**PART-I: Computer Networks**

**Q1**: Setup 4 nodes, two TCP client and server pair and two UDP client and server pair. Send packets to respective clients from both the servers. Monitor the traffic for both the pair and plot the number of bytes received. Make necessary assumptions.

To set up a network simulation with two TCP client-server pairs and two UDP client-server pairs, we'll need to perform the following steps:

1. **Setup Environment**: Use a programming language like Python to create TCP and UDP server-client pairs. We'll assume that all nodes (clients and servers) are running on localhost for simplicity.
2. **Define Traffic Monitoring**: We'll count the bytes received by each client and plot this data using a library like Matplotlib.
3. **Assumptions**:
   * Each client will receive a fixed number of packets from its respective server.
   * The servers will run indefinitely or until stopped manually.
   * A simple logging mechanism will be used to monitor traffic.

Here's a sample implementation:

**TCP Client-Server Setup**

**TCP Server (tcp\_server.py)**:

python

Copy code

import socket

import time

def start\_tcp\_server(port):

server\_socket = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)

server\_socket.bind(('localhost', port))

server\_socket.listen(1)

print(f"TCP Server listening on port {port}")

conn, addr = server\_socket.accept()

print(f"Connection from {addr}")

while True:

message = "Hello TCP Client!"

conn.sendall(message.encode())

time.sleep(1) # Send a packet every second

conn.close()

server\_socket.close()

if \_\_name\_\_ == "\_\_main\_\_":

start\_tcp\_server(5000)

**TCP Client (tcp\_client.py)**:

python

Copy code

import socket

def start\_tcp\_client(port):

client\_socket = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)

client\_socket.connect(('localhost', port))

total\_bytes\_received = 0

try:

while True:

data = client\_socket.recv(1024)

if not data:

break

total\_bytes\_received += len(data)

except KeyboardInterrupt:

pass

finally:

print(f"Total bytes received by TCP Client: {total\_bytes\_received}")

client\_socket.close()

if \_\_name\_\_ == "\_\_main\_\_":

start\_tcp\_client(5000)

**UDP Client-Server Setup**

**UDP Server (udp\_server.py)**:

python

Copy code

import socket

import time

def start\_udp\_server(port):

server\_socket = socket.socket(socket.AF\_INET, socket.SOCK\_DGRAM)

server\_socket.bind(('localhost', port))

print(f"UDP Server listening on port {port}")

while True:

message = "Hello UDP Client!"

server\_socket.sendto(message.encode(), ('localhost', 6000))

time.sleep(1) # Send a packet every second

if \_\_name\_\_ == "\_\_main\_\_":

start\_udp\_server(6000)

**UDP Client (udp\_client.py)**:

python

Copy code

import socket

def start\_udp\_client(port):

client\_socket = socket.socket(socket.AF\_INET, socket.SOCK\_DGRAM)

total\_bytes\_received = 0

while True:

data, addr = client\_socket.recvfrom(1024)

total\_bytes\_received += len(data)

print(f"Received {len(data)} bytes from {addr}")

if \_\_name\_\_ == "\_\_main\_\_":

start\_udp\_client(6000)

**Traffic Monitoring and Plotting**

To monitor the traffic and plot the results, you can use the following code snippet:

python

Copy code

import matplotlib.pyplot as plt

# Example data

tcp\_bytes\_received = [0] # Replace with actual received data

udp\_bytes\_received = [0] # Replace with actual received data

# Example of how to update the lists based on actual traffic

# tcp\_bytes\_received.append(new\_tcp\_bytes)

# udp\_bytes\_received.append(new\_udp\_bytes)

plt.plot(tcp\_bytes\_received, label='TCP Bytes Received')

plt.plot(udp\_bytes\_received, label='UDP Bytes Received')

plt.xlabel('Time (s)')

plt.ylabel('Bytes Received')

plt.title('Network Traffic Monitoring')

plt.legend()

plt.show()

**Running the Setup**

1. Open four terminal windows.
2. In the first terminal, run the TCP server:

bash

Copy code

python tcp\_server.py

1. In the second terminal, run the TCP client:

bash

Copy code

python tcp\_client.py

1. In the third terminal, run the UDP server:

bash

Copy code

python udp\_server.py

1. In the fourth terminal, run the UDP client:

bash

Copy code

python udp\_client.py

**Monitoring Traffic**

* The total bytes received will be printed by the clients.
* You can periodically update the tcp\_bytes\_received and udp\_bytes\_received lists and plot them.

**Notes**

* Make sure to handle exceptions and cleanup resources properly.
* You can run the clients and servers on different machines in a real network scenario.
* Adjust sleep times, message sizes, and protocols according to your requirements.

**PART-II: Data Mining Lab**

Q1. iris.arff

To implement a simple K-Means algorithm to demonstrate clustering on the Iris dataset (commonly available in the .arff format), you can follow the steps below. We'll use Python with libraries such as numpy, pandas, and matplotlib for data handling and visualization.

### Step 1: Install Required Libraries

Make sure you have the required libraries installed. You can install them using pip:

bash

Copy code

pip install numpy pandas matplotlib scikit-learn

### Step 2: Load the Iris Dataset

You can use the pandas library to read the .arff file. If you don't have the .arff file, you can convert it to CSV or directly use the dataset from the UCI Machine Learning Repository.

For this example, I'll show how to load the Iris dataset using the sklearn library, which has a built-in function to fetch it directly.

