1. ***Explanation of Calculate the mean and standard deviation.***

1. Mean (Average):

The **mean** is the average of a set of numbers. It's calculated by adding up all the values in the data set and then dividing by the number of values.

Formula:

Mean=∑Xin\text{Mean} = \frac{\sum X_i}{n}

Where:

- ∑Xi\sum X_i is the sum of all the values in the data set
- nn is the number of values in the data set

Example: If your data set is: 2, 4, 6, 8, 10,

Mean=2+4+6+8+105=305=6\text{Mean} = \frac{2 + 4 + 6 + 8 + 10}{5} = \frac{30}{5} = 6

---

2. Standard Deviation:

The **standard deviation** measures the spread or variability of a set of data points around the mean. A higher standard deviation means the data points are more spread out from the mean, while a lower standard deviation means they are closer to the mean.

Steps to calculate standard deviation:

1. Find the **mean** of the data.
2. Subtract the mean from each data point and square the result (this is called the squared deviation).
3. Find the average of these squared deviations.
4. Take the square root of that average. This gives you the **standard deviation**.

Formula:

Standard Deviation=∑(Xi−μ)2n\text{Standard Deviation} = \sqrt{\frac{\sum (X_i - \mu)^2}{n}}
Where:

- XiX_i is each data point
- μ\mu is the mean
- nn is the number of data points

NAME: Rohan Bhimrao Patil **DATE:**

ROLL_NO: 150 **BATCH: B6**

PRACT_NAME: 1.calculate the mean and standard deviation

```
import numpy as np

# Sample data
data = np.array([10, 20, 30, 40, 50])

# Calculate mean
mean_value = np.mean(data)

# Calculate standard deviation
std_dev = np.std(data, ddof=1) # Set ddof=0 for population standard deviation

# Display results
print(f"Mean: {mean_value}")
print(f"Standard Deviation: {std_dev}")
```

output:

Mean: 30.0

Standard Deviation: 15.811388300841896

2 Explanation of Read the CSV file.

1. What is a CSV File?

A **CSV (Comma Separated Values)** file is a simple text file used to store data. Each line in the file represents a row, and each value in that row is separated by a comma. This structure makes it easy to organize and manage tabular data, such as a spreadsheet.

Example of a CSV file:

pgsql
Copy
Name, Age, City
John, 25, New York
Jane, 30, London
Tom, 22, Tokyo

- Each line is a **record** (or row), and each field (or value) is separated by a comma.
- The **first line** typically contains **column headers** (e.g., Name, Age, City) that define the data for each column.

2 .How to Read a CSV File:

Reading a CSV file means loading the data from the file into a usable data structure (like a table or list) so that it can be processed, analyzed, or displayed.

The process typically involves:

- **Opening the CSV file** to access its contents.
- **Parsing** the data so that each value can be extracted and interpreted correctly.
- **Storing the data** in a format that can be easily used for further processing (such as a list of lists, or a more structured format like a **DataFrame** in libraries like Panda
- **Why Use CSV?**
- **Portability**: CSV files can be opened by many different programs, from text editors to advanced data analysis tools.

- **Simplicity**: CSV files are simple text files that don’t require complex formats or software to create and read.
- **Ease of manipulation**: With the right tools (like Pandas in Python), working with CSV files is straightforward and efficient

NAME: Rohan Bhimrao Patil **DATE:**

ROLL_NO: 150 **BATCH: B6**

PRACT_NAME: 2.Read the csv file

```
import pandas as pd

# Load the data from a CSV file
file_path = 'hh.csv'  # Replace with your file path
sales_data = pd.read_csv(file_path)

# Display basic data structure
print("Display rows of the dataset:")
print(sales_data)
```

OUTPUT:

```
C:\Users\comp\PycharmProjects\pythonProject\venv\Scripts\python.exe
C:/Users/comp/PycharmProjects/pythonProject/jj.py
Display rows of the dataset:
      Date  Region Product  Sales
0   1/5/2023  North       A    100
1  1/12/2023  South       B    200
2   2/1/2023  North       A    150
3  2/14/2023   East       C    300
4  2/21/2023   West       B    250
```

3. Explanation of Perform data filtering, and calculate aggregate statistics.

1. **Data Filtering**:

Data filtering is the process of selecting specific rows from a dataset that meet certain conditions or criteria. It is essential when you want to focus on a subset of the data that matches specific requirements.

Common Data Filtering Operations:

- **Condition-based filtering**: You can filter data based on specific conditions. For example, selecting all records where a particular column (e.g., age or salary) is greater than a certain value.
- **Range-based filtering**: Select rows where values in a column fall within a specific range, such as ages between 20 and 30.
- **Category-based filtering**: Filter data based on categorical variables, like selecting all records from a particular city or country.

Example of Filtering:

- **Select records where "Age" > 30**: This would give you all individuals in the dataset who are older than 30 years.

Why Use Filtering?

- **Focus**: To focus on specific parts of the data that are most relevant to your analysis.
- **Data Quality**: To remove outliers, irrelevant data, or incomplete records.
- **Segmentation**: To divide the data into smaller, more meaningful subsets based on certain criteria.

2. **Calculating Aggregate Statistics**:

Aggregate statistics provide a summary of data by calculating certain metrics over columns or rows. These statistics are useful for understanding the overall trends, central tendency, and spread of the data.

Common Aggregate Statistics:

- **Mean (Average)**: The mean is the sum of all values in a column divided by the number of values. It represents the central tendency of the data.

