

INDEX

S.NO.	EXPERIMENT	DATE	PAGE NO.	SIGNATURE
1.	Setting up the Jupyter Notebook and Executing a Python Program.			
2.	Installing Keras, Tensorflow and Pytorch, Pandas, numpy etc libraries and making use of them.			
3.	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.			
4.	Program to demonstrate k-means clustering algorithm.			
5.	Program to demonstrate DBSCAN clustering algorithm.			
6.	Program to demonstrate PCA and LDA on Iris dataset.			
7.	Compare the performance of PCA and Autoencoders on a given dataset.			
8.	Build Generative adversarial model for fake (news/image/audio/video) prediction.			
9.	Outlier detection in time series dataset using RNN.			
10.	Anomaly detection using Self-Organizing Network.			

EXPERIMENT – 01

AIM : Setting up the Jupyter Notebook and Executing a Python Program.

THEORY :

Jupyter Notebook is an open-source web application that allows you to create and share documents that contain live code, equations, visualizations, and narrative text. It is widely used for data analysis, scientific research, machine learning, and more.

Some key features of Jupyter Notebook include:

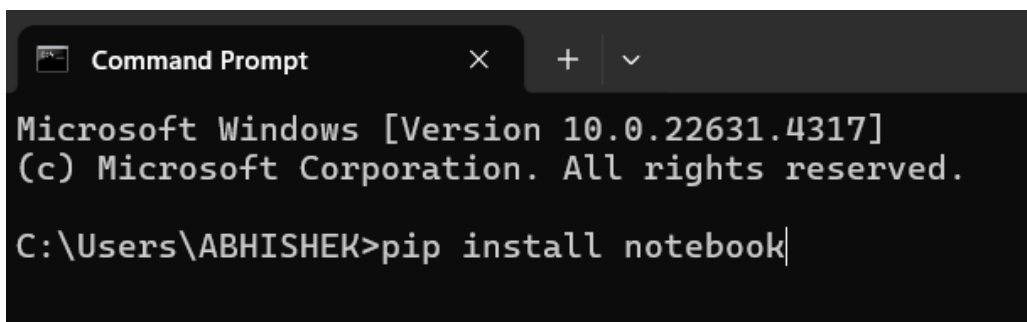
- **Interactive coding environment:** You can run code in chunks or cells.
- **Rich text support:** You can include explanations and notes using Markdown.
- **Visualization support:** It allows inline plotting and rendering of visualizations using libraries like Matplotlib, Seaborn, and Plotly.
- **Sharing and collaboration:** You can share notebooks and collaborate with others.

Setting Up Jupyter Notebook

To use Jupyter Notebooks, you need to install the necessary software on your computer.

Steps for Installation:

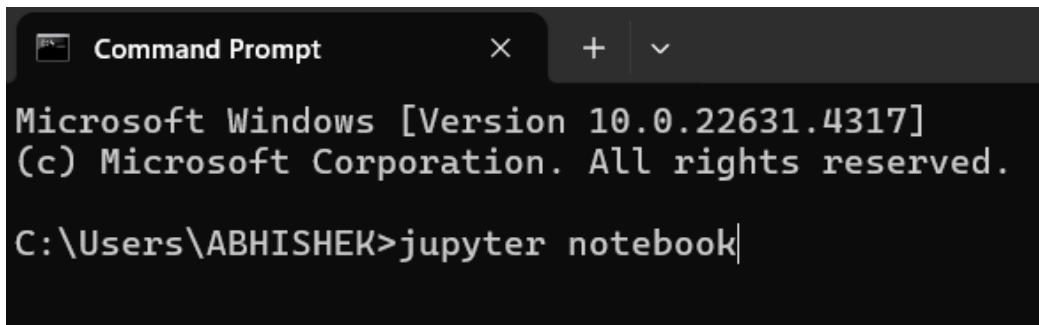
1. **Install Anaconda:** Anaconda is a distribution of Python and R for scientific computing and data science. It comes pre-packaged with Jupyter and many useful libraries like NumPy, Pandas, Matplotlib, etc.
2. **Install Jupyter Notebook via pip (if you don't use Anaconda):** If you don't use Anaconda, you can install Jupyter directly using pip:



```
Command Prompt
Microsoft Windows [Version 10.0.22631.4317]
(c) Microsoft Corporation. All rights reserved.
C:\Users\ABHISHEK>pip install notebook
```

3. **Launch Jupyter Notebook:**

After installation, open a terminal and type:



```
Microsoft Windows [Version 10.0.22631.4317]
(c) Microsoft Corporation. All rights reserved.

C:\Users\ABHISHEK>jupyter notebook|
```

CODE :

```
# Addition
a = 10
b = 5
addition = a + b
print(f'Addition of {a} and {b} is: {addition}')

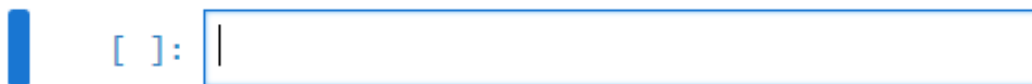
# Subtraction
subtraction = a - b
print(f'Subtraction of {a} and {b} is: {subtraction}')

# Multiplication
multiplication = a * b
print(f'Multiplication of {a} and {b} is: {multiplication}')

# Division
division = a / b
print(f'Division of {a} by {b} is: {division}')
```

OUTPUT :

```
Addition of 10 and 5 is: 15
Subtraction of 10 and 5 is: 5
Multiplication of 10 and 5 is: 50
Division of 10 by 5 is: 2.0
```



EXPERIMENT – 02

AIM : Installing Keras, Tensorflow and Pytorch, Pandas, numpy etc libraries and making use of them.

THEORY :

- **Keras**: High-level neural networks API, written in Python and capable of running on top of TensorFlow. Keras is user-friendly and simplifies the process of building and training deep learning models.
- **TensorFlow**: An open-source framework developed by Google for building and training machine learning and deep learning models. TensorFlow supports a variety of tasks, including classification, regression, clustering, and more.
- **PyTorch**: Another popular deep learning framework developed by Facebook's AI Research lab. It provides flexibility and ease of use for building neural networks, with a dynamic computation graph (eager execution).
- **Pandas**: A powerful data manipulation and analysis library that makes working with structured data (like CSV, Excel, SQL databases) efficient. It provides DataFrames to work with tabular data.
- **NumPy**: A core library for scientific computing in Python. It provides support for multi-dimensional arrays and matrices, along with a large collection of high-level mathematical functions to operate on these arrays.

Steps for Installing Libraries:

1. **Install Anaconda (Optional, but Recommended)**: Anaconda is a distribution of Python that comes pre-installed with many useful data science libraries, including Pandas, NumPy, TensorFlow, Keras, and PyTorch.
2. **Install Libraries via pip (If you don't use Anaconda)**: If you're using a standard Python installation, you can install the required libraries using pip, the Python package manager. Open a terminal or command prompt and execute the following commands:

`pip install tensorflow pytorch pandas numpy`

CODE :

```
# Import libraries
import pandas as pd
import numpy as np
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Input
```

```

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

# Load and preprocess data using Pandas
data = {'Feature1': [0.1, 0.2, 0.3, 0.4, 0.5],
        'Feature2': [1.1, 1.2, 1.3, 1.4, 1.5],
        'Label': [0, 1, 0, 1, 0]} # Binary classification labels

df = pd.DataFrame(data)

# Convert DataFrame to NumPy arrays
X = df[['Feature1', 'Feature2']].values
y = df['Label'].values

# Standardize the features (mean = 0, std = 1) using NumPy
X = (X - np.mean(X, axis=0)) / np.std(X, axis=0)

# Build and train the model using Keras (TensorFlow)
# Keras model definition
keras_model = Sequential([
    Input(shape=(2,)), # Define input shape (2 features)
    Dense(8, activation='relu'),
    Dense(4, activation='relu'),
    Dense(1, activation='sigmoid') # Output layer for binary classification
])

# Compile the Keras model
keras_model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])

# Train the Keras model
keras_model.fit(X, y, epochs=10, batch_size=2, verbose=1)

# Build and train the model using PyTorch
# PyTorch model definition
class SimpleModel(nn.Module):
    def __init__(self):
        super(SimpleModel, self).__init__()
        self.fc1 = nn.Linear(2, 8)
        self.fc2 = nn.Linear(8, 4)
        self.fc3 = nn.Linear(4, 1)

    def forward(self, x):
        x = torch.relu(self.fc1(x))
        x = torch.relu(self.fc2(x))
        x = torch.sigmoid(self.fc3(x))

```

```

    return x

# Initialize the PyTorch model, loss function, and optimizer
pytorch_model = SimpleModel()
criterion = nn.BCELoss() # Binary Cross-Entropy Loss
optimizer = optim.Adam(pytorch_model.parameters(), lr=0.001)

# Convert the NumPy data to PyTorch tensors
X_tensor = torch.tensor(X, dtype=torch.float32)
y_tensor = torch.tensor(y, dtype=torch.float32).view(-1, 1)

# Train the PyTorch model
num_epochs = 10
for epoch in range(num_epochs):
    optimizer.zero_grad()
    outputs = pytorch_model(X_tensor)
    loss = criterion(outputs, y_tensor)
    loss.backward()
    optimizer.step()

    if (epoch+1) % 2 == 0:
        print(f'Epoch [{epoch+1}/{num_epochs}], Loss: {loss.item():.4f}')

# Evaluate and compare both models
# Keras evaluation
keras_loss, keras_acc = keras_model.evaluate(X, y, verbose=0)
print(f'\nKeras Model Accuracy: {keras_acc * 100:.2f}%')

# PyTorch evaluation
with torch.no_grad():
    pytorch_predictions = pytorch_model(X_tensor)
    pytorch_predictions = (pytorch_predictions > 0.5).float()

# Print PyTorch model predictions
print(f'PyTorch Model Predictions: {pytorch_predictions.view(-1).numpy()}')

```

OUTPUT :

```
Epoch 1/10
3/3 ————— 1s 6ms/step - accuracy: 0.5500 - loss: 0.8463
Epoch 2/10
3/3 ————— 0s 4ms/step - accuracy: 0.5500 - loss: 0.7318
Epoch 3/10
3/3 ————— 0s 3ms/step - accuracy: 0.5500 - loss: 0.8459
Epoch 4/10
3/3 ————— 0s 2ms/step - accuracy: 0.5500 - loss: 0.7656
Epoch 5/10
3/3 ————— 0s 3ms/step - accuracy: 0.4250 - loss: 0.8236.97
Epoch 6/10
3/3 ————— 0s 5ms/step - accuracy: 0.5500 - loss: 0.7488
Epoch 7/10
3/3 ————— 0s 3ms/step - accuracy: 0.5500 - loss: 0.8255
Epoch 8/10
3/3 ————— 0s 3ms/step - accuracy: 0.6600 - loss: 0.7542
Epoch 9/10
3/3 ————— 0s 2ms/step - accuracy: 0.6750 - loss: 0.71990
Epoch 10/10
3/3 ————— 0s 3ms/step - accuracy: 0.5500 - loss: 0.7807
Epoch [2/10], Loss: 0.6743
Epoch [4/10], Loss: 0.6739
Epoch [6/10], Loss: 0.6736
Epoch [8/10], Loss: 0.6733
Epoch [10/10], Loss: 0.6730
```

Keras Model Accuracy: 60.00%

PyTorch Model Predictions: [0. 0. 0. 0. 0.]

