

<b>Expr No: 1</b>	<b>Anaconda and Jupyter Notebook</b>
<b>Date:</b>	

**Aim:**

To write a program to familiarize with Anaconda and Jupyter Notebook by importing necessary Python libraries, loading the Iris dataset, performing exploratory data analysis (EDA), and understanding how different features contribute to flower classification.

**Algorithm:****Step 1: Launch Jupyter Notebook**

- Open Anaconda Navigator.
- Launch Jupyter Notebook.

**Step 2: Import Python Libraries**

- Import essential libraries like pandas, numpy, matplotlib.pyplot, seaborn, and sklearn.datasets.

**Step 3: Load the Dataset**

- Load the Iris dataset using `sklearn.datasets.load_iris()`.

**Step 4: Inspect the Dataset**

- Check dataset shape, feature names, target classes.
- Convert to a pandas DataFrame for easier manipulation.
- Check for missing values.

**Step 5: Perform Exploratory Data Analysis (EDA)**

- Use descriptive statistics (`.describe()`) to understand the data.
- Visualize data distributions using histograms, boxplots, or pairplots.
- Explore relationships between features using correlation matrices or scatter plots.

**Step 6: Interpret Feature Contribution**

- Observe which features separate the classes clearly.
- Identify patterns that may help in flower classification.

**Code:**

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.datasets import load_iris

iris = load_iris()
iris_df = pd.DataFrame(data=iris.data, columns=iris.feature_names)
```

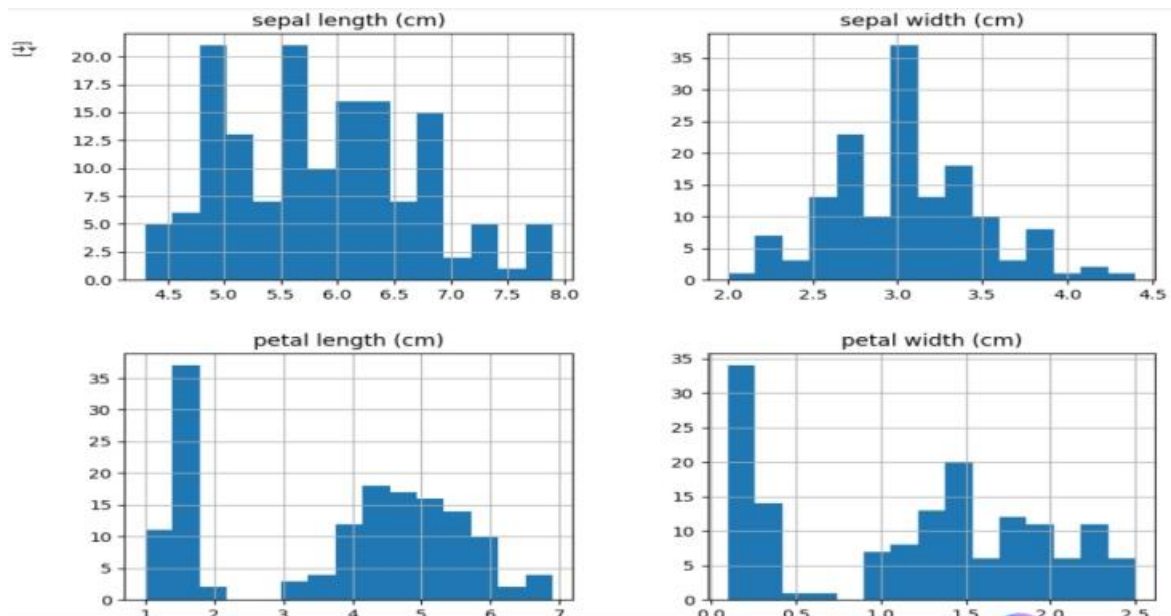
```
iris_df['species'] = pd.Categorical.from_codes(iris.target, iris.target_names)

print("Dataset Shape:", iris_df.shape)
print("\nFirst 5 rows:\n", iris_df.head())
print("\nDataset Info:\n")
iris_df.info()
print("\nCheck Missing Values:\n", iris_df.isnull().sum())
print("\nStatistical Summary:\n", iris_df.describe())

sns.pairplot(iris_df, hue='species')
plt.show()

plt.figure(figsize=(8,6))
sns.heatmap(iris_df.corr(), annot=True, cmap='coolwarm')
plt.title('Feature Correlation Heatmap')
plt.show()

plt.figure(figsize=(12,6))
sns.boxplot(x='species', y='sepal length (cm)', data=iris_df)
plt.title('Sepal Length Distribution per Species')
plt.show()
```

**Output :**

**Result:** Thus, the program to familiarize with Anaconda and Jupyter Notebook by importing necessary Python libraries, loading the Iris dataset was written, executed and verified.

<b>Expr No: 2</b>	<b>User interface and deploy ML models using Streamlit</b>
<b>Date:</b>	

**Aim:**

To write a program to create a web-based ML application using **Streamlit** for the frontend interface and **FastAPI** for backend model serving.

**Algorithm:****Step 1:** Train and save the ML model

- Use a simple ML algorithm (e.g., Logistic Regression).
- Save the model as a .pkl file using joblib or pickle.

**Step 2:** Build FastAPI backend

- Create endpoints (e.g., /predict) to handle model predictions.
- Load the trained model inside FastAPI and return predictions as JSON.

**Step 3:** Create Streamlit frontend

- Build a UI for user inputs (sepal/petal length, etc.).
- Send these inputs to FastAPI using HTTP requests (requests library).
- Display the prediction result.

**Step 4:** Run and test

- Start FastAPI backend (uvicorn main:app --reload).
- Start Streamlit frontend (streamlit run app.py).
- Enter input values → get model prediction.

**Code:****Trainmodel.py**

```
from sklearn.datasets import load_iris
from sklearn.linear_model import LogisticRegression
import joblib
iris = load_iris()
X, y = iris.data, iris.target
model = LogisticRegression(max_iter=200)
model.fit(X, y)
joblib.dump(model, "iris_model.pkl")
print(" Model trained and saved successfully!")
```

**main.py**

```
from fastapi import FastAPI
from pydantic import BaseModel
import joblib
import numpy as np
```

