.47330564e-04 -4.52391042e-04

2.08805272e-03 1.48588856e-03 -9.09245191e-01 -3.46566286e-01

-5.32188383e-01 -2.55611746e-01]

[-5.46228283e-01 9.88933442e-01 1.72862672e+00 -2.80824712e-01

8.02575038e-01 -6.63743887e-04 -1.50576826e+00 -2.27532045e-01

-1.10869421e+00 -1.27273653e+00 -7.91440041e-01 -4.17540215e-01

-1.49949594e-01 3.29504647e-04 -7.51215139e-01 -9.54338851e-01

-1.94666568e+00 1.48588856e-03 1.47089491e+00 1.30948292e-03

-5.32188383e-01 -2.79190595e-01]]

First 5 rows of scaled X\_test:

-----------------------------

[[-3.35418157e-01 9.10695914e-01 5.18158655e-01 2.28429722e+00

4.86027818e-01 4.66044439e-01 4.12849545e-01 7.72372587e-02

1.32779424e+00 4.86092258e-01 -5.65209799e-01 1.55299671e-01

1.38270405e+00 1.23000969e-01 2.73960077e-01 2.60859794e-01

1.56682642e+00 2.37425495e-01 2.34458492e-01 6.29779853e-01

1.87903388e+00 -2.55611746e-01]

[-1.67054895e+00 7.85515869e-01 2.55263037e-02 -2.80824712e-01

1.65492948e-03 -6.63743887e-04 -1.35099677e-02 7.72372587e-02

2.20299490e-01 1.58536025e+00 -1.12749314e-01 4.07316938e-02

-5.19900472e-01 -7.54130875e-01 -2.90629172e-01 6.58279129e-02

2.08805272e-03 1.48588856e-03 1.10814850e-01 3.23143057e-02

-5.32188383e-01 -2.79190595e-01]

[-1.10838862e+00 1.00458095e+00 1.94428824e-01 6.06180068e-01

-7.80161058e-01 -5.58765272e-01 4.12849545e-01 1.06773750e+00

6.63297390e-01 2.66238660e-01 1.35774726e+00 1.87381933e+00

4.84251912e-01 2.69189610e-01 1.55099183e-01 2.00849985e-01

-6.29106144e-01 -7.12348609e-01 5.89933961e-01 3.52905575e-01

1.87903388e+00 -2.79190595e-01]

[ 6.48362429e-01 2.22205668e-01 -9.03437559e-01 2.22610433e-01

-1.34994605e+00 -2.79138643e+00 -1.29258851e+00 -7.51473932e-02

-1.10869421e+00 -1.27273653e+00 3.39711170e-01 -4.17540215e-01

3.78551661e-01 1.09759191e+00 5.56254702e-01 7.85945628e-01

1.12763991e+00 1.66208665e+00 -2.13749708e-01 -8.27453190e-01

1.87903388e+00 -1.37717503e-01]

[ 1.35106285e+00 6.60335824e-01 -4.67106048e-01 -2.80824712e-01

1.65492948e-03 -6.63743887e-04 1.99669789e-01 3.05814237e-01

4.41798440e-01 2.66238660e-01 5.65941413e-01 1.30097944e+00

4.84251912e-01 9.02673720e-01 2.59102466e-01 5.45906390e-01

2.08805272e-03 1.48588856e-03 6.44484848e-02 -5.21434251e-01

-5.32188383e-01 -2.79190595e-01]]

-----------------------------

Mean cross-validation score:

0.999947254602036

-----------------------------

CGPA: [1 0 1 2 0 1 2 1 0 0]

Inter: [1 0 0 0 1 1 1 0 1 1]

PK: [1 0 2 2 0 0 0 1 0 2]

CS: [0 1 2 0 1 1 2 0 0 0]

Job: [1 1 0 0 1 1 0 1 1 1]

precision recall f1-score support

0 0.00 0.00 0.00 0.0

1 0.00 0.00 0.00 3.0

accuracy 0.00 3.0

macro avg 0.00 0.00 0.00 3.0

weighted avg 0.00 0.00 0.00 3.0

[[0 0]

[3 0]]

Predicted Value: Got JOB

1. **Aim :** Implementing Perceptron ANN model (Boolean function implementation)

**Description :**

The **Perceptron** is one of the simplest types of **Artificial Neural Networks (ANN)** and serves as the foundation for more complex neural network models. It is a binary classifier that learns to map inputs to binary outputs. The **Perceptron model** can be used to implement **Boolean functions**, which are functions that return either 0 or 1.

**Mathematical Model of the Perceptron:**

The output of the Perceptron is computed as:

Where:

* X(i) is the input value (either 0 or 1 for Boolean functions).
* W(i)​ is the weight for input x(i).
* b is the bias term.
* f is the activation function, often a **step function**:

The Perceptron classifies based on the threshold determined by the weighted sum of the inputs plus bias.

**Training the Perceptron (Supervised Learning):**

Training a Perceptron involves adjusting the weights and bias to minimize the error in the model’s predictions. The model is trained using labeled data (inputs paired with their correct output), and the weights are updated iteratively using an optimization rule called **Perceptron learning rule**.

The Perceptron learning rule is:

The weights are updated whenever there’s a misclassification until the Perceptron converges to an optimal set of weights.

**Source Code :**

import numpy as np

# Inputs and expected outputs for AND gate

X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])

Y = np.array([0, 0, 0, 1])

# Step function as activation function

step\_function = lambda x: 1 if x >= 0 else 0

# Initialize weights and bias

w = np.array([0.3, -0.2])

bias = -0.4

learning\_rate = 0.2

# Training the perceptron

epochs = 0

converged = False

while not converged:

converged = True

for i in range(len(X)):

# Compute the weighted sum of inputs

weighted\_sum = np.dot(X[i], w) + bias

# Apply step activation function

output = step\_function(weighted\_sum)

# Compute the error

error = Y[i] - output

# Update weights if the output is incorrect

if error != 0:

w += learning\_rate \* error \* X[i]

converged = False

epochs += 1

# Print weights update for each epoch

print(f"Epoch {epochs}:")

print("Weights:", w)

print()

# Print the final weights

print("Final weights:", w)

# Test the trained perceptron

print("\nTesting the trained perceptron:")

for i in range(len(X)):

weighted\_sum = np.dot(X[i], w) + bias

output = step\_function(weighted\_sum)

print(f"Input: {X[i]}, Predicted Output: {output}")

**Output :**

Epoch 1:

Weights: [0.5 0. ]

Epoch 2:

Weights: [0.5 0.2]

Epoch 3:

Weights: [0.3 0.2]

Epoch 4:

Weights: [0.3 0.2]

Final weights: [0.3 0.2]

Testing the trained perceptron:

Input: [0 0], Predicted Output: 0

Input: [0 1], Predicted Output: 0

Input: [1 0], Predicted Output: 0

Input: [1 1], Predicted Output: 1