EXPERIMENT – 03

AIM : 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.

THEORY :

EM (Expectation-Maximization) Algorithm:

The EM algorithm is a powerful method for finding maximum likelihood estimates of parameters in probabilistic models, particularly when the data has missing values or latent variables. It consists of two steps:

- **E-Step (Expectation Step):** Compute the expected value of the latent variables given the current parameters.
- **M-Step (Maximization Step):** Maximize the expected log-likelihood found in the E-step with respect to the parameters.

The EM algorithm can be applied to clustering tasks, specifically in Gaussian Mixture Models (GMM), where the goal is to find a mixture of multiple Gaussian distributions that best fit the data.

k-Means Clustering:

k-Means is a centroid-based clustering algorithm where the number of clusters (k) is predefined. The algorithm works by:

- Randomly initializing k centroids.
- Assigning each data point to the nearest centroid.
- Recalculating the centroids based on the mean of the points in each cluster.
- Repeating the above steps until convergence.

CODE :

```
# Import necessary libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans
from sklearn.mixture import GaussianMixture
from sklearn.metrics import silhouette_score

# Load the dataset from CSV file
data = pd.read_csv('data.csv')
```



```

X = data[['x', 'y']].values

# k-Means Clustering
kmeans = KMeans(n_clusters=3, random_state=42)
kmeans_labels = kmeans.fit_predict(X)

# Gaussian Mixture Model (EM Algorithm) Clustering
gmm = GaussianMixture(n_components=3, random_state=42)
gmm_labels = gmm.fit_predict(X)

# Visualize Clusters
plt.figure(figsize=(12, 6))

# Plot k-Means Clustering
plt.subplot(1, 2, 1)
plt.scatter(X[:, 0], X[:, 1], c=kmeans_labels, cmap='viridis', label='k-Means Clusters')
plt.scatter(kmeans.cluster_centers_[:, 0], kmeans.cluster_centers_[:, 1], s=200, c='red',
            marker='X', label='Centroids')
plt.title('k-Means Clustering')
plt.xlabel('X')
plt.ylabel('Y')
plt.legend()

# Plot Gaussian Mixture Model (EM) Clustering
plt.subplot(1, 2, 2)
plt.scatter(X[:, 0], X[:, 1], c=gmm_labels, cmap='viridis', label='GMM Clusters')
plt.title('Gaussian Mixture Model (EM) Clustering')
plt.xlabel('X')
plt.ylabel('Y')
plt.legend()

plt.tight_layout()
plt.show()

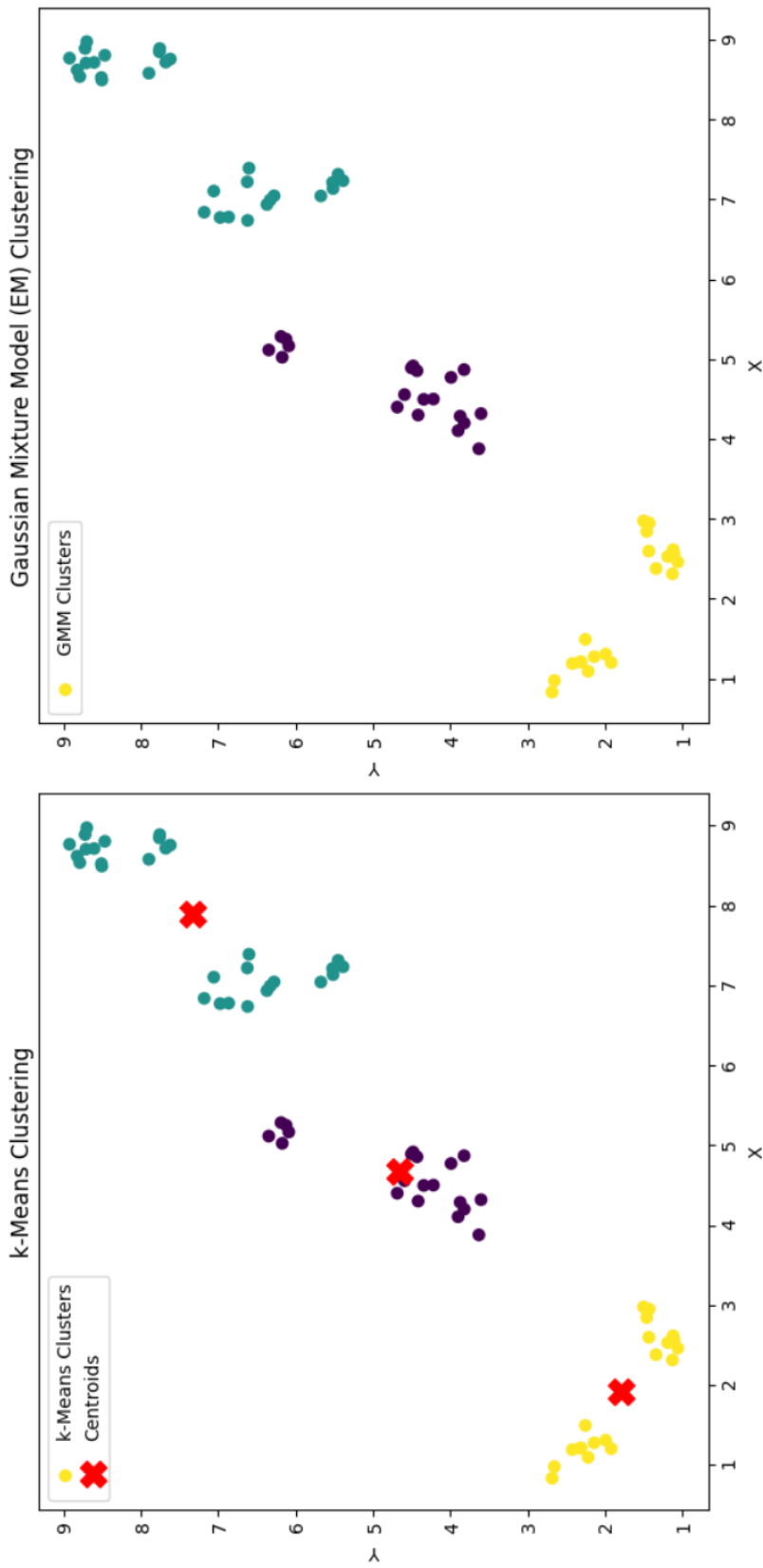
# Comparing the results
# Silhouette Score to evaluate the clustering quality
kmeans_score = silhouette_score(X, kmeans_labels)
gmm_score = silhouette_score(X, gmm_labels)

print(f"Silhouette Score for k-Means: {kmeans_score}")
print(f"Silhouette Score for GMM: {gmm_score}")

# Comments on the quality of clustering
if kmeans_score > gmm_score:
    print("k-Means clustering is better in terms of cluster quality.")
else:
    print("Gaussian Mixture Model (EM) clustering is better in terms of cluster quality.")

```

OUTPUT :



Silhouette Score for k-Means: 0.6252493370841674
Silhouette Score for GMM: 0.6252493370841674
Gaussian Mixture Model (EM) clustering is better in terms of cluster quality.

EXPERIMENT – 04

AIM : Program to demonstrate k-means clustering algorithm.

THEORY :

K-Means Clustering is a popular unsupervised machine learning algorithm used to partition a dataset into **K distinct clusters**. It works by minimizing the variance within each cluster, ensuring that similar data points are grouped together. The algorithm is based on the concept of centroids, which represent the center of each cluster.

Advantages:

- Simple and efficient for small to medium-sized datasets.
- Can work with continuous data.
- Works well when clusters are spherical and well-separated.

Disadvantages:

- Requires the number of clusters (K) to be specified in advance.
- Sensitive to initial placement of centroids.
- Not suitable for clusters with non-convex shapes.

CODE :

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans
from sklearn.datasets import make_blobs

# Creating sample dataset
X, y = make_blobs(n_samples=300, centers=4, random_state=42)

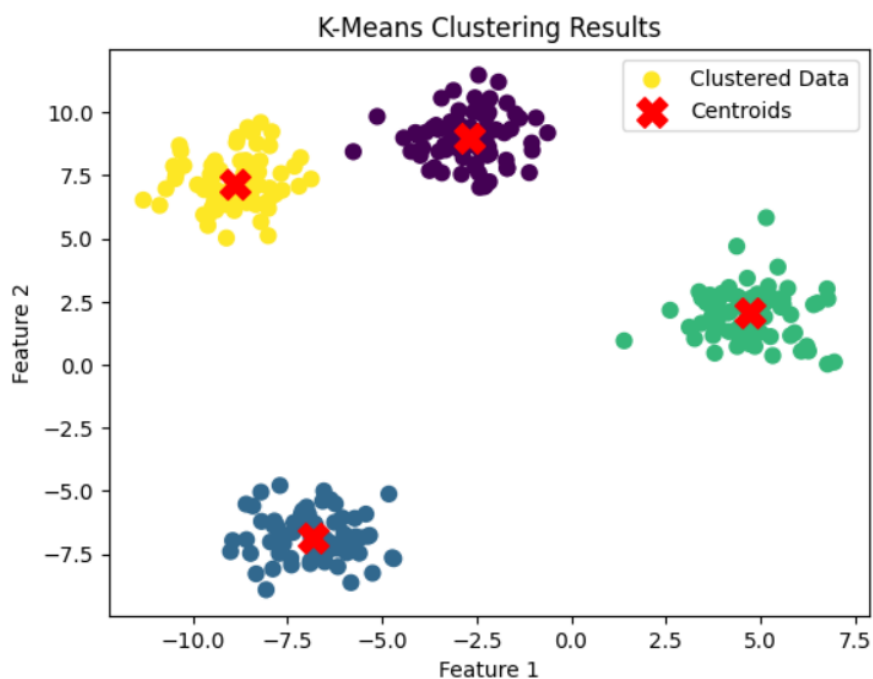
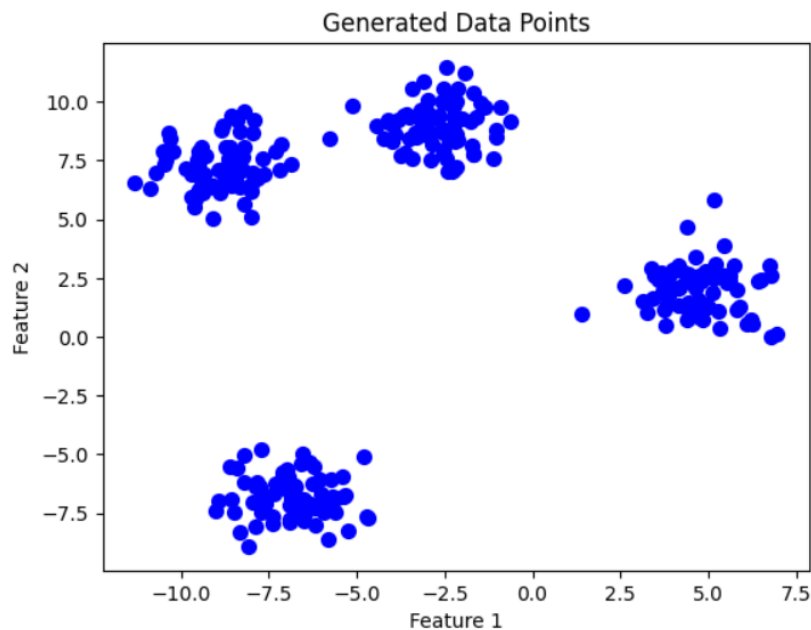
# Visualize the dataset
plt.scatter(X[:, 0], X[:, 1], s=50, c='blue', label="Data Points")
plt.title("Generated Data Points")
plt.xlabel("Feature 1")
plt.ylabel("Feature 2")
plt.show()