### Step 3: K-Means Implementation

Here’s a complete code snippet to implement the K-Means algorithm on the Iris dataset and visualize the clustering results.

python

Copy code

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

from sklearn import datasets

from sklearn.cluster import KMeans

from sklearn.preprocessing import StandardScaler

from sklearn.decomposition import PCA

# Step 1: Load the iris dataset

iris = datasets.load\_iris()

X = iris.data # features

y = iris.target # labels (actual species)

# Step 2: Standardize the features

scaler = StandardScaler()

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

# Step 3: Implement K-Means

# Set the number of clusters to 3 (for the 3 species)

kmeans = KMeans(n\_clusters=3, random\_state=42)

kmeans.fit(X\_scaled)

# Step 4: Get cluster labels

labels = kmeans.labels\_

# Step 5: Reduce dimensions for visualization (using PCA)

pca = PCA(n\_components=2)

X\_pca = pca.fit\_transform(X\_scaled)

# Step 6: Plot the results

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

plt.scatter(X\_pca[:, 0], X\_pca[:, 1], c=labels, cmap='viridis', marker='o', edgecolor='k', s=50)

plt.scatter(kmeans.cluster\_centers\_[:, 0], kmeans.cluster\_centers\_[:, 1], c='red', marker='X', s=200, label='Centroids')

plt.title('K-Means Clustering on Iris Dataset')

plt.xlabel('Principal Component 1')

plt.ylabel('Principal Component 2')

plt.legend()

plt.grid()

plt.show()

### Explanation of the Code

1. **Load the Dataset**: The Iris dataset is loaded using sklearn.datasets.
2. **Standardization**: The features are standardized to have a mean of 0 and a variance of 1 using StandardScaler.
3. **K-Means Clustering**: The KMeans algorithm is applied to the standardized data to cluster the data into 3 groups.
4. **PCA for Visualization**: Principal Component Analysis (PCA) is used to reduce the dimensionality of the dataset from 4 to 2 for visualization purposes.
5. **Plotting**: The resulting clusters and the centroids are plotted using matplotlib.

### Step 4: Run the Code

Save the above code in a Python script (e.g., kmeans\_iris.py) and run it. You should see a scatter plot showing the clusters formed by the K-Means algorithm on the Iris dataset, along with the centroids marked in red.

### Notes

* You can change the n\_clusters parameter in the KMeans object to test with different numbers of clusters.
* If you want to load the dataset from a .arff file, you can use the scipy library's arff module to load it, but the example provided above uses a direct approach with sklearn.

b) Student.arff.

To demonstrate the K-Means clustering algorithm using a hypothetical dataset called student.arff, we will need to load the dataset, preprocess it, apply K-Means clustering, and visualize the results. Here's a step-by-step guide:

### Step 1: Sample student.arff Dataset

Assuming the student.arff dataset looks something like this (you can modify the content based on your actual dataset):

plaintext

Copy code

@RELATION student

@ATTRIBUTE age NUMERIC

@ATTRIBUTE study\_time NUMERIC

@ATTRIBUTE grades NUMERIC

@ATTRIBUTE passed {yes, no}

@DATA

18, 5, 90, yes

19, 3, 70, no

20, 4, 85, yes

22, 2, 60, no

21, 5, 95, yes

19, 4, 80, no

20, 5, 92, yes

### Step 2: Install Required Libraries

Ensure you have the necessary libraries installed:

bash

Copy code

pip install numpy pandas matplotlib scikit-learn scipy

### Step 3: Load the Dataset

You can load the .arff file using scipy or pandas. Here's how to do it using pandas:

python

Copy code

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

from sklearn.cluster import KMeans

from sklearn.preprocessing import StandardScaler

from sklearn.decomposition import PCA

from scipy.io import arff

# Step 1: Load the student dataset from an .arff file

data = arff.loadarff('student.arff')

df = pd.DataFrame(data[0]) # Convert to a DataFrame

# Convert byte columns to strings if necessary

for col in df.select\_dtypes([np.object]):

df[col] = df[col].str.decode('utf-8')

# Step 2: Select features for clustering (assuming 'passed' is not needed for clustering)

X = df[['age', 'study\_time', 'grades']].values

# Step 3: Standardize the features

scaler = StandardScaler()

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

# Step 4: Implement K-Means

kmeans = KMeans(n\_clusters=2, random\_state=42) # Assuming two clusters

kmeans.fit(X\_scaled)

# Step 5: Get cluster labels

labels = kmeans.labels\_

# Step 6: Reduce dimensions for visualization (using PCA)

pca = PCA(n\_components=2)

X\_pca = pca.fit\_transform(X\_scaled)

# Step 7: Plot the results

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

plt.scatter(X\_pca[:, 0], X\_pca[:, 1], c=labels, cmap='viridis', marker='o', edgecolor='k', s=50)

plt.scatter(kmeans.cluster\_centers\_[:, 0], kmeans.cluster\_centers\_[:, 1], c='red', marker='X', s=200, label='Centroids')

plt.title('K-Means Clustering on Student Dataset')

plt.xlabel('Principal Component 1')

plt.ylabel('Principal Component 2')

plt.legend()

plt.grid()

plt.show()

### Explanation of the Code

1. **Load the Dataset**: The student.arff dataset is loaded using scipy.io.arff and converted into a pandas DataFrame.
2. **Data Preprocessing**: The features relevant for clustering (age, study\_time, grades) are selected. If any of the columns are in bytes format (like the passed column), we decode them to strings.
3. **Standardization**: The features are standardized using StandardScaler.
4. **K-Means Clustering**: K-Means clustering is applied to the standardized data.
5. **PCA for Visualization**: PCA is used to reduce the dimensionality of the data to 2D for easy visualization.
6. **Plotting**: A scatter plot is created to visualize the clusters, with centroids marked in red.

### Step 4: Run the Code

Save the code in a Python script (e.g., kmeans\_student.py) and run it. Ensure that the student.arff file is in the same directory as your script. You should see a scatter plot showing the clusters formed by the K-Means algorithm on the student dataset.