```
app = FastAPI(title="Iris Prediction API")
model = joblib.load("iris_model.pkl")
class IrisInput(BaseModel):
    sepal_length: float
    sepal_width: float
    petal_length: float
    petal_width: float
    @app.post("/predict")
    def predict(data: IrisInput):
        features = np.array([[data.sepal_length, data.sepal_width, data.petal_length,
        data.petal_width]])
        prediction = model.predict(features)[0]
        return {"predicted_class": int(prediction)}
app.py
import streamlit as st
import requests
st.title("Iris Flower Species Predictor")
st.write("Enter the details below to predict the flower species:")
sepal_length = st.number_input("Sepal Length (cm)", min_value=0.0, step=0.1)
sepal_width = st.number_input("Sepal Width (cm)", min_value=0.0, step=0.1)
petal_length = st.number_input("Petal Length (cm)", min_value=0.0, step=0.1)
petal_width = st.number_input("Petal Width (cm)", min_value=0.0, step=0.1)
if st.button("Predict"):
    url = "http://127.0.0.1:8000/predict"
    payload = {
        "sepal_length": sepal_length,
        "sepal_width": sepal_width,
        "petal_length": petal_length,
        "petal_width": petal_width
    }
    response = requests.post(url, json=payload)
    result = response.json()
    species = ["Setosa", "Versicolor", "Virginica"]
    st.success(f" Predicted Species: {species[result['predicted_class']]}")
```

**Output :**

## Iris Flower Species Predictor

Enter the details below to predict the flower species:

Sepal Length (cm)

5.10 - +

Sepal Width (cm)

3.20 - +

Petal Length (cm)

4.50 - +

Petal Width (cm)

0.20 - +

Predict

Predicted Species: Versicolor

**Result:** Thus, the program to understand and implement linear regression using PyTorch and TensorFlow, compare their syntax, and explore differences in their training workflows was written, executed and verified.

<b>Expr No: 3</b>	<b>PyTorch and TensorFlow</b>
<b>Date:</b>	

**Aim:**

To write a program to understand and implement linear regression using PyTorch and TensorFlow, compare their syntax, and explore differences in their training workflows.

**Algorithm:****Step 1: Data Preparation**

1. Generate or load a dataset with input features X and target Y.  
Example:  $Y = 2X + 3 + \text{noise}$ .
2. Convert the data into appropriate tensor/data structure:
  - PyTorch: `torch.tensor`
  - TensorFlow: `tf.Tensor`

**Step 2: Model Definition**

1. Define a linear regression model:
  - PyTorch: Use `torch.nn.Linear`
  - TensorFlow: Use `tf.keras.Sequential` with a Dense layer
2. Initialize model parameters (weights and bias).

**Step 3: Loss Function**

1. Use Mean Squared Error (MSE) as the loss function:
  - PyTorch: `torch.nn.MSELoss()`
  - TensorFlow: `tf.keras.losses.MeanSquaredError()`

**Step 4: Optimizer**

1. Select an optimizer for gradient descent to update weights:
  - PyTorch: `torch.optim.SGD(model.parameters(), lr=0.01)`
  - TensorFlow: `tf.keras.optimizers.SGD(learning_rate=0.01)`

**Step 5: Training**

1. Train the model for multiple epochs:  
PyTorch Workflow:

- Loop over epochs
- Zero the gradients: `optimizer.zero_grad()`
- Forward pass: compute predictions
- Compute loss
- Backward pass: `loss.backward()`
- Update weights: `optimizer.step()`

## TensorFlow Workflow:

- Use high-level API: `model.fit(X, Y, epochs=100)`

- OR custom loop:
  - Open `tf.GradientTape()`
  - Compute predictions and loss
  - Compute gradients
  - Apply gradients using `optimizer.apply_gradients()`

**Step 6: Model Evaluation**

1. Compare predicted vs actual values.
2. Plot regression line over data points.
3. Analyze feature contribution and slope/intercept values.
4. Compare syntax differences and workflow differences between PyTorch and TensorFlow.

**Code:**

```
import torch
import torch.nn as nn
import torch.optim as optim

x_train = torch.tensor([[1.0], [2.0], [3.0], [4.0]])
y_train = torch.tensor([[3.0], [5.0], [7.0], [9.0]])

model = nn.Linear(1, 1)
criterion = nn.MSELoss()
optimizer = optim.SGD(model.parameters(), lr=0.01)

for epoch in range(100):
    model.train()

    optimizer.zero_grad()
    outputs = model(x_train)
    loss = criterion(outputs, y_train)

    loss.backward()
    optimizer.step()

    if (epoch+1) % 20 == 0:
        print(f'Epoch [{epoch+1}/100], Loss: {loss.item():.4f}')

print(f'Learned weight: {model.weight.item():.4f}, bias: {model.bias.item():.4f}')
```

```
import tensorflow as tf

x_train = tf.constant([[1.0], [2.0], [3.0], [4.0]])
y_train = tf.constant([[3.0], [5.0], [7.0], [9.0]])

W = tf.Variable(tf.random.normal([1, 1]))
b = tf.Variable(tf.random.normal([1]))
optimizer = tf.optimizers.SGD(learning_rate=0.01)

for epoch in range(100):
    with tf.GradientTape() as tape:
        y_pred = tf.matmul(x_train, W) + b
        loss = tf.reduce_mean(tf.square(y_train - y_pred))
        gradients = tape.gradient(loss, [W, b])
        optimizer.apply_gradients(zip(gradients, [W, b]))
    if (epoch+1) % 20 == 0:
        print(f'Epoch {(epoch+1)/100}, Loss: {loss.numpy():.4f}')
print(f'Learned weight: {W.numpy()[0][0]:.4f}, bias: {b.numpy()[0]:.4f}')
```

**Output :**

```
Epoch [20/100], Loss: 0.0432
Epoch [40/100], Loss: 0.0173
Epoch [60/100], Loss: 0.0153
Epoch [80/100], Loss: 0.0136
Epoch [100/100], Loss: 0.0121
Learned weight: 1.9089, bias: 1.2680
```

---