# Apply K-Means clustering
kmeans = KMeans(n_clusters=4, random_state=42)
kmeans.fit(X)

# Extract the centroids and labels
centroids = kmeans.cluster_centers_
labels = kmeans.labels_
```

```
# Visualize the clustering results
plt.scatter(X[:, 0], X[:, 1], c=labels, s=50, cmap='viridis', label="Clustered Data")
plt.scatter(centroids[:, 0], centroids[:, 1], marker='X', s=200, c='red', label="Centroids")
plt.title("K-Means Clustering Results")
plt.xlabel("Feature 1")
plt.ylabel("Feature 2")
plt.legend()
plt.show()
```

OUTPUT :



EXPERIMENT – 05

AIM : Program to demonstrate DBSCAN clustering algorithm.

THEORY :

DBSCAN is a density-based clustering algorithm that groups together points that are close to each other based on a distance metric and a minimum number of points. Unlike K-Means, DBSCAN does not require the user to specify the number of clusters in advance and can find arbitrarily shaped clusters. DBSCAN also marks outliers (points that don't belong to any cluster) as noise.

CODE :

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.cluster import DBSCAN
from sklearn.datasets import make_blobs

# Step 1: Creating synthetic dataset
X, y = make_blobs(n_samples=300, centers=3, cluster_std=0.60, random_state=42)

# Adding some noise to the dataset
X = np.vstack([X, np.random.rand(20, 2) * 5])

# Visualize the original dataset
plt.scatter(X[:, 0], X[:, 1], s=50, c='blue', label="Data Points")
plt.title("Generated Data Points with Noise")
plt.xlabel("Feature 1")
plt.ylabel("Feature 2")
plt.show()

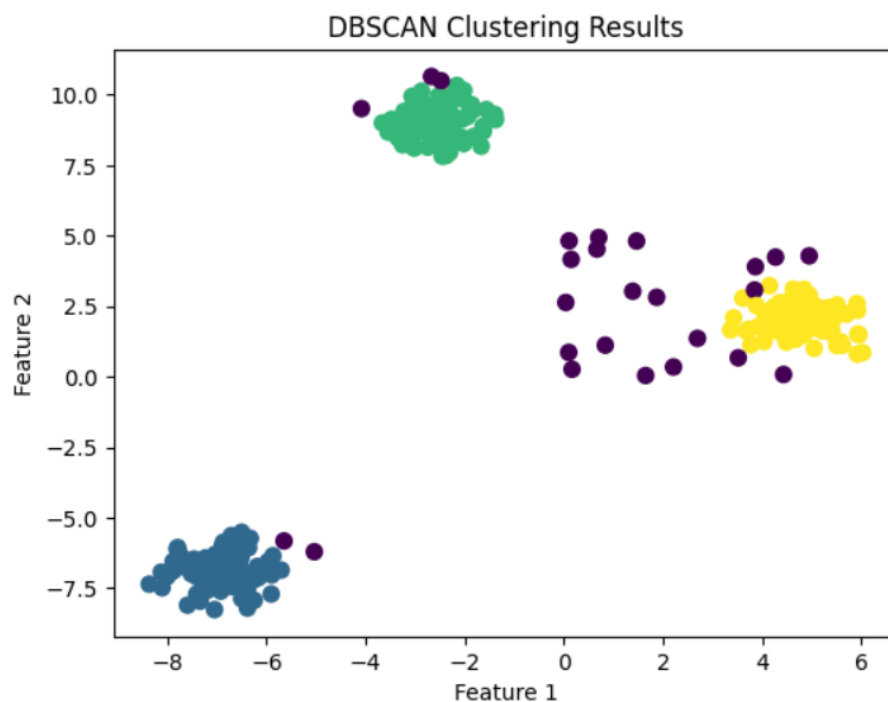
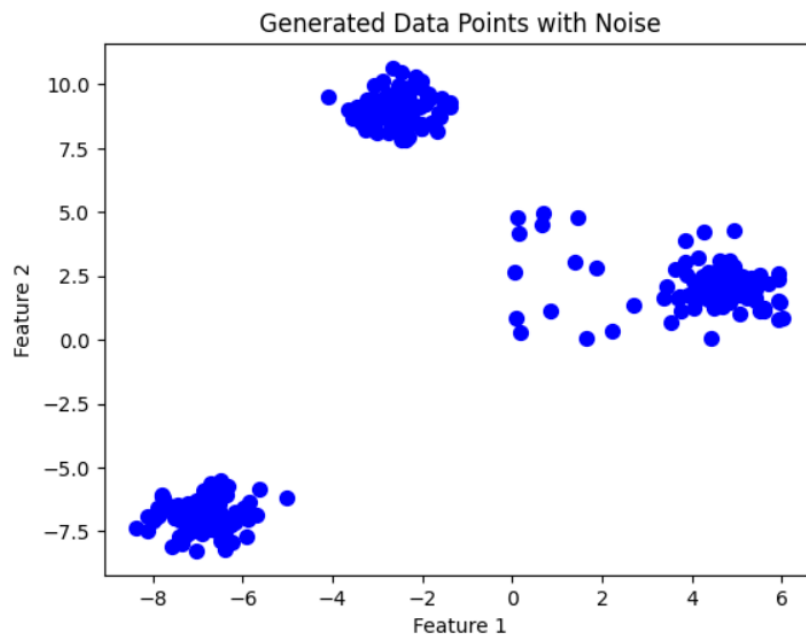
# Apply DBSCAN clustering
dbscan = DBSCAN(eps=0.5, min_samples=5)
dbscan.fit(X)

# Extract the labels (cluster assignments)
labels = dbscan.labels_

# Visualize the clustering results
# Points labeled as -1 are considered noise (outliers)
plt.scatter(X[:, 0], X[:, 1], c=labels, cmap='viridis', s=50)
```

```
plt.title("DBSCAN Clustering Results")
plt.xlabel("Feature 1")
plt.ylabel("Feature 2")
plt.show()
```

OUTPUT :



EXPERIMENT – 06

AIM : Program to demonstrate PCA and LDA on Iris dataset.

THEORY :

Principal Component Analysis (PCA):

PCA is a technique used for dimensionality reduction, often employed when you have high-dimensional data. It transforms the data into a new coordinate system where the greatest variance comes to lie on the first axis, the second greatest on the second axis, and so on. PCA tries to preserve as much variance as possible when reducing the dimensions.

Linear Discriminant Analysis (LDA):

LDA is another dimensionality reduction technique, but it is supervised, meaning it uses the labels of the data to maximize the separability between different classes. LDA works by finding the axes that maximize the separation between multiple classes.

CODE :

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn import datasets
from sklearn.decomposition import PCA
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis as LDA
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler

# Load the Iris dataset
iris = datasets.load_iris()
X = iris.data
y = iris.target

# Split the data into training and testing sets (for LDA)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

# Standardize the data
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

# Principal Component Analysis
pca = PCA(n_components=2)
X_pca = pca.fit_transform(X_train_scaled)
```

```

# Visualize the PCA result
plt.figure(figsize=(8,6))
plt.scatter(X_pca[:, 0], X_pca[:, 1], c=y_train, cmap='viridis', edgecolor='k', s=100)
plt.colorbar()
plt.title('PCA - Iris Dataset')
plt.xlabel('Principal Component 1')
plt.ylabel('Principal Component 2')
plt.show()

# Linear Discriminant Analysis
lda = LDA(n_components=2)
X_lda = lda.fit_transform(X_train_scaled, y_train)

# Visualize the LDA result
plt.figure(figsize=(8,6))
plt.scatter(X_lda[:, 0], X_lda[:, 1], c=y_train, cmap='viridis', edgecolor='k', s=100)
plt.colorbar()
plt.title('LDA - Iris Dataset')
plt.xlabel('LD 1')
plt.ylabel('LD 2')
plt.show()