```
Epoch [20/100], Loss: 0.0353
Epoch [40/100], Loss: 0.0019
Epoch [60/100], Loss: 0.0016
Epoch [80/100], Loss: 0.0015
Epoch [100/100], Loss: 0.0013
Learned weight: 1.9702, bias: 1.0875
```

**Result:** Thus, the program to understand and implement linear regression using PyTorch and TensorFlow, compare their syntax, and explore differences in their training workflows was written, executed and verified.

Expr No: 4	K-Means Clustering
Date:	

**Aim:**

To determine the optimal number of clusters using the Elbow Method, visualize centroid updates during K-Means iterations, and assign both existing and new data points to their respective clusters.

**Algorithm:****Step 1: Import Python Libraries**

- Import essential libraries like pandas, numpy, matplotlib.pyplot and sklearn.clusters.K-Means

**Step 2: Define the Dataset**

- Create a small two-dimensional dataset X representing sample data points for clustering.

**Step 3: Determine the Optimal Number of Clusters (Elbow Method)**

- For each value of k in a specified range (1 to 6), apply the KMeans algorithm.
- Compute and record the inertia (within-cluster sum of squared distances).
- Plot the relationship between k and inertia.
- Identify the “elbow point”, where the reduction in inertia becomes marginal, indicating the most appropriate number of clusters.

**Step 4: Initialize Cluster Centroids**

- Select the optimal number of clusters,  $k = 3$ .
- Randomly initialize k centroids from the existing data points.

**Step 5: Iterative K-Means Process**

- Assign each point to the nearest centroid, update centroids as the mean of assigned points, visualize the clusters after each iteration, and repeat until centroids stop changing

**Step 6: Assign a New Data Point to a Cluster**

- Introduce a new data point (e.g., [7, 3]).
- Compute its distance from all final centroids.
- Assign it to the cluster corresponding to the smallest distance value.

**Step 7: Final Visualization**

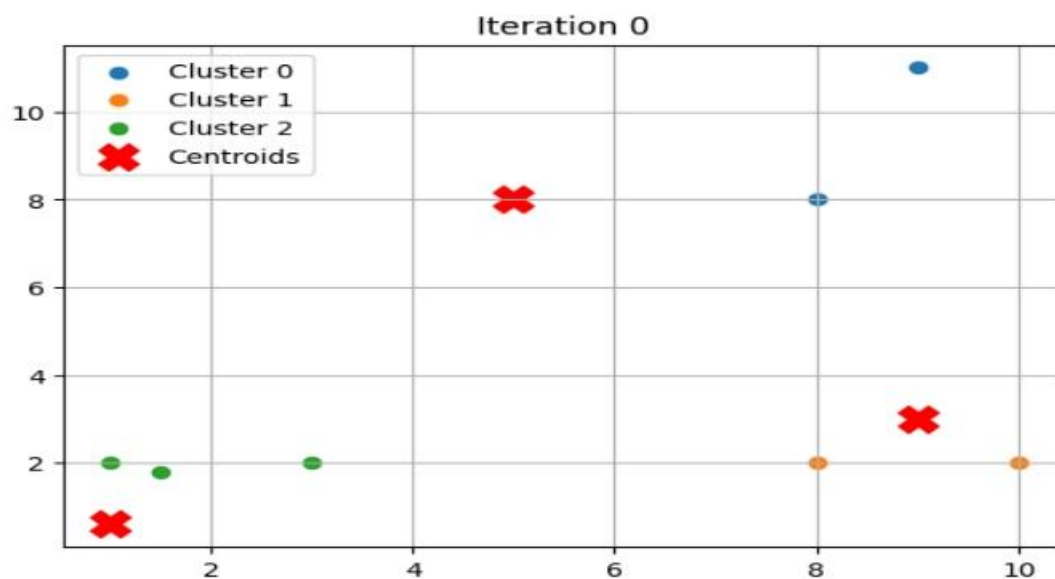
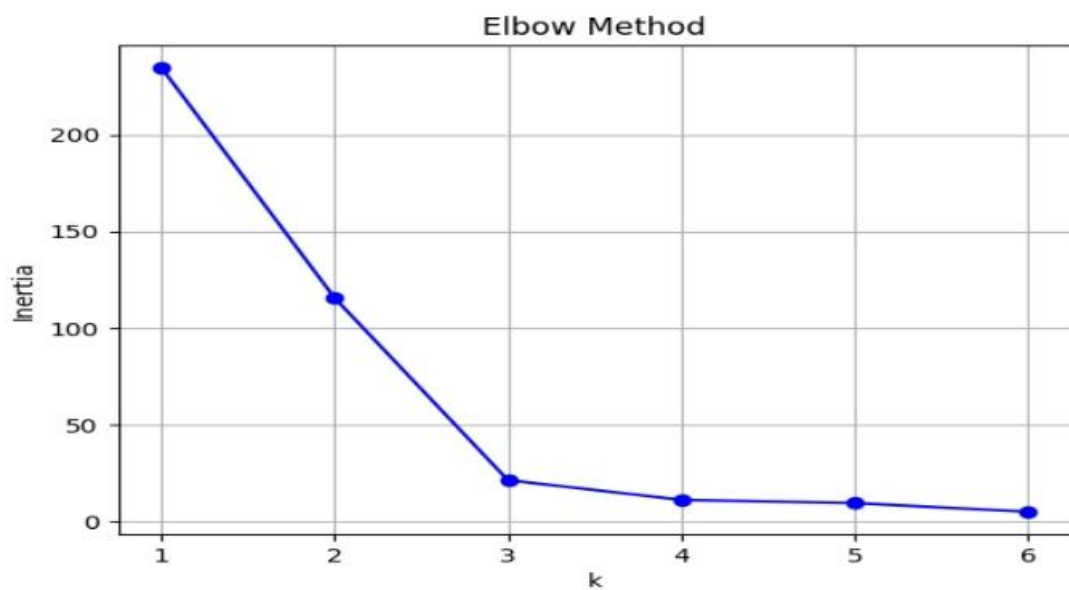
- Plot the final clusters with distinct colors.
- Mark the centroids using red “X” markers.
- Indicate the new data point using a black “\*” symbol.
- Display the finalized clustering pattern.

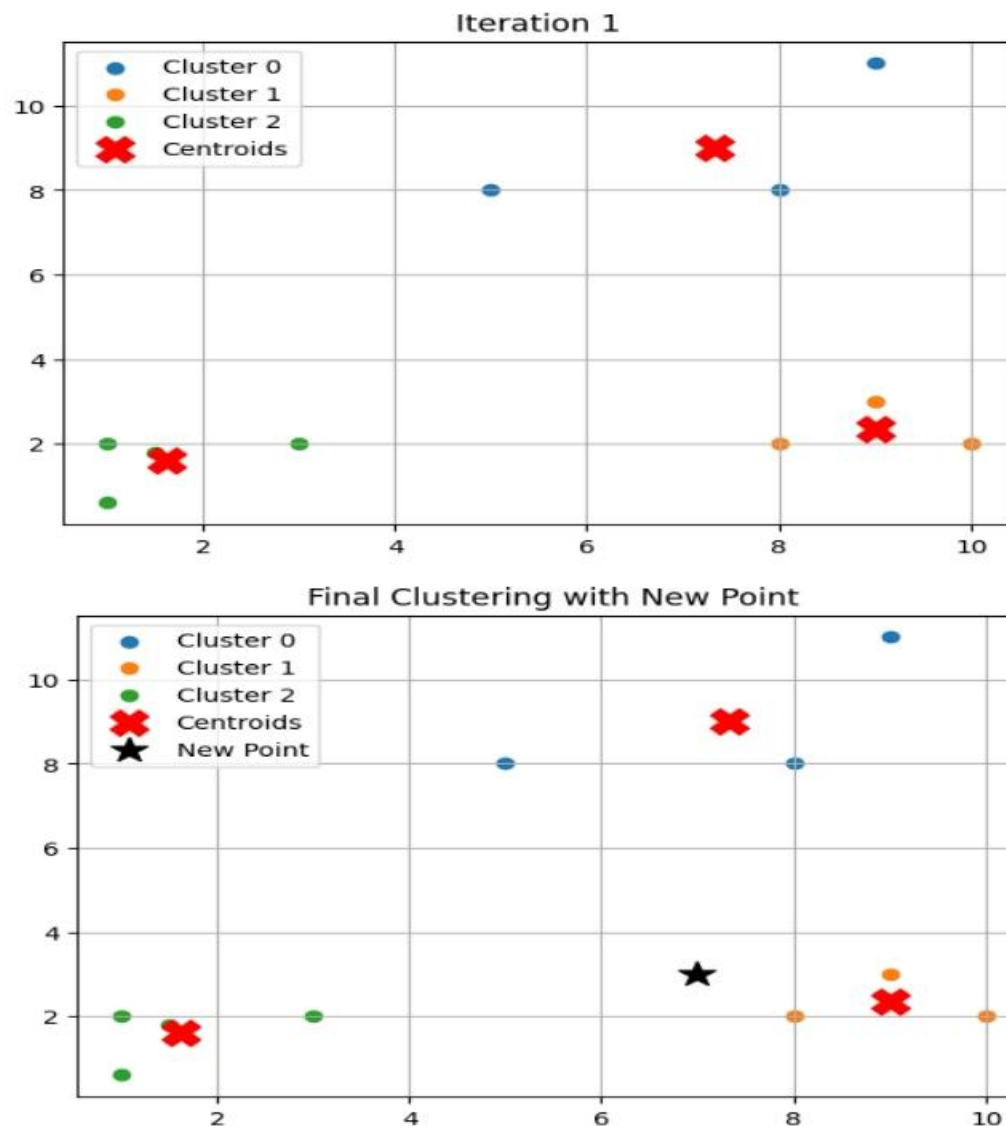
**Code:**

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans
X = np.array([
    [1, 2], [1.5, 1.8], [5, 8], [8, 8],
    [1, 0.6], [9, 11], [8, 2], [10, 2],
    [9, 3], [3, 2]
])
inertias = []
for k in range(1, 7):
    km = KMeans(n_clusters=k, n_init='auto', random_state=0)
    km.fit(X)
    inertias.append(km.inertia_)
plt.plot(range(1, 7), inertias, 'bo-')
plt.xlabel('k'), plt.ylabel('Inertia'), plt.title('Elbow Method')
plt.grid(), plt.show()
k = 3
np.random.seed(0)
centroids = X[np.random.choice(len(X), k, replace=False)]
def plot_clusters(X, centroids, labels, title):
    for i in range(k):
        plt.scatter(*X[labels == i].T, label=f'Cluster {i}')
    plt.scatter(*centroids.T, c='red', marker='X', s=200, label='Centroids')
    plt.title(title)
    plt.legend(), plt.grid(True)
    plt.show()
for i in range(5):
    distances = np.linalg.norm(X[:, None] - centroids, axis=2)
    labels = np.argmin(distances, axis=1)
    plot_clusters(X, centroids, labels, f'Iteration {i}')
    new_centroids = np.array([X[labels == j].mean(axis=0) for j in range(k)])
    if np.allclose(centroids, new_centroids):
        break
    centroids = new_centroids
new_point = np.array([[7, 3]])
dist = np.linalg.norm(new_point - centroids, axis=1)
assigned = np.argmin(dist)
```

```
print(f"New point {new_point[0]} assigned to Cluster {assigned}")
for i in range(k):
    plt.scatter(*X[labels == i].T, label=f'Cluster {i}')
plt.scatter(*centroids.T, c='red', marker='X', s=200, label='Centroids')
plt.scatter(*new_point.T, c='black', marker='*', s=200, label='New Point')
plt.title('Final Clustering with New Point')
plt.legend(), plt.grid(True)
plt.show()
```

**Output :**





**Result:** Thus, the program To determine the optimal number of clusters using the Elbow Method, by importing necessary Python libraries was written, executed and verified.

Expr No: 5	Hierarchial Clustering
Date:	

**Aim:**

To perform **Hierarchical Clustering using the Single Linkage method**, generate distance matrices at each step, and visualize the clustering process using a **dendrogram**.

**Algorithm:**

**Step 1:** Import the required libraries – numpy, pandas, scipy.cluster.hierarchy, and matplotlib.pyplot.

**Step 2:** Define a set of 2D data points and assign labels (A–H) for identification.

**Step 3:** Compute the pairwise Euclidean distances between all points using `pdist()`.

**Step 4:** Convert the computed distances into a matrix form using `squareform()` and display it as a distance matrix.

**Step 5:** Apply the Hierarchical Clustering algorithm using the `linkage()` function with the method set to 'single' (Single Linkage).

**Step 6:** Initialize each point as an individual cluster.

**Step 7:** Iteratively merge the two closest clusters based on the minimum distance between their points.

**Step 8:** After each merge, display the updated cluster combinations and print the new distance matrix.

**Step 9:** Continue merging until all data points are grouped into a single cluster.

**Step 10:** Plot a dendrogram using the `dendrogram()` function to visualize how the clusters merge at different distances.

**Code:**

```
import numpy as np
import pandas as pd
from scipy.spatial.distance import pdist, squareform
from scipy.cluster.hierarchy import linkage, dendrogram
import matplotlib.pyplot as plt

points = np.array([
    [0, 0], [1, 1], [2, 1], [5, 5],
    [6, 5], [20, 20], [26, 25], [28, 29]
])

labels = ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H']
Z = linkage(points, method='single')
dist_matrix = squareform(pdist(points))
df = pd.DataFrame(dist_matrix, index=labels, columns=labels)
```

```
print("Distance Matrix - Step 1")
print(df.round(2))
clusters = {i: [label] for i, label in enumerate(labels)}
active_ids = list(range(len(labels)))
def single_linkage(cl1, cl2):
    return min(np.linalg.norm(points[i] - points[j]) for i in cl1 for j in cl2)
for step, (i, j, dist, _) in enumerate(Z, 1):
    i, j = int(i), int(j)
    new_id = max(clusters) + 1
    new_cluster = clusters[i] + clusters[j]
    clusters[new_id] = new_cluster
    active_ids = [id_ for id_ in active_ids if id_ not in (i, j)] + [new_id]
    print(f"\nStep {step}: Merging {clusters[i]} and {clusters[j]} at distance {dist:.2f} → New
cluster: {' '.join(new_cluster)}")
    names = [' '.join(clusters[k]) for k in active_ids]
    mat = np.zeros((len(active_ids), len(active_ids)))
    for a in range(len(active_ids)):
        for b in range(len(active_ids)):
            if a != b:
                mat[a, b] = single_linkage(
                    [labels.index(x) for x in clusters[active_ids[a]]],
                    [labels.index(x) for x in clusters[active_ids[b]]]
                )
    print(f"\nDistance Matrix - Step {step + 1}")
    print(pd.DataFrame(mat, index=names, columns=names).round(2))
plt.figure(figsize=(10, 6))
dendrogram(Z, labels=labels)
plt.title("Hierarchical Clustering Dendrogram (Single Linkage)")
plt.xlabel("Data Point")
plt.ylabel("Distance")
plt.grid(True)
plt.tight_layout()
plt.show()
```

**Output:**

Step 1: Merging ['B'] and ['C'] at distance 1.00 → New cluster: BC

Distance Matrix - Step 2

	A	D	E	F	G	H	BC
A	0.00	7.07	7.81	28.28	36.07	40.31	1.41
D	7.07	0.00	1.00	21.21	29.00	33.24	5.00
E	7.81	1.00	0.00	20.52	28.28	32.56	5.66
F	28.28	21.21	20.52	0.00	7.81	12.04	26.17
G	36.07	29.00	28.28	7.81	0.00	4.47	33.94
H	40.31	33.24	32.56	12.04	4.47	0.00	38.21
BC	1.41	5.00	5.66	26.17	33.94	38.21	0.00


Step 2: Merging ['D'] and ['E'] at distance 1.00 → New cluster: DE

Distance Matrix - Step 3

	A	F	G	H	BC	DE
A	0.00	28.28	36.07	40.31	1.41	7.07
F	28.28	0.00	7.81	12.04	26.17	20.52
G	36.07	7.81	0.00	4.47	33.94	28.28
H	40.31	12.04	4.47	0.00	38.21	32.56
BC	1.41	26.17	33.94	38.21	0.00	5.00
DE	7.07	20.52	28.28	32.56	5.00	0.00

Step 3: Merging ['A'] and ['B', 'C'] at distance 1.41 → New cluster: ABC

Distance Matrix - Step 4



	F	G	H	DE	ABC
F	0.00	7.81	12.04	20.52	26.17
G	7.81	0.00	4.47	28.28	33.94
H	12.04	4.47	0.00	32.56	38.21
DE	20.52	28.28	32.56	0.00	5.00
ABC	26.17	33.94	38.21	5.00	0.00

Step 4: Merging ['G'] and ['H'] at distance 4.47 → New cluster: GH

Distance Matrix - Step 5

	F	DE	ABC	GH
F	0.00	20.52	26.17	7.81
DE	20.52	0.00	5.00	28.28
ABC	26.17	5.00	0.00	33.94
GH	7.81	28.28	33.94	0.00

Step 5: Merging ['D', 'E'] and ['A', 'B', 'C'] at distance 5.00 → New cluster: DEABC