# PCA on the whole dataset (for comparison)
X_pca_full = pca.fit_transform scaler.transform(X))

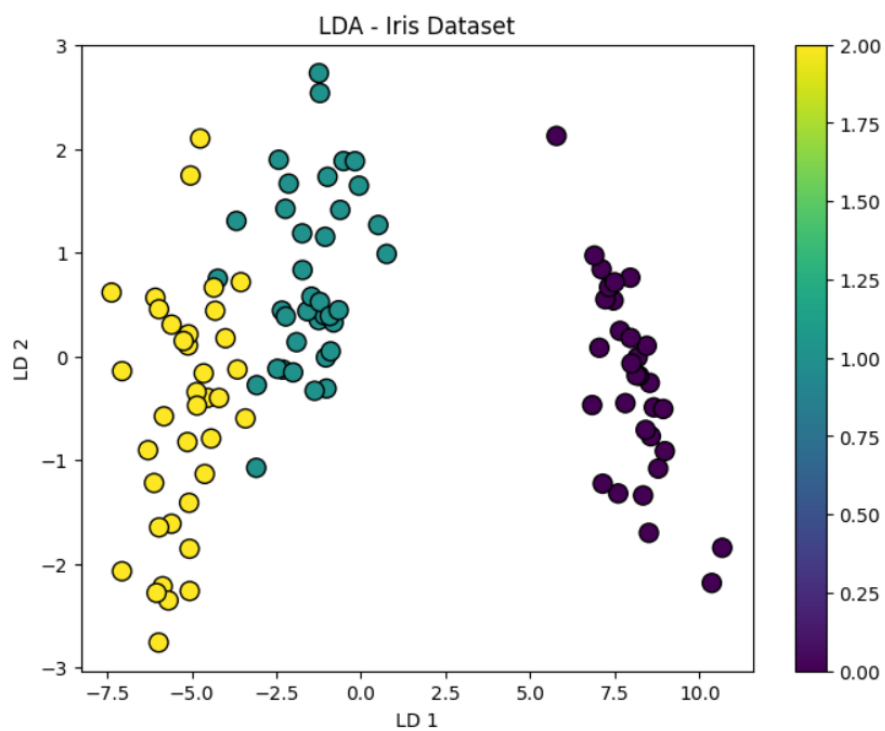
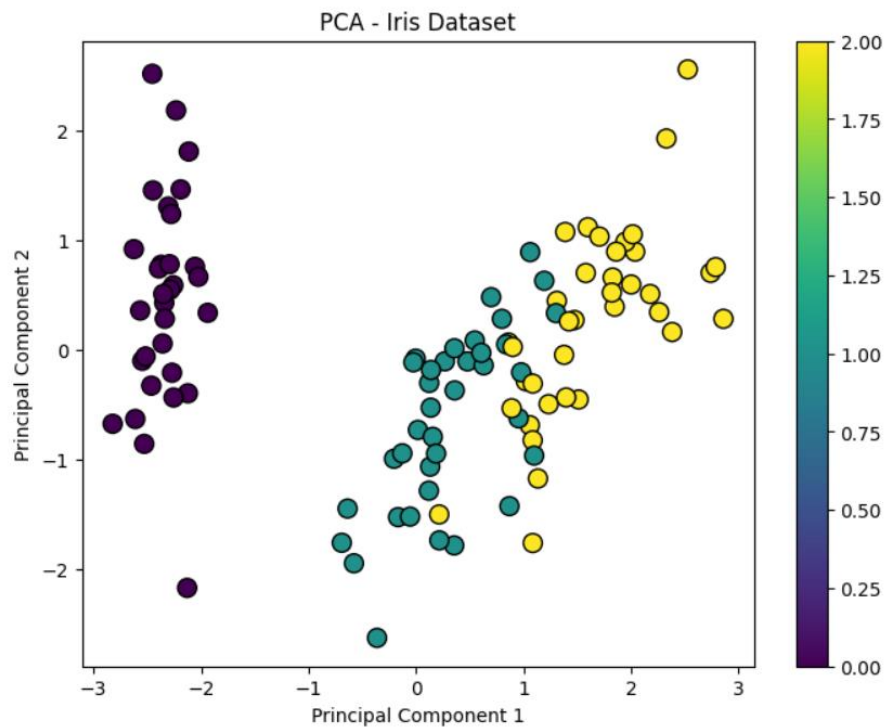
# Visualize the full PCA result
plt.figure(figsize=(8,6))
plt.scatter(X_pca_full[:, 0], X_pca_full[:, 1], c=y, cmap='viridis', edgecolor='k', s=100)
plt.colorbar()
plt.title('PCA - Whole Iris Dataset')
plt.xlabel('Principal Component 1')
plt.ylabel('Principal Component 2')
plt.show()

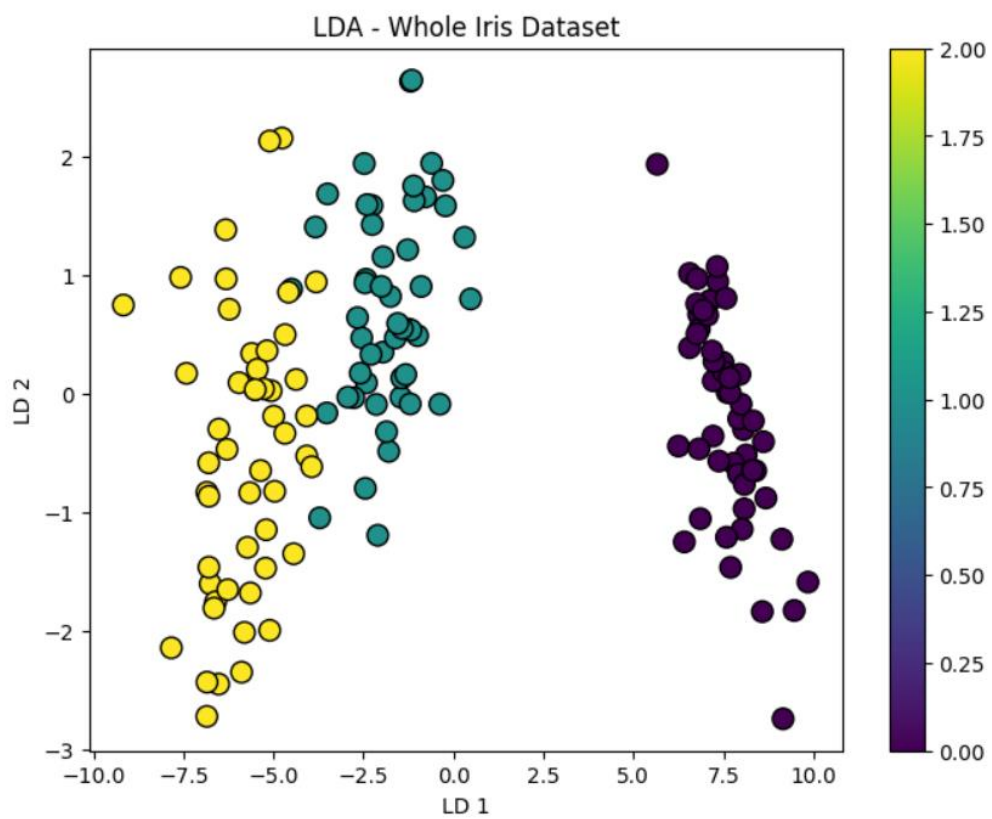
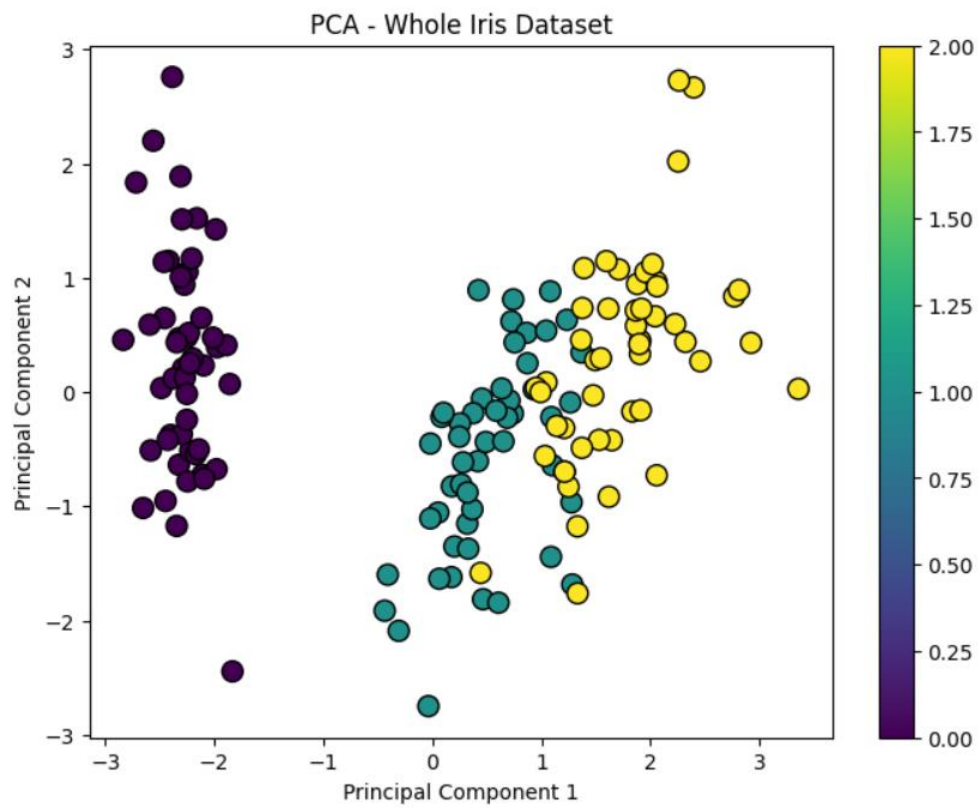
# LDA on the whole dataset (for comparison)
lda_full = LDA(n_components=2)
X_lda_full = lda_full.fit_transform scaler.transform(X), y)

# Visualize the full LDA result
plt.figure(figsize=(8,6))
plt.scatter(X_lda_full[:, 0], X_lda_full[:, 1], c=y, cmap='viridis', edgecolor='k', s=100)
plt.colorbar()
plt.title('LDA - Whole Iris Dataset')
plt.xlabel('LD 1')
plt.ylabel('LD 2')
plt.show()

```


OUTPUT :





EXPERIMENT – 07

AIM : Compare the performance of PCA and Autoencoders on a given dataset.

THEORY :

Principal Component Analysis (PCA):

PCA is a linear technique for dimensionality reduction. It transforms the data into a new coordinate system where the first few components (principal components) capture the maximum variance in the data. PCA does not require any training and is computationally less expensive compared to deep learning methods.

- **Advantages:**
 - Simple and interpretable.
 - Effective for linearly separable data.
 - Computationally less intensive.
- **Disadvantages:**
 - Only works for linear relationships.
 - May not perform well when the data has complex patterns or non-linearity.

Autoencoders:

Autoencoders are a type of neural network used for unsupervised learning of efficient codings. They consist of two parts:

1. Encoder: Maps the input data into a lower-dimensional space.
2. Decoder: Reconstructs the data back from the lower-dimensional representation.

The idea is that the network learns a compressed representation of the data while minimizing the reconstruction error.

- **Advantages:**
 - Can model complex, non-linear relationships in data.
 - More flexible than PCA as they can capture more intricate patterns.
- **Disadvantages:**
 - Requires more computational power.
 - Sensitive to hyperparameter tuning and requires more data for training.
 - Training may be more time-consuming

CODE :

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn import datasets
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import mean_squared_error
from tensorflow.keras.models import Sequential
```

```

from tensorflow.keras.layers import Input, Dense
import tensorflow as tf

# Load the Iris dataset
iris = datasets.load_iris()
X = iris.data
y = iris.target

scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Split the data into train and test sets
X_train, X_test, _, _ = train_test_split(X_scaled, y, test_size=0.2, random_state=42)

# PCA Implementation

# Apply PCA to reduce the dataset to 2 dimensions
from sklearn.decomposition import PCA

pca = PCA(n_components=2)
X_train_pca = pca.fit_transform(X_train)
X_test_pca = pca.transform(X_test)

# Visualize the PCA result
plt.figure(figsize=(8, 6))
plt.scatter(X_train_pca[:, 0], X_train_pca[:, 1], c=y[:len(X_train)], cmap='viridis',
            edgecolor='k', s=100)
plt.title('PCA - Iris Dataset')
plt.xlabel('Principal Component 1')
plt.ylabel('Principal Component 2')
plt.colorbar()
plt.show()

# Autoencoder Implementation
autoencoder = Sequential()
autoencoder.add(Input(shape=(X_train.shape[1],)))
autoencoder.add(Dense(64, activation='relu'))
autoencoder.add(Dense(2, activation='relu'))
autoencoder.add(Dense(64, activation='relu'))
autoencoder.add(Dense(X_train.shape[1], activation='sigmoid'))

autoencoder.compile(optimizer='adam', loss='mse')

# Train the autoencoder
autoencoder.fit(X_train, X_train, epochs=100, batch_size=16, validation_data=(X_test,
X_test), verbose=0)

```

```

encoder = Sequential(autoencoder.layers[:2])
X_train_ae = encoder.predict(X_train)
X_test_ae = encoder.predict(X_test)
# Visualize
plt.figure(figsize=(8, 6))
plt.scatter(X_train_ae[:, 0], X_train_ae[:, 1], c=y[:len(X_train)], cmap='viridis', edgecolor='k',
s=100)
plt.title('Autoencoder - Iris Dataset')
plt.xlabel('Latent Space 1')
plt.ylabel('Latent Space 2')
plt.colorbar()
plt.show()

# Comparison of PCA and Autoencoders

# Reconstruct the data from PCA and Autoencoder
X_train_pca_reconstructed = pca.inverse_transform(X_train_pca)
X_test_pca_reconstructed = pca.inverse_transform(X_test_pca)

X_train_ae_reconstructed = autoencoder.predict(X_train)
X_test_ae_reconstructed = autoencoder.predict(X_test)

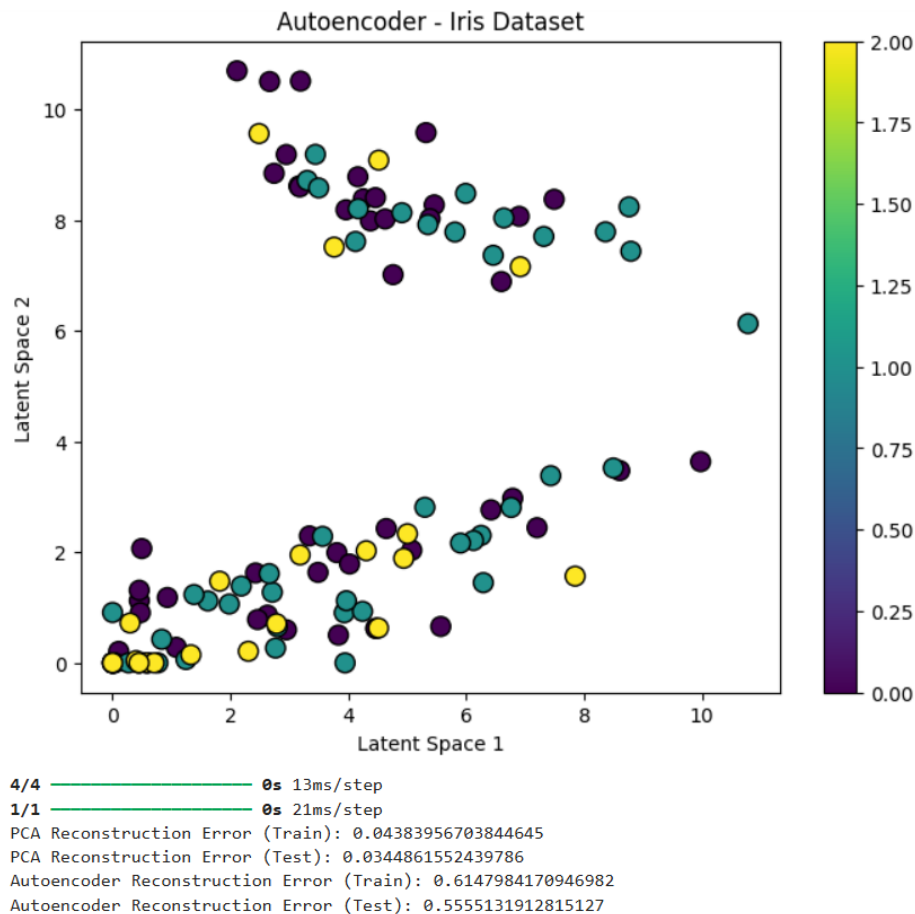
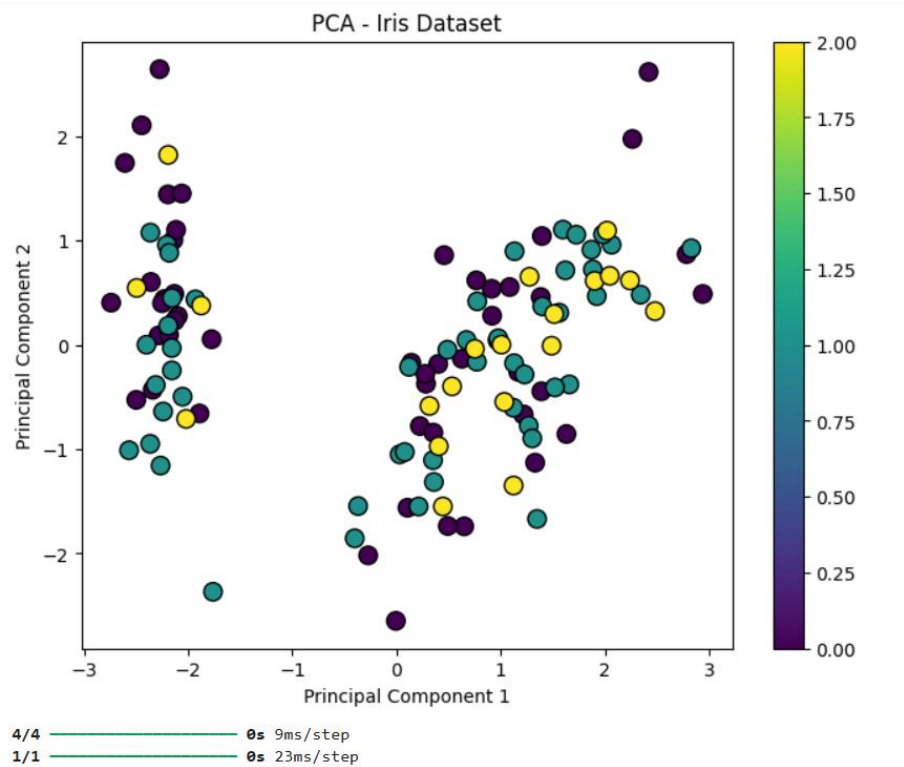
# Calculate the reconstruction errors
pca_train_error = mean_squared_error(X_train, X_train_pca_reconstructed)
pca_test_error = mean_squared_error(X_test, X_test_pca_reconstructed)

ae_train_error = mean_squared_error(X_train, X_train_ae_reconstructed)
ae_test_error = mean_squared_error(X_test, X_test_ae_reconstructed)

print("PCA Reconstruction Error (Train):", pca_train_error)
print("PCA Reconstruction Error (Test):", pca_test_error)
print("Autoencoder Reconstruction Error (Train):", ae_train_error)
print("Autoencoder Reconstruction Error (Test):", ae_test_error)

```

OUTPUT :



EXPERIMENT – 08

AIM : Build Generative adversarial model for fake (news/image/audio/video) prediction.

THEORY :

Introduction to Generative Adversarial Networks (GANs):

A **Generative Adversarial Network (GAN)** is a type of deep learning architecture consisting of two neural networks, a **generator** and a **discriminator**, that are trained simultaneously through adversarial training.

- **Generator:** This network generates fake data, starting with random noise and learning to create data samples (like images, text, etc.) that look real.
- **Discriminator:** This network evaluates data samples and determines whether they are real (from the training dataset) or fake (produced by the generator).

The two networks compete in a game-theoretic scenario:

- The **generator** aims to fool the discriminator into classifying fake data as real.
- The **discriminator** aims to correctly distinguish between real and fake data.

As training progresses, both networks improve: the generator gets better at creating realistic data, and the discriminator gets better at distinguishing real from fake. Eventually, the generator produces highly realistic data that is difficult for the discriminator to differentiate from real data.

CODE :

```
import torch
import torch.nn as nn
import torch.optim as optim
import torchvision
from torchvision import datasets, transforms
import matplotlib.pyplot as plt
import numpy as np

# Set device (Use CUDA if available, else CPU)
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

# basic transform for image normalization
transform = transforms.Compose([
    transforms.ToTensor(),
```

```

        transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
    ])

# Load the CIFAR-10 dataset (for fake image generation)
train_dataset = datasets.CIFAR10(root='./data', train=True, download=True,
transform=transform)
dataloader = torch.utils.data.DataLoader(train_dataset, batch_size=32, shuffle=True)

# Hyperparameters
latent_dim = 100 # Dimension of the latent space
lr = 0.0002 # Learning rate
beta1 = 0.5 # Adam optimizer beta1
beta2 = 0.999 # Adam optimizer beta2
num_epochs = 10 # Number of training epochs

# Generator model
class Generator(nn.Module):
    def __init__(self, latent_dim):
        super(Generator, self).__init__()
        self.model = nn.Sequential(
            nn.Linear(latent_dim, 128 * 8 * 8),
            nn.ReLU(),
            nn.Unflatten(1, (128, 8, 8)),
            nn.Upsample(scale_factor=2),
            nn.Conv2d(128, 128, kernel_size=3, padding=1),
            nn.BatchNorm2d(128, momentum=0.78),
            nn.ReLU(),
            nn.Upsample(scale_factor=2),
            nn.Conv2d(128, 64, kernel_size=3, padding=1),
            nn.BatchNorm2d(64, momentum=0.78),
            nn.ReLU(),
            nn.Conv2d(64, 3, kernel_size=3, padding=1),
            nn.Tanh()
        )

    def forward(self, z):
        return self.model(z)

# Discriminator model
class Discriminator(nn.Module):
    def __init__(self):
        super(Discriminator, self).__init__()
        self.model = nn.Sequential(
            nn.Conv2d(3, 32, kernel_size=3, stride=2, padding=1),
            nn.LeakyReLU(0.2),
            nn.Dropout(0.25),