Distance Matrix - Step 6

	F	GH	DEABC
F	0.00	7.81	20.52
GH	7.81	0.00	28.28
DEABC	20.52	28.28	0.00

Step 6: Merging ['F'] and ['G', 'H'] at distance 7.81 → New cluster: FGH

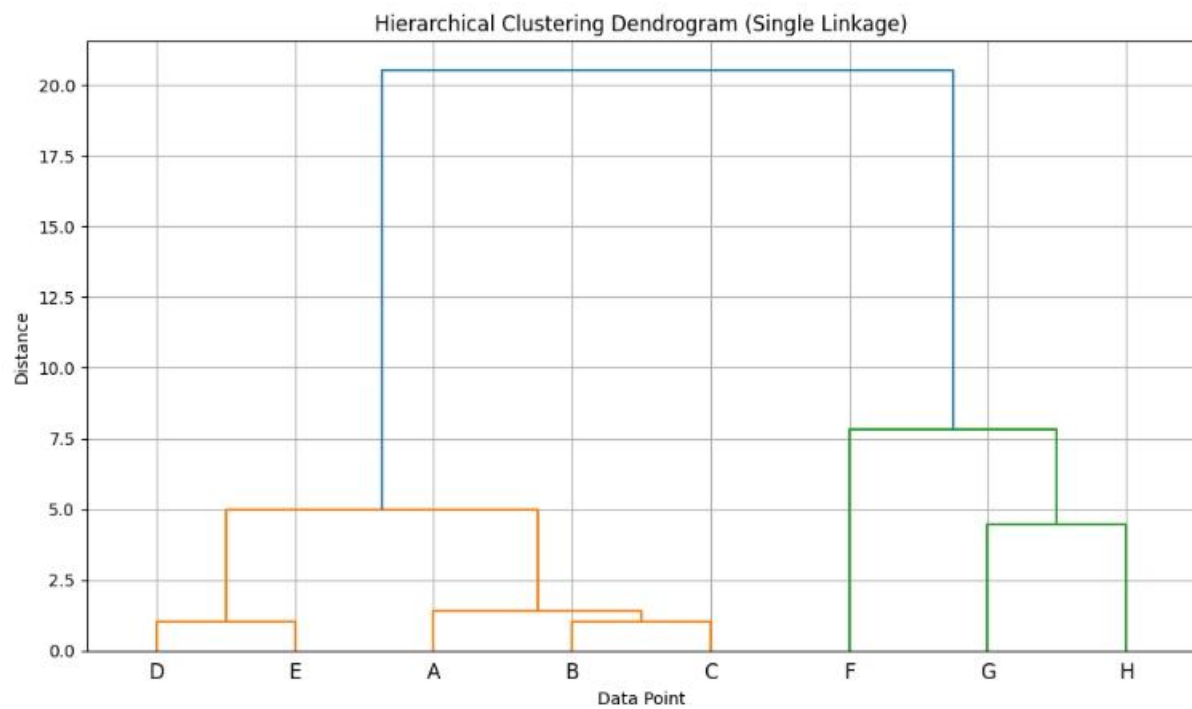
Distance Matrix - Step 7

	DEABC	FGH
DEABC	0.00	20.52
FGH	20.52	0.00

Step 7: Merging ['D', 'E', 'A', 'B', 'C'] and ['F', 'G', 'H'] at distance 20.52 → New cluster: DEABCFGH

Distance Matrix - Step 8

	DEABCFGH
DEABCFGH	0.0



**Result:** Thus, the program to perform Hierarchical Clustering using the Single Linkage method by importing necessary Python libraries was written, executed and verified.

Expr No: 6	DBSCAN Clustering
Date:	

**AIM:**

To implement the DBSCAN (Density-Based Spatial Clustering of Applications with Noise) algorithm to identify core, border, and noise points in a dataset and visualize the clustering structure.

**Algorithm:**

**Step 1:** Import necessary libraries — numpy, matplotlib.pyplot, and DBSCAN from sklearn.cluster.

**Step 2:** Define the dataset X as a set of 2D coordinate points representing data to be clustered.

**Step 3:** Initialize the **DBSCAN** model by setting two parameters:

- eps: The maximum distance between two samples for one to be considered as in the neighborhood of the other.
- min\_samples: The minimum number of points required to form a dense region (cluster).

**Step 4:** Fit the DBSCAN model to the dataset using fit(X) and obtain cluster labels using db.labels\_.

**Step 5:** Identify **core points**, **border points**, and **noise points**:

- Core points have at least min\_samples neighbors within eps distance.
- Border points are close to a core point but have fewer than min\_samples neighbors.
- Noise points do not belong to any cluster.

**Step 6:** Plot the clustering results:

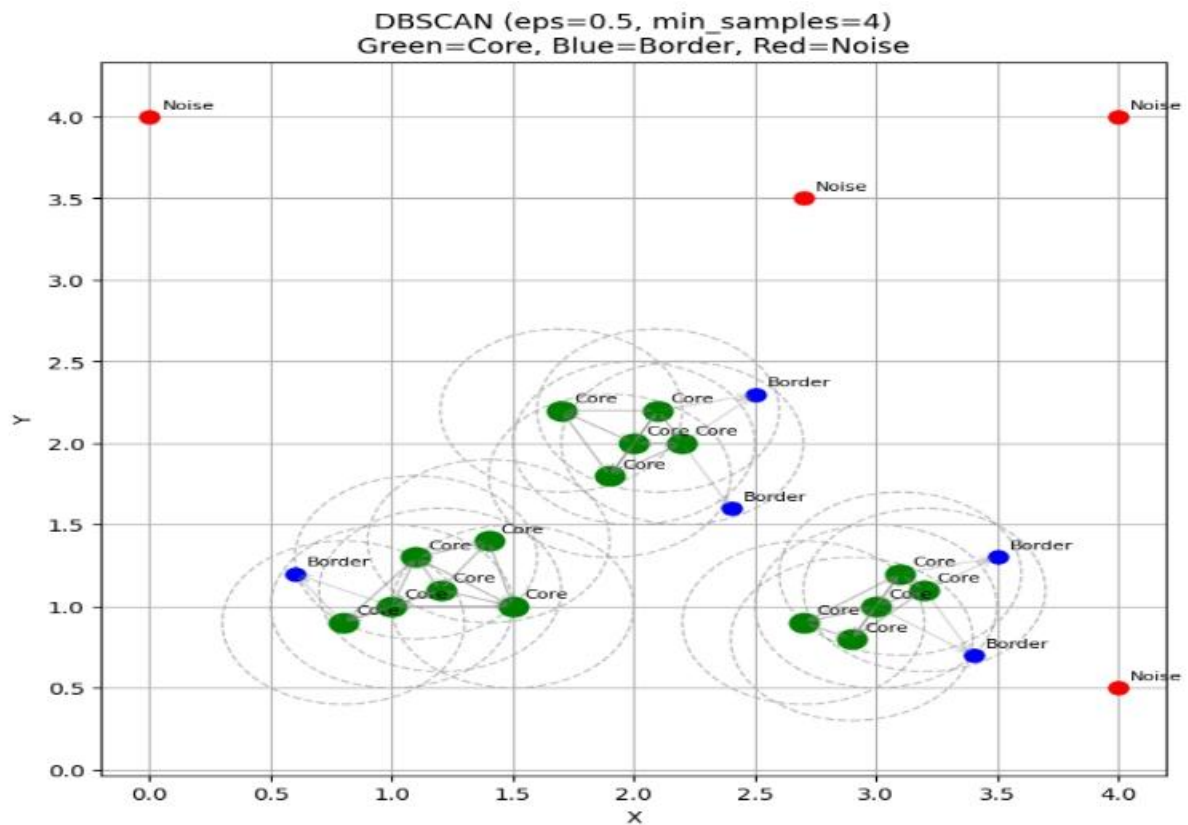
- Green circles represent **core points** (dense cluster centers).
- Blue circles represent **border points** (points near core areas).
- Red circles represent **noise points** (outliers).
- Dashed gray circles show the neighborhood radius (eps) around each core point.
- Light gray arrows indicate neighborhood connections between core points.

**Step 7:** Display the plot with appropriate labels, title, and grid for clarity.

**Code:**

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.cluster import DBSCAN
X = np.array([
    [1.0, 1.0], [1.2, 1.1], [0.8, 0.9], [1.1, 1.3],
    [1.5, 1.0], [0.6, 1.2], [1.4, 1.4],
```

```
[2.0, 2.0], [2.1, 2.2], [1.9, 1.8], [2.2, 2.0],
[2.5, 2.3], [1.7, 2.2], [2.4, 1.6],
[3.0, 1.0], [3.1, 1.2], [2.9, 0.8], [3.2, 1.1],
[3.5, 1.3], [2.7, 0.9], [3.4, 0.7],
[0.0, 4.0], [4.0, 4.0], [2.7, 3.5], [4.0, 0.5]
])
eps = 0.5
min_samples = 4
db = DBSCAN(eps=eps, min_samples=min_samples).fit(X)
labels = db.labels_
core_mask = np.zeros_like(labels, dtype=bool)
core_mask[db.core_sample_indices_] = True
plt.figure(figsize=(8, 8))
ax = plt.gca()
for i, (x, y) in enumerate(X):
    if labels[i] == -1:
        plt.plot(x, y, 'o', color='red', markersize=8)
        plt.text(x + 0.05, y + 0.05, 'Noise', fontsize=8)
    elif core_mask[i]:
        plt.plot(x, y, 'o', color='green', markersize=12)
        plt.text(x + 0.05, y + 0.05, 'Core', fontsize=8)
        ax.add_patch(plt.Circle((x, y), eps, color='gray', fill=False, linestyle='--', alpha=0.5))
    else:
        plt.plot(x, y, 'o', color='blue', markersize=8)
        plt.text(x + 0.05, y + 0.05, 'Border', fontsize=8)
for core_pt in X[core_mask]:
    for other_pt in X:
        if np.linalg.norm(core_pt - other_pt) <= eps and not np.array_equal(core_pt, other_pt):
            ax.annotate("",
                        xy=other_pt, xycoords='data',
                        xytext=core_pt, textcoords='data',
                        arrowprops=dict(arrowstyle="->", color='gray', alpha=0.3))
plt.title(f"DBSCAN (eps={eps}, min_samples={min_samples})\nGreen=Core, Blue=Border,
Red=Noise")
plt.xlabel("X")
plt.ylabel("Y")
plt.grid(True)
plt.axis('equal')
plt.show()
```

**Output:**

**Result:** Thus, the program to implement the DBSCAN by importing necessary Python libraries was written, executed and verified.

<b>Expr No: 7</b>	<b>Principal Component Analysis (PCA)</b>
<b>Date:</b>	

**Aim:**

To write a Python program to perform Principal Component Analysis (PCA) manually using linear algebra concepts — including mean centering, covariance computation, eigen decomposition, and projection onto the principal component

**Algorithm:****Step 1: Import the required libraries**

- Import numpy, pandas, and matplotlib.pyplot for numerical computation, data handling, and visualization..

**Step 2: Create the Dataset**

- Define a small 2D dataset manually using NumPy array.
- Convert it into a pandas DataFrame for easier display.

**Step 3: Mean Center the Data**

- Compute the mean of each feature.
- Subtract the mean from the dataset to center it around the origin.

**Step 4: Compute Covariance Matrix**

- Use np.cov() to find the covariance between features.

**Step 5: Perform Eigen Decomposition**

- Compute eigenvalues and eigenvectors using np.linalg.eigh().
- Sort them in descending order of eigenvalues.

**Step 6: Determine Explained Variance****Step 7: Project Data onto Principal Component****Step 8: Visualize****Code:**

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
X = np.array([
    [2.5, 2.4],
    [0.5, 0.7],
    [2.2, 2.9],
    [1.9, 2.2],
    [3.1, 3.0],
    [2.3, 2.7],
```

[2.0, 1.6],

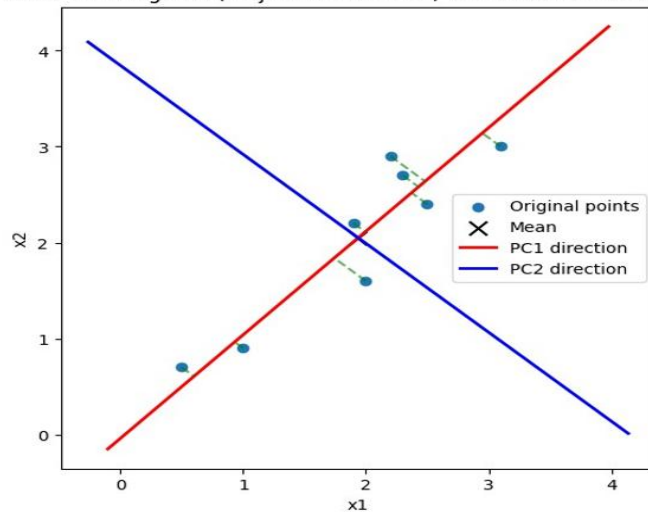
[1.0, 0.9] ])

```
df = pd.DataFrame(X, columns=['x1','x2']) print("Original
2D data:\n", df)
X_mean = X.mean(axis=0) Xc = X - X_mean cov =
np.cov(Xc, rowvar=False) eigvals, eigvecs =
np.linalg.eigh(cov) order = eigvals.argsort()[::-1]
eigvals = eigvals[order] eigvecs = eigvecs[:, order]
explained = eigvals / eigvals.sum() print("\nExplained
variance ratio (per PC):", explained)
W = eigvecs[:, :1]
X_pca_1d = Xc.dot(W) print("\nProjected 1D
coordinates:\n", X_pca_1d)
plt.figure(figsize=(6,6)) plt.scatter(X[:,0], X[:,1], label="Original points")
plt.scatter(X_mean[0], X_mean[1], color="black", s=100, marker="x", label="Mean")
vec1 = eigvecs[:,0] * 3 plt.plot([X_mean[0]-vec1[0], X_mean[0]+vec1[0]],
[X_mean[1]-vec1[1], X_mean[1]+vec1[1]],
'r-', lw=2, label="PC1 direction") vec2 =
eigvecs[:,1] * 3 plt.plot([X_mean[0]-vec2[0],
X_mean[0]+vec2[0]],
[X_mean[1]-vec2[1], X_mean[1]+vec2[1]],
'b-', lw=2, label="PC2 direction")
for i in range(len(X)):
    proj = X_mean + (X_pca_1d[i] * eigvecs[:,0])
    plt.plot([X[i,0], proj[0]], [X[i,1], proj[1]], 'g--', alpha=0.6)

plt.xlabel("x1"); plt.ylabel("x2") plt.title("2D → 1D using PCA (Projection on
to PC1, with Mean & PC2 shown)") plt.legend() plt.axis('equal') plt.show()
```

**Output:**

2D → 1D using PCA (Projection onto PC1, with Mean & PC2 shown)

**Result:**

Thus, the program to perform manual PCA using eigen decomposition was successfully written, executed, and verified.