```



```

        nn.Conv2d(32, 64, kernel_size=3, stride=2, padding=1),
        nn.ZeroPad2d((0, 1, 0, 1)),
        nn.BatchNorm2d(64, momentum=0.82),
        nn.LeakyReLU(0.25),
        nn.Dropout(0.25),
        nn.Conv2d(64, 128, kernel_size=3, stride=2, padding=1),
        nn.BatchNorm2d(128, momentum=0.82),
        nn.LeakyReLU(0.2),
        nn.Dropout(0.25),
        nn.Conv2d(128, 256, kernel_size=3, stride=1, padding=1),
        nn.BatchNorm2d(256, momentum=0.8),
        nn.LeakyReLU(0.25),
        nn.Dropout(0.25),
        nn.Flatten(),
        nn.Linear(256 * 5 * 5, 1),
        nn.Sigmoid() # Sigmoid to predict real/fake
    )

    def forward(self, img):
        return self.model(img)

# Initialize Generator and Discriminator
generator = Generator(latent_dim).to(device)
discriminator = Discriminator().to(device)

# Loss function
adversarial_loss = nn.BCELoss()

# Optimizers for both models
optimizer_G = optim.Adam(generator.parameters(), lr=lr, betas=(beta1, beta2))
optimizer_D = optim.Adam(discriminator.parameters(), lr=lr, betas=(beta1, beta2))

# Training loop
for epoch in range(num_epochs):
    for i, batch in enumerate(dataloader):
        real_images = batch[0].to(device)
        valid = torch.ones(real_images.size(0), 1, device=device)
        fake = torch.zeros(real_images.size(0), 1, device=device)

        # Train Discriminator

        optimizer_D.zero_grad()

        # Sample noise and generate fake images
        z = torch.randn(real_images.size(0), latent_dim, device=device) # Latent space

```

```

fake_images = generator(z) # Generate fake images

# Compute discriminator loss
real_loss = adversarial_loss(discriminator(real_images), valid)
fake_loss = adversarial_loss(discriminator(fake_images.detach()), fake)
d_loss = (real_loss + fake_loss) / 2

# Backpropagate and update the discriminator
d_loss.backward()
optimizer_D.step()

# Train Generator

optimizer_G.zero_grad()

# Generate fake images again
gen_images = generator(z)

# Compute generator loss
g_loss = adversarial_loss(discriminator(gen_images), valid)

# Backpropagate and update the generator
g_loss.backward()
optimizer_G.step()

if (i + 1) % 100 == 0:
    print(f'Epoch [{epoch+1}/{num_epochs}] Batch {i+1}/{len(dataloader)} '
          f'Discriminator Loss: {d_loss.item():.4f} Generator Loss: {g_loss.item():.4f}')

# Save generated images for every epoch
if (epoch + 1) % 10 == 0:
    with torch.no_grad():
        z = torch.randn(16, latent_dim, device=device)
        generated = generator(z).detach().cpu()
        grid = torchvision.utils.make_grid(generated, nrow=4, normalize=True)
        plt.imshow(np.transpose(grid, (1, 2, 0)))
        plt.axis("off")
        plt.show()