<b>Expr No: 8</b>	<b>Building Neural Networks with Numpy</b>
<b>Date:</b>	

**Aim:**

To write a Python program to build and understand a 2-layer feedforward neural network using NumPy, applying ReLU and Softmax activations to classify input data (e.g., Spam vs. Not Spam emails).

**Algorithm:****Step 1: Import Required Libraries and Create Dataset**

- Import the NumPy library for numerical operations.
- Define an input matrix representing email features (like presence of certain words).

**Step 2: Initialize Network Parameters**

- Randomly initialize weights and biases for the hidden layer ( $W_1$ ,  $b_1$ ) and output layer ( $W_2$ ,  $b_2$ ).
- Use `np.random.seed()` for reproducibility.

**Step 3: Define Activation Functions**

- Use ReLU (Rectified Linear Unit) for the hidden layer to introduce non-linearity.
- Use Softmax for the output layer to convert scores into probability values.

**Step 4: Compute Covariance Matrix**

- Use `np.cov()` to find the covariance between features.

**Step 5: Perform Forward Propagation**

- Compute  $Z_1 = X \cdot W_1 + b_1$  for hidden layer linear transformation.
- Apply  $A_1 = \text{ReLU}(Z_1)$  for non-linear activation.
- Compute  $Z_2 = A_1 \cdot W_2 + b_2$  for output layer transformation.
- Apply  $A_2 = \text{Softmax}(Z_2)$  to obtain final output probabilities.

**Step 6: Display Results****Step 7: Analyze Model Behavior****Code:**

```
import numpy as np
```

```
np.random.seed(42)
```

```
W1 = np.random.randn(3, 4)
```

```
b1 = np.random.randn(1, 4)
```

```
W2 = np.random.randn(4, 2)
```

```
b2 = np.random.randn(1, 2)
```

```
def relu(x):
```

```
return np.maximum(0, x)
```

```
def softmax(x):
```

```
    exp_vals = np.exp(x - np.max(x, axis=1, keepdims=True))
```

```
    return exp_vals / np.sum(exp_vals, axis=1, keepdims=True)
```

```
Z1 = np.dot(X, W1) + b1
```

```
A1 = relu(Z1)
```

```
Z2 = np.dot(A1, W2) + b2
```

```
A2 = softmax(Z2)
```

```
print("Input (Emails):\n", X)
```

```
print("\nHidden Layer Activations:\n", A1)
```

```
print("\nOutput Probabilities [Not Spam, Spam]:\n", A2)
```

```
print("\nPredicted Labels:", np.argmax(A2, axis=1))
```

```
Z1 = np.dot(X, W1) + b1
```

```
A1 = relu(Z1)
```

```
Z2 = np.dot(A1, W2) + b20
```

```
A2 = softmax(Z2)
```

```
print("Input (Emails):\n", X)
```

```
print("\nHidden Layer Activations:\n", A1)
```

```
print("\nOutput Probabilities [Not Spam, Spam]:\n", A2)
```

```
print("\nPredicted Labels:", np.argmax(A2, axis=1))
```

### Output :

```
Input (Emails):
[[2 1 0]
 [0 0 1]
 [0 1 0]
 [1 0 0]]

Hidden Layer Activations:
[[1.0012372  0.          1.14967206  3.25120691]
 [0.          0.          0.          0.          ]
 [0.0078089  0.          0.          0.2051472 ]
 [0.73867642 0.          0.          0.96074233]]

Output Probabilities [Not Spam, Spam]:
[[0.99193607 0.00806393]
 [0.34179499 0.65820501]
 [0.41107286 0.58892714]
 [0.4496973  0.5503027 ]]

Predicted Labels: [0 1 1 1]
```

### Result:

Thus, the program to build and execute a 2-layer neural network using NumPy was successfully written, executed, and verified.

<b>Expr No: 9</b>	<b>Train Neural Networks with Backpropagation</b>
<b>Date:</b>	

**Aim:**

Write a python program to Train Neural Networks with Backpropagation

**Algorithm:**

Step 1: Import the necessary libraries such as NumPy and TensorFlow for numerical computation and neural network building.

Step 2: Define the input data and their corresponding target outputs for classification.

Step 3: Build a neural network model using TensorFlow's Sequential API with input, hidden, and output layers.

Step 4: Select an appropriate activation function for each layer, such as ReLU for the hidden layer and Softmax for the output layer.

Step 5: Choose an optimizer like Adam or SGD and compile the model with a suitable loss function and evaluation metrics.

Step 6: Train the model using the training data for a fixed number of epochs and batch size.

Step 7: Evaluate the trained model to compute the loss and accuracy values.

Step 8: Use the trained model to make predictions on the dataset and compare the predicted outputs with the true labels.

Step 9: Analyze the error and adjust hyperparameters or optimizer settings to reduce the network's error.

Step 10: End.

**Code:**

```
import numpy as np
import tensorflow as tf
```

```
X = np.array([
    [2,1,0,0,0,0],
    [0,0,1,0,0,0],
    [0,0,0,1,1,1],
    [0,0,0,1,0,1],
    [1,0,1,0,0,0],
    [0,1,0,0,1,0],
    [0,0,0,0,1,0],
    [1,1,0,0,0,0],
```

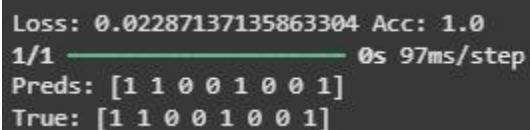
```
], dtype=float)
y = np.array([1,1,0,0,1,0,0,1])

# Simple Keras model
model = tf.keras.Sequential([
    tf.keras.layers.Input(shape=(6,)),
    tf.keras.layers.Dense(5, activation='relu'),
    tf.keras.layers.Dense(2, activation='softmax')
])

# Choose optimizer and compile
# Common choices: SGD(learning_rate=0.1), Adam(learning_rate=0.01)
optimizer = tf.keras.optimizers.Adam(learning_rate=0.01)
model.compile(optimizer=optimizer,
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

# Train
history = model.fit(X, y, epochs=200, batch_size=4, verbose=2, validation_split=0.25)

# Evaluate / predictions
loss, acc = model.evaluate(X, y, verbose=0)
print("Loss:", loss, "Acc:", acc)
preds = model.predict(X)
pred_classes = np.argmax(preds, axis=1)
print("Preds:", pred_classes)
print("True:", y)
```

**Output:**

```
Loss: 0.02287137135863304 Acc: 1.0
1/1 ————— 0s 97ms/step
Preds: [1 1 0 0 1 0 0 1]
True: [1 1 0 0 1 0 0 1]
```

**Result:**

Thus, the python program to train the neural networks in backpropagation was written, executed and verified.

<b>Expr No: 10</b>	<b>Isolation Forest Algorithm</b>
<b>Date:</b>	

**Aim:**

Write a python program to implement the Isolation Forest Algorithm.

**Algorithm:**

Step 1: Import the required libraries such as NumPy, Matplotlib, and IsolationForest from scikit-learn.

Step 2: Create the dataset consisting of mostly normal data points and a few anomaly points.

Step 3: Combine the normal and anomalous data into a single dataset for analysis.

Step 4: Initialize the Isolation Forest model and set the contamination parameter to indicate the expected proportion of anomalies.