```

OUTPUT :

```
Downloading https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz to ./data/cifar-10-python.tar.gz
100%|██████████| 170M/170M [00:04<00:00, 42.2MB/s]
Extracting ./data/cifar-10-python.tar.gz to ./data
Epoch [1/10] Batch 100/1563 Discriminator Loss: 0.5745 Generator Loss: 1.4467
Epoch [1/10] Batch 200/1563 Discriminator Loss: 0.8197 Generator Loss: 0.8564
Epoch [1/10] Batch 300/1563 Discriminator Loss: 0.6418 Generator Loss: 0.9940
Epoch [1/10] Batch 400/1563 Discriminator Loss: 0.5475 Generator Loss: 1.3837
Epoch [1/10] Batch 500/1563 Discriminator Loss: 0.6735 Generator Loss: 1.2630
Epoch [1/10] Batch 600/1563 Discriminator Loss: 0.7789 Generator Loss: 0.7003
Epoch [1/10] Batch 700/1563 Discriminator Loss: 0.7065 Generator Loss: 1.0550
Epoch [1/10] Batch 800/1563 Discriminator Loss: 0.7095 Generator Loss: 1.0452
Epoch [1/10] Batch 900/1563 Discriminator Loss: 0.6573 Generator Loss: 0.9470
Epoch [1/10] Batch 1000/1563 Discriminator Loss: 0.6706 Generator Loss: 0.9152
Epoch [1/10] Batch 1100/1563 Discriminator Loss: 0.5890 Generator Loss: 0.9628
Epoch [1/10] Batch 1200/1563 Discriminator Loss: 0.6281 Generator Loss: 1.0012
Epoch [1/10] Batch 1300/1563 Discriminator Loss: 0.5867 Generator Loss: 1.2633
Epoch [1/10] Batch 1400/1563 Discriminator Loss: 0.5972 Generator Loss: 1.0518
Epoch [1/10] Batch 1500/1563 Discriminator Loss: 0.6760 Generator Loss: 0.9922
Epoch [2/10] Batch 100/1563 Discriminator Loss: 0.5538 Generator Loss: 1.1048
Epoch [2/10] Batch 200/1563 Discriminator Loss: 0.6561 Generator Loss: 0.9170
Epoch [2/10] Batch 300/1563 Discriminator Loss: 0.6438 Generator Loss: 0.8676
Epoch [2/10] Batch 400/1563 Discriminator Loss: 0.8785 Generator Loss: 0.8077
Epoch [2/10] Batch 500/1563 Discriminator Loss: 0.6555 Generator Loss: 0.9876
Epoch [2/10] Batch 600/1563 Discriminator Loss: 0.6834 Generator Loss: 0.8081
Epoch [2/10] Batch 700/1563 Discriminator Loss: 0.6809 Generator Loss: 1.0555
Epoch [2/10] Batch 800/1563 Discriminator Loss: 0.7741 Generator Loss: 0.8928
Epoch [2/10] Batch 900/1563 Discriminator Loss: 0.5742 Generator Loss: 1.0952
Epoch [2/10] Batch 1000/1563 Discriminator Loss: 0.6333 Generator Loss: 1.0209
```

```
Epoch [2/10] Batch 1100/1563 Discriminator Loss: 0.5677 Generator Loss: 0.9551
Epoch [2/10] Batch 1200/1563 Discriminator Loss: 0.5031 Generator Loss: 1.1467
Epoch [2/10] Batch 1300/1563 Discriminator Loss: 0.5098 Generator Loss: 1.1826
Epoch [2/10] Batch 1400/1563 Discriminator Loss: 0.6197 Generator Loss: 1.2225
Epoch [2/10] Batch 1500/1563 Discriminator Loss: 0.5664 Generator Loss: 1.2609
Epoch [3/10] Batch 100/1563 Discriminator Loss: 0.5721 Generator Loss: 1.7740
Epoch [3/10] Batch 200/1563 Discriminator Loss: 0.3977 Generator Loss: 0.8989
Epoch [3/10] Batch 300/1563 Discriminator Loss: 0.4870 Generator Loss: 1.0619
Epoch [3/10] Batch 400/1563 Discriminator Loss: 0.6620 Generator Loss: 1.3916
Epoch [3/10] Batch 500/1563 Discriminator Loss: 0.5573 Generator Loss: 0.6968
Epoch [3/10] Batch 600/1563 Discriminator Loss: 0.5280 Generator Loss: 1.4265
Epoch [3/10] Batch 700/1563 Discriminator Loss: 0.6936 Generator Loss: 0.8045
Epoch [3/10] Batch 800/1563 Discriminator Loss: 0.6619 Generator Loss: 1.2627
Epoch [3/10] Batch 900/1563 Discriminator Loss: 0.5159 Generator Loss: 1.1222
Epoch [3/10] Batch 1000/1563 Discriminator Loss: 0.5847 Generator Loss: 1.1352
Epoch [3/10] Batch 1100/1563 Discriminator Loss: 0.3291 Generator Loss: 1.4526
Epoch [3/10] Batch 1200/1563 Discriminator Loss: 0.4952 Generator Loss: 1.3053
Epoch [3/10] Batch 1300/1563 Discriminator Loss: 0.6531 Generator Loss: 1.3212
Epoch [3/10] Batch 1400/1563 Discriminator Loss: 0.6813 Generator Loss: 1.3028
Epoch [3/10] Batch 1500/1563 Discriminator Loss: 0.5862 Generator Loss: 1.2482
Epoch [4/10] Batch 100/1563 Discriminator Loss: 0.5290 Generator Loss: 1.3146
Epoch [4/10] Batch 200/1563 Discriminator Loss: 0.6265 Generator Loss: 1.0730
Epoch [4/10] Batch 300/1563 Discriminator Loss: 0.6495 Generator Loss: 1.0531
Epoch [4/10] Batch 400/1563 Discriminator Loss: 0.6102 Generator Loss: 1.0971
Epoch [4/10] Batch 500/1563 Discriminator Loss: 0.6705 Generator Loss: 1.5779
Epoch [4/10] Batch 600/1563 Discriminator Loss: 0.5904 Generator Loss: 0.7792
Epoch [4/10] Batch 700/1563 Discriminator Loss: 0.3972 Generator Loss: 0.9888
Epoch [4/10] Batch 800/1563 Discriminator Loss: 0.5531 Generator Loss: 0.9137
```

```
Epoch [4/10] Batch 900/1563 Discriminator Loss: 0.5906 Generator Loss: 0.9865
Epoch [4/10] Batch 1000/1563 Discriminator Loss: 0.6848 Generator Loss: 0.8131
Epoch [4/10] Batch 1100/1563 Discriminator Loss: 0.6270 Generator Loss: 1.0064
Epoch [4/10] Batch 1200/1563 Discriminator Loss: 0.8108 Generator Loss: 1.2412
Epoch [4/10] Batch 1300/1563 Discriminator Loss: 0.4088 Generator Loss: 1.6907
Epoch [4/10] Batch 1400/1563 Discriminator Loss: 0.4347 Generator Loss: 1.8275
Epoch [4/10] Batch 1500/1563 Discriminator Loss: 0.5731 Generator Loss: 0.9994
Epoch [5/10] Batch 100/1563 Discriminator Loss: 0.7262 Generator Loss: 1.1746
Epoch [5/10] Batch 200/1563 Discriminator Loss: 0.4070 Generator Loss: 1.6474
Epoch [5/10] Batch 300/1563 Discriminator Loss: 0.6541 Generator Loss: 1.2130
Epoch [5/10] Batch 400/1563 Discriminator Loss: 0.5686 Generator Loss: 1.1974
Epoch [5/10] Batch 500/1563 Discriminator Loss: 0.6711 Generator Loss: 1.6026
Epoch [5/10] Batch 600/1563 Discriminator Loss: 0.4470 Generator Loss: 1.8596
Epoch [5/10] Batch 700/1563 Discriminator Loss: 0.7790 Generator Loss: 1.2159
Epoch [5/10] Batch 800/1563 Discriminator Loss: 0.7324 Generator Loss: 1.8639
Epoch [5/10] Batch 900/1563 Discriminator Loss: 0.7639 Generator Loss: 1.2598
Epoch [5/10] Batch 1000/1563 Discriminator Loss: 0.7390 Generator Loss: 0.9058
Epoch [5/10] Batch 1100/1563 Discriminator Loss: 0.5609 Generator Loss: 1.3039
Epoch [5/10] Batch 1200/1563 Discriminator Loss: 0.4314 Generator Loss: 1.7995
Epoch [5/10] Batch 1300/1563 Discriminator Loss: 0.5228 Generator Loss: 1.6312
Epoch [5/10] Batch 1400/1563 Discriminator Loss: 0.5785 Generator Loss: 1.0496
Epoch [5/10] Batch 1500/1563 Discriminator Loss: 0.6171 Generator Loss: 1.3908
Epoch [6/10] Batch 100/1563 Discriminator Loss: 0.4739 Generator Loss: 2.1964
Epoch [6/10] Batch 200/1563 Discriminator Loss: 0.7407 Generator Loss: 0.9552
Epoch [6/10] Batch 300/1563 Discriminator Loss: 0.8169 Generator Loss: 1.2808
Epoch [6/10] Batch 400/1563 Discriminator Loss: 0.4568 Generator Loss: 1.5499
Epoch [6/10] Batch 500/1563 Discriminator Loss: 0.6527 Generator Loss: 2.0531
Epoch [6/10] Batch 600/1563 Discriminator Loss: 0.4803 Generator Loss: 1.2961
```

```
Epoch [6/10] Batch 700/1563 Discriminator Loss: 0.7843 Generator Loss: 0.7619
Epoch [6/10] Batch 800/1563 Discriminator Loss: 0.5150 Generator Loss: 0.8410
Epoch [6/10] Batch 900/1563 Discriminator Loss: 0.4531 Generator Loss: 0.9930
Epoch [6/10] Batch 1000/1563 Discriminator Loss: 0.4714 Generator Loss: 2.0468
Epoch [6/10] Batch 1100/1563 Discriminator Loss: 0.4713 Generator Loss: 1.1189
Epoch [6/10] Batch 1200/1563 Discriminator Loss: 0.6002 Generator Loss: 1.0968
Epoch [6/10] Batch 1300/1563 Discriminator Loss: 0.5357 Generator Loss: 0.9317
Epoch [6/10] Batch 1400/1563 Discriminator Loss: 0.5832 Generator Loss: 1.4388
Epoch [6/10] Batch 1500/1563 Discriminator Loss: 0.6302 Generator Loss: 1.0179
Epoch [7/10] Batch 100/1563 Discriminator Loss: 0.7931 Generator Loss: 1.5894
Epoch [7/10] Batch 200/1563 Discriminator Loss: 0.5064 Generator Loss: 0.6649
Epoch [7/10] Batch 300/1563 Discriminator Loss: 0.5900 Generator Loss: 0.6795
Epoch [7/10] Batch 400/1563 Discriminator Loss: 0.4263 Generator Loss: 1.5336
Epoch [7/10] Batch 500/1563 Discriminator Loss: 0.2786 Generator Loss: 1.2543
Epoch [7/10] Batch 600/1563 Discriminator Loss: 0.5823 Generator Loss: 1.0702
Epoch [7/10] Batch 700/1563 Discriminator Loss: 1.0532 Generator Loss: 0.6475
Epoch [7/10] Batch 800/1563 Discriminator Loss: 0.4752 Generator Loss: 1.2546
Epoch [7/10] Batch 900/1563 Discriminator Loss: 0.5045 Generator Loss: 0.8967
Epoch [7/10] Batch 1000/1563 Discriminator Loss: 0.6303 Generator Loss: 1.1613
Epoch [7/10] Batch 1100/1563 Discriminator Loss: 0.5062 Generator Loss: 1.8349
Epoch [7/10] Batch 1200/1563 Discriminator Loss: 0.6101 Generator Loss: 0.6606
Epoch [7/10] Batch 1300/1563 Discriminator Loss: 0.5520 Generator Loss: 1.4129
Epoch [7/10] Batch 1400/1563 Discriminator Loss: 0.6727 Generator Loss: 1.0635
Epoch [7/10] Batch 1500/1563 Discriminator Loss: 0.7505 Generator Loss: 0.9012
Epoch [8/10] Batch 100/1563 Discriminator Loss: 0.5717 Generator Loss: 1.2343
Epoch [8/10] Batch 200/1563 Discriminator Loss: 0.5547 Generator Loss: 1.1601
Epoch [8/10] Batch 300/1563 Discriminator Loss: 0.5946 Generator Loss: 1.4899
Epoch [8/10] Batch 400/1563 Discriminator Loss: 0.4337 Generator Loss: 1.2628
```

```
Epoch [8/10] Batch 500/1563 Discriminator Loss: 0.4089 Generator Loss: 1.6692
Epoch [8/10] Batch 600/1563 Discriminator Loss: 0.6496 Generator Loss: 1.0786
Epoch [8/10] Batch 700/1563 Discriminator Loss: 0.5380 Generator Loss: 0.8724
Epoch [8/10] Batch 800/1563 Discriminator Loss: 0.7403 Generator Loss: 1.6225
Epoch [8/10] Batch 900/1563 Discriminator Loss: 0.5181 Generator Loss: 1.3905
Epoch [8/10] Batch 1000/1563 Discriminator Loss: 0.6168 Generator Loss: 1.2573
Epoch [8/10] Batch 1100/1563 Discriminator Loss: 0.7884 Generator Loss: 0.9349
Epoch [8/10] Batch 1200/1563 Discriminator Loss: 0.7354 Generator Loss: 0.5850
Epoch [8/10] Batch 1300/1563 Discriminator Loss: 0.4245 Generator Loss: 1.9349
Epoch [8/10] Batch 1400/1563 Discriminator Loss: 0.4125 Generator Loss: 1.3667
Epoch [8/10] Batch 1500/1563 Discriminator Loss: 0.6452 Generator Loss: 1.5506
Epoch [9/10] Batch 100/1563 Discriminator Loss: 0.4158 Generator Loss: 1.2288
Epoch [9/10] Batch 200/1563 Discriminator Loss: 0.4390 Generator Loss: 1.2814
Epoch [9/10] Batch 300/1563 Discriminator Loss: 0.6351 Generator Loss: 0.9215
Epoch [9/10] Batch 400/1563 Discriminator Loss: 0.5345 Generator Loss: 1.2378
Epoch [9/10] Batch 500/1563 Discriminator Loss: 0.5275 Generator Loss: 0.7853
Epoch [9/10] Batch 600/1563 Discriminator Loss: 0.5799 Generator Loss: 1.2538
Epoch [9/10] Batch 700/1563 Discriminator Loss: 0.6635 Generator Loss: 1.0088
Epoch [9/10] Batch 800/1563 Discriminator Loss: 0.6335 Generator Loss: 0.9135
Epoch [9/10] Batch 900/1563 Discriminator Loss: 0.6835 Generator Loss: 1.3035
Epoch [9/10] Batch 1000/1563 Discriminator Loss: 0.5679 Generator Loss: 0.9503
Epoch [9/10] Batch 1100/1563 Discriminator Loss: 0.8647 Generator Loss: 1.1308
Epoch [9/10] Batch 1200/1563 Discriminator Loss: 0.2885 Generator Loss: 1.7467
Epoch [9/10] Batch 1300/1563 Discriminator Loss: 0.3590 Generator Loss: 1.2848
Epoch [9/10] Batch 1400/1563 Discriminator Loss: 0.5680 Generator Loss: 0.8865
Epoch [9/10] Batch 1500/1563 Discriminator Loss: 0.7409 Generator Loss: 1.1086
Epoch [10/10] Batch 100/1563 Discriminator Loss: 0.4533 Generator Loss: 1.1949
Epoch [10/10] Batch 200/1563 Discriminator Loss: 0.4877 Generator Loss: 1.2146
```

```
Epoch [10/10] Batch 300/1563 Discriminator Loss: 0.5462 Generator Loss: 0.7146
Epoch [10/10] Batch 400/1563 Discriminator Loss: 1.0229 Generator Loss: 0.9949
Epoch [10/10] Batch 500/1563 Discriminator Loss: 0.5761 Generator Loss: 1.4498
Epoch [10/10] Batch 600/1563 Discriminator Loss: 0.5318 Generator Loss: 0.9658
Epoch [10/10] Batch 700/1563 Discriminator Loss: 0.5056 Generator Loss: 1.1784
Epoch [10/10] Batch 800/1563 Discriminator Loss: 0.8730 Generator Loss: 1.3191
Epoch [10/10] Batch 900/1563 Discriminator Loss: 0.6652 Generator Loss: 0.9617
Epoch [10/10] Batch 1000/1563 Discriminator Loss: 0.5836 Generator Loss: 0.9748
Epoch [10/10] Batch 1100/1563 Discriminator Loss: 0.4097 Generator Loss: 1.7757
Epoch [10/10] Batch 1200/1563 Discriminator Loss: 0.4569 Generator Loss: 1.2204
Epoch [10/10] Batch 1300/1563 Discriminator Loss: 0.6526 Generator Loss: 0.8639
Epoch [10/10] Batch 1400/1563 Discriminator Loss: 0.5287 Generator Loss: 0.9376
Epoch [10/10] Batch 1500/1563 Discriminator Loss: 0.7067 Generator Loss: 1.0947
```




EXPERIMENT – 09

AIM : Outlier detection in time series dataset using RNN.

THEORY :

Outliers are data points that deviate significantly from the normal behavior of the time series. In time series data, outliers are important because they can be caused by unusual events (e.g., system failures, fraud, or external interventions) and may distort model predictions.

RNN for Outlier Detection

- **RNN (Recurrent Neural Networks)** are neural networks designed for sequential data, with loops that allow the network to maintain a memory of previous inputs.
- **LSTM (Long Short-Term Memory)** is a special type of RNN that solves the vanishing gradient problem, making it effective for learning long-term dependencies in time series data.
- In **outlier detection**, an RNN (or LSTM) model is trained on the time series to predict future values. The difference between predicted and actual values is then used to detect anomalies or outliers. Large discrepancies between predicted and actual values are flagged as outliers.

CODE :

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.preprocessing import MinMaxScaler
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Input, LSTM, Dense
from sklearn.metrics import mean_absolute_error

np.random.seed(42)
time_steps = 200
trend = np.linspace(0, 50, time_steps)
noise = np.random.normal(0, 2, time_steps)

data = trend + noise
data[50] = 80
data[150] = -30

# Converting data to a pandas DataFrame
df = pd.DataFrame(data, columns=['value'])
```

```

plt.figure(figsize=(10, 6))
plt.plot(df['value'])
plt.title("Synthetic Time Series with Outliers")
plt.show()

# Normalize the data to the range [0, 1]
scaler = MinMaxScaler(feature_range=(0, 1))
scaled_data = scaler.fit_transform(df['value'].values.reshape(-1, 1))

# Function to create sequences from the time series data
def create_sequences(data, seq_length):
    x, y = [], []
    for i in range(len(data) - seq_length):
        x.append(data[i:i+seq_length])
        y.append(data[i+seq_length])
    return np.array(x), np.array(y)

# Create sequences of length 10
seq_length = 10
X, y = create_sequences(scaled_data, seq_length)

# Reshape X to be [samples, time steps, features] for LSTM
X = X.reshape(X.shape[0], X.shape[1], 1)

train_size = int(len(X) * 0.8)
X_train, X_test = X[:train_size], X[train_size:]
y_train, y_test = y[:train_size], y[train_size:]

# Build the LSTM model
model = Sequential()
model.add(Input(shape=(X_train.shape[1], 1)))
model.add(LSTM(64, return_sequences=False))
model.add(Dense(1))
model.compile(optimizer='adam', loss='mean_squared_error')

model.fit(X_train, y_train, epochs=20, batch_size=32, validation_data=(X_test, y_test))

y_pred = model.predict(X_test)

# Inverse scale the predictions and actual values
y_pred_rescaled = scaler.inverse_transform(y_pred)
y_test_rescaled = scaler.inverse_transform(y_test.reshape(-1, 1))

# absolute error
errors = np.abs(y_pred_rescaled - y_test_rescaled)

```



```

# threshold for outlier detection
threshold = np.mean(errors) + 2 * np.std(errors)

outliers = errors > threshold

# outlier indices
outlier_indices = np.where(outliers)[0]
outlier_predictions = y_pred_rescaled[outlier_indices]
outlier_actuals = y_test_rescaled[outlier_indices]

# Plot the results
plt.figure(figsize=(10, 6))
plt.plot(df.index[train_size + seq_length:], y_test_rescaled, label='True Values')
plt.plot(df.index[train_size + seq_length:], y_pred_rescaled, label='Predicted Values')

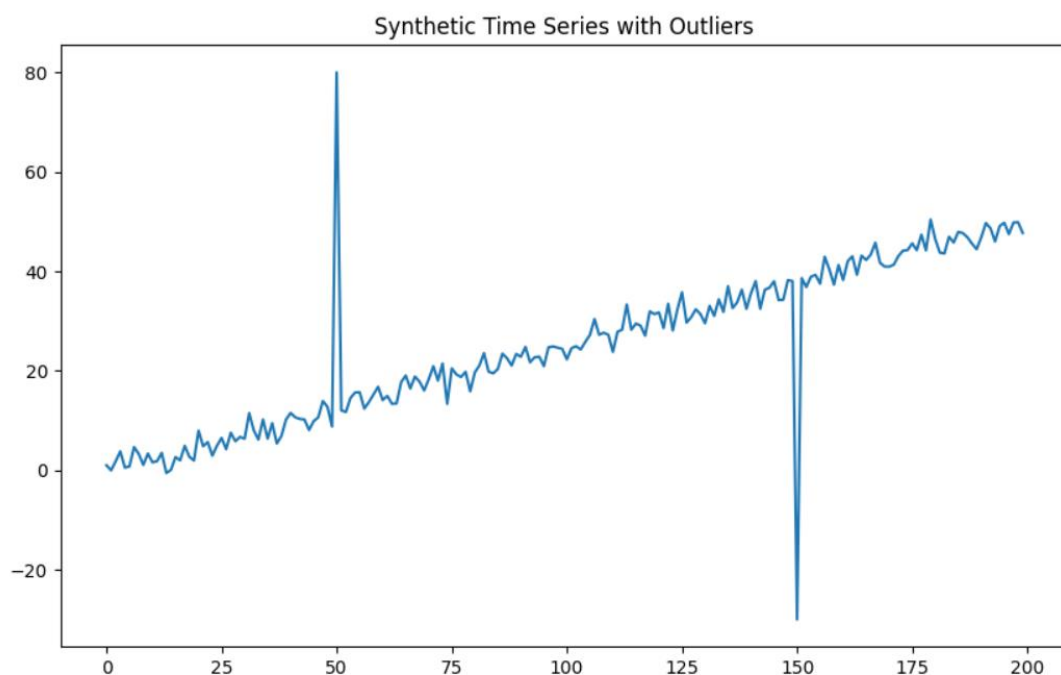
# Scatter plot for the outliers
plt.scatter(df.index[train_size + seq_length:][outlier_indices], outlier_predictions, color='red',
label='Outliers')

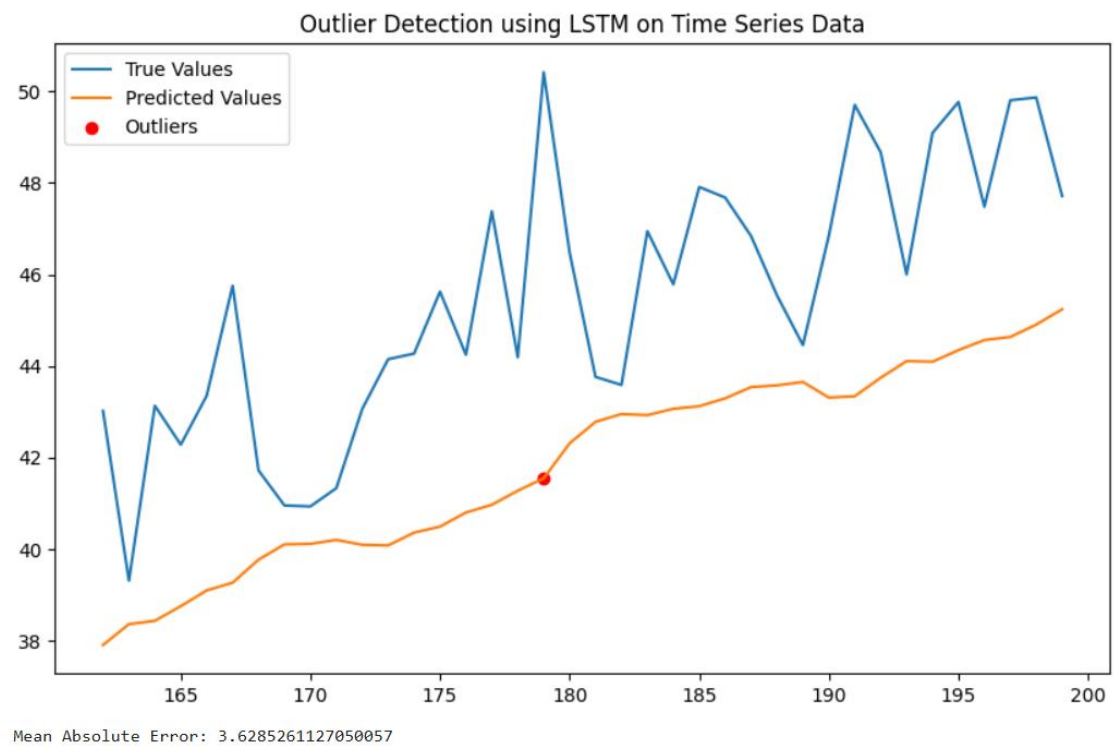
plt.legend()
plt.title("Outlier Detection using LSTM on Time Series Data")
plt.show()

# Mean Absolute Error
mae = mean_absolute_error(y_test_rescaled, y_pred_rescaled)
print(f'Mean Absolute Error: {mae}')

```

OUTPUT :





EXPERIMENT – 10

AIM : Anomaly detection using Self-Organizing Network.

THEORY :

Anomaly detection refers to the identification of data points that deviate significantly from the normal pattern in a dataset. Anomalies may indicate important events such as fraud, system failures, or rare occurrences that are worth investigating.

A Self-Organizing Map (SOM) is a type of unsupervised neural network that reduces the dimensionality of the data while preserving the topological features. SOMs map high-dimensional data to a lower-dimensional (typically 2D) grid, where similar data points are grouped together.

- **Best Matching Unit (BMU)**: The neuron in the SOM grid that is closest to a given input data point.
- **U-Matrix**: A visualization tool to view the distance between neighboring neurons in the SOM grid. Large distances indicate potential anomalies.

CODE :

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn import datasets
from sklearn.preprocessing import StandardScaler
from minisom import MiniSom

iris = datasets.load_iris()
X = iris.data
y = iris.target

# Normalize the data (SOM is sensitive to the scale of features)
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Create and train the Self-Organizing Map (SOM)
som = MiniSom(x=10, y=10, input_len=X_scaled.shape[1], sigma=1.0, learning_rate=0.5)
som.train(X_scaled, 100)

# Map the data points to the SOM grid
mapped = np.array([som.winner(x) for x in X_scaled])

# Calculate the Euclidean distance between each point and its winning SOM node
```

```

distances = np.linalg.norm(X_scaled - np.array([som.get_weights()[x[0], x[1]] for x in
mapped]), axis=1)

threshold = np.mean(distances) + 2 * np.std(distances)
anomalies = distances > threshold
plt.figure(figsize=(8, 8))

# Plot normal points (not anomalies)
plt.scatter(X_scaled[~anomalies, 0], X_scaled[~anomalies, 1], color='blue', label='Normal')

# Plot anomalies
plt.scatter(X_scaled[anomalies, 0], X_scaled[anomalies, 1], color='red', label='Anomaly')
plt.title("Anomaly Detection using Self-Organizing Map on Iris Dataset")
plt.xlabel("Feature 1 (Sepal Length - scaled)")
plt.ylabel("Feature 2 (Sepal Width - scaled)")
plt.legend()
plt.show()

```

OUTPUT :