Step 5: Fit the Isolation Forest model to the dataset to build the ensemble of isolation trees.

Step 6: Predict the class of each data point using the trained model where normal points are labeled as 1 and anomalies as -1.

Step 7: Visualize the results by plotting the data points with different colors for normal and anomalous observations.

Step 8: Analyze the output to identify the anomalies based on their isolation scores and shorter path lengths in the trees.

Step 9: End.

**Code:**

# Step 1: Import Libraries

```
import numpy as np
```

```
import matplotlib.pyplot as plt
```

```
from sklearn.ensemble import IsolationForest
```

# Step 2: Create Sample Data

# Normal data (clustered around 0)

```
X_normal = 0.3 * np.random.randn(100, 2)
```

# Add some anomalies

```
X_outliers = np.random.uniform(low=-4, high=4, size=(10, 2))
```

# Combine the data

```
X = np.vstack([X_normal, X_outliers])
```

# Step 3: Initialize Isolation Forest

```
iso_forest = IsolationForest(contamination=0.1, random_state=42)
```

```
# contamination = expected fraction of anomalies
```

```
# Step 4: Fit the model
```

```
iso_forest.fit(X)
```

```
# Step 5: Predict anomalies
```

```
# -1 -> anomaly, 1 -> normal
```

```
y_pred = iso_forest.predict(X)
```

```
# Step 6: Visualize
```

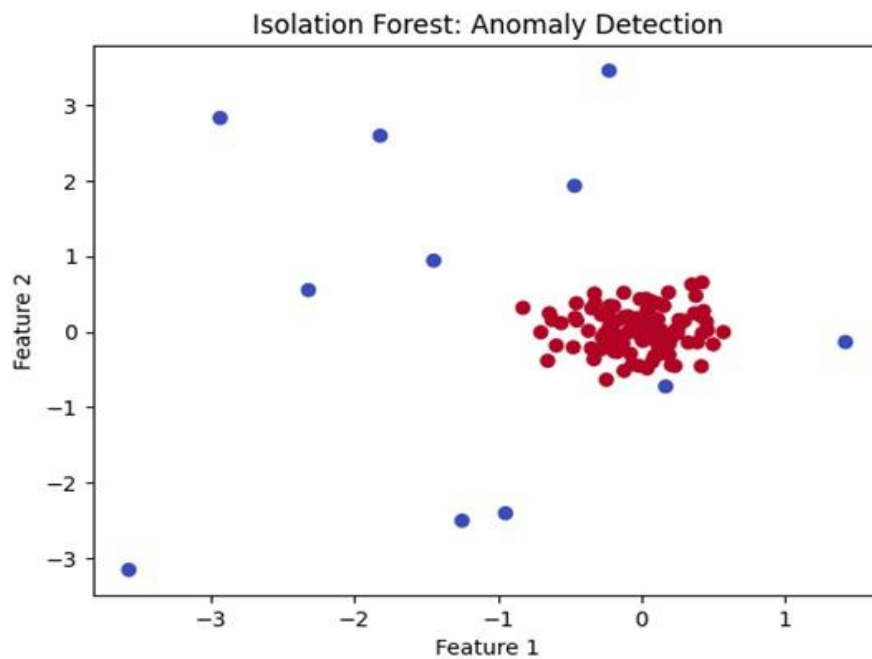
```
plt.scatter(X[:, 0], X[:, 1], c=y_pred, cmap='coolwarm')
```

```
plt.title("Isolation Forest: Anomaly Detection")
```

```
plt.xlabel("Feature 1")
```

```
plt.ylabel("Feature 2")
```

```
plt.show()
```

**Output:****Result:**

Thus, the python program to implement the Isolation Forest Algorithm was written, executed and verified.

<b>Expr No: 11</b>	<b>One Class SVM for Outlier Detection</b>
<b>Date:</b>	

**Aim:**

Write a python program to implement the One Class SVM for Outlier Detection.

**Algorithm:**

Step 1: Import the required libraries such as NumPy, Matplotlib, and One Class SVM from scikit-learn.

Step 2: Create the dataset by generating mostly normal data points and a few anomalous data points.

Step 3: Combine the normal and anomalous data into a single dataset for analysis.

Step 4: Initialize the One-Class SVM model with a suitable kernel function (commonly 'rbf') and set parameters such as gamma and nu.

Step 5: Train the model using only the normal data to learn the boundary that encloses most of the data points.

Step 6: Predict the labels for all data points using the trained model, where normal points are labeled as +1 and anomalies as -1.

Step 7: Visualize the results by plotting the normal and anomalous data points in different colors.

Step 8: Analyze how the kernel function influences the decision boundary — for instance, using an RBF kernel helps detect complex, non-linear data distributions effectively.

Step 9: End.

**Code:**

# Step 1: Import Libraries

```
import numpy as np
```

```
import matplotlib.pyplot as plt
```

```
from sklearn.svm import OneClassSVM
```

# Step 2: Create Sample Data

```
X_normal = 0.3 * np.random.randn(100, 2) # Normal points
```

```
X_outliers = np.random.uniform(low=-4, high=4, size=(10, 2)) # Anomalies
```

```
X = np.vstack([X_normal, X_outliers])
```

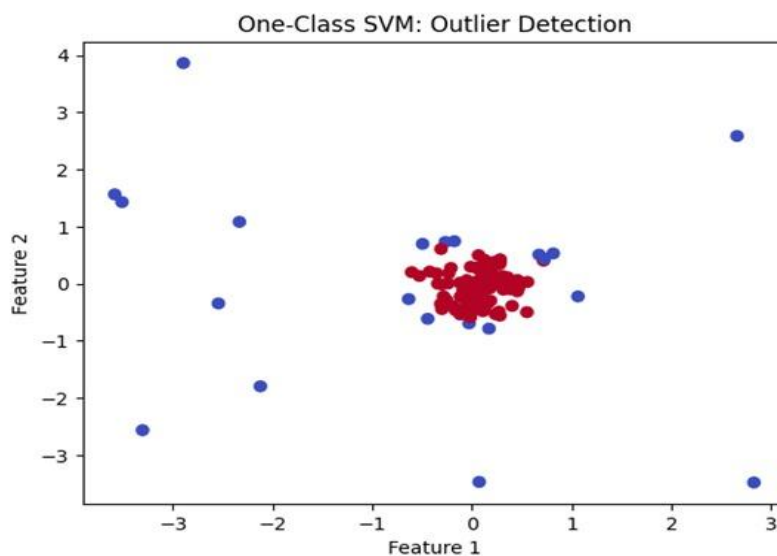
# Step 3: Initialize One-Class SVM

```
# kernel='rbf' captures nonlinear patterns
oc_svm = OneClassSVM(kernel='rbf', gamma=0.5, nu=0.1)
# nu = expected fraction of outliers
```

```
# Step 4: Fit the model
oc_svm.fit(X_normal) # Train only on normal data
```

```
# Step 5: Predict anomalies
# +1 -> normal, -1 -> anomaly
y_pred = oc_svm.predict(X)
```

```
# Step 6: Visualize
plt.scatter(X[:, 0], X[:, 1], c=y_pred, cmap='coolwarm')
plt.title("One-Class SVM: Outlier Detection")
plt.xlabel("Feature 1")
plt.ylabel("Feature 2")
plt.show()
```

**Output:****Result:**

Thus, the python program to implement the One Class SVM for Outlier Detection was written, executed and verified