  Mean=Sum of valuesNumber of values\text{Mean} = \frac{\text{Sum of values}}{\text{Number of values}}

- **Sum**: The total of all values in a column. It is useful when you want to know the overall magnitude or total count (e.g., total sales or total revenue).

NAME: Rohan Bhimrao Patil **DATE:**

ROLL_NO: 150 **BATCH: B6**

PRACT_NAME: 3.perfrom data filtering , and calculate aggregate statistics.

```
import pandas as pd

# Load the data from a CSV file
file_path = 'hh.csv'  # Replace with your file path
sales_data = pd.read_csv(file_path)

# Display basic data structure
print("First few rows of the dataset:")
print(sales_data.head())

# **Data Filtering**: Select data where Sales > 150
filtered_data = sales_data[sales_data['Sales'] > 150]
print("\nFiltered Data (Sales > 150):")
print(filtered_data)

# **Aggregate Statistics**: Calculate total and average sales by region
region_sales = sales_data.groupby('Region')['Sales'].agg(['sum', 'mean']).reset_index()
print("\nTotal and Average Sales by Region:")
print(region_sales)

# **Aggregate Statistics**: Calculate total sales and count by product
product_stats = sales_data.groupby('Product')['Sales'].agg(['sum', 'count']).reset_index()
print("\nTotal Sales and Transaction Count by Product:")
print(product_stats)
```

OUTPUT:

```
C:\Users\comp\PycharmProjects\pythonProject\venv\Scripts\python.exe
C:/Users/comp/PycharmProjects/pythonProject/3rd.py
First few rows of the dataset:
      Date  Region Product  Sales
0   1/5/2023  North       A    100
1  1/12/2023  South       B    200
2   2/1/2023  North       A    150
3  2/14/2023   East       C    300
4  2/21/2023   West       B    250
S

Filtered Data (Sales > 150):
      Date  Region Product  Sales
1  1/12/2023  South       B    200
3  2/14/2023   East       C    300
4  2/21/2023   West       B    250

Total and Average Sales by Region:
  Region  sum   mean
0   East  300  300.0
1  North  250  125.0
2  South  200  200.0
3   West  250  250.0

Total Sales and Transaction Count by Product:
  Product  sum  count
0       A  250      2
1       B  450      2
2       C  300      1
```

4 *Explanation of Calculate total sales by month*..

1. **Understanding the Data**:

In most cases, a dataset that tracks sales will include at least the following two columns:

- **Sales amount**: This column records the value of each sale (e.g., `Amount` or `Sales`).
- **Date**: This column records the date of each sale.

For this task, we'll assume the dataset contains columns like `Date` and `Sales`, and you want to group the data by **month** to calculate the total sales for each month.

2. **Steps for Calculating Total Sales by Month**:

a. Prepare the Data:

Ensure that your **Date column** is in a proper **datetime format** so you can easily extract the month. If it's in a text or string format, you'll need to convert it into a datetime format.

b. Extract the Month and Year:

To group by month, you typically need to extract both the **month** and the **year** from the Date column (so you can avoid confusion if your data spans multiple years).

c. Group by Month:

Once you have the date formatted, you can group the data by the **month** and **year** and then sum the sales for each group.

d. Calculate Total Sales:

After grouping the data, you can calculate the total sales for each month by summing up the `Sales` values within each group.

NAME: Rohan Bhimrao Patil **DATE:**

ROLL_NO: 150 **BATCH: B6**

PRACT_NAME: 4 Calculate total sales by month

```
import pandas as pd

# Sample data (replace with your actual data source, e.g., CSV file)

data = {
 "Date": ["2023-01-15", "2023-01-20", "2023-02-10", "2023-02-15", "2023-03-01"],
 "Sales": [200, 150, 300, 250, 400],
}

# Convert the data to a DataFrame
df = pd.DataFrame(data)

# Ensure the Date column is in datetime format
df['Date'] = pd.to_datetime(df['Date'])

# Extract year and month from the Date column
df['YearMonth'] = df['Date'].dt.to_period('M')

# Group by YearMonth and calculate total sales
monthly_sales = df.groupby('YearMonth')['Sales'].sum()

# Print the result
print(monthly_sales)
```

OUTPUT:

```
YearMonth
2023-01    350
2023-02    550
2023-03    40
```

5 Explanation of Implement the Clustering using K-means.

K-Means Clustering

K-Means clustering is an unsupervised machine learning algorithm used to divide a dataset into a set of clusters based on similarity. The goal of the K-means algorithm is to group data points such that the points in the same cluster are more similar to each other than to points in other clusters.

Steps for Implementing K-Means Clustering

1. **Select the number of clusters (K)**: The first step in K-means clustering is to decide how many clusters (K) you want to divide the data into. This can be done based on prior knowledge or techniques like the **Elbow Method**.
2. **Randomly initialize centroids**: The algorithm starts by randomly selecting K points from the dataset as initial centroids.
3. **Assign each point to the nearest centroid**: Each point in the dataset is assigned to the nearest centroid. The distance between a point and the centroid is usually calculated using the Euclidean distance.
4. **Update the centroids**: After assigning points to centroids, the centroids are recalculated as the mean of all the points in each cluster.
5. **Repeat**: Steps 3 and 4 are repeated until the centroids no longer change significantly, meaning the algorithm has converged.
6. **Result**: The final output is K clusters, with each point assigned to one cluster based on its proximity to the centroid.

NAME: Rohan Bhimrao Patil **DATE:**

ROLL_NO: 150 **BATCH: B6**

PRACT_NAME: 5 Implement the clustering using K-means

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

# Generate sample data
X, y = make_blobs(n_samples=300, centers=4, cluster_std=0.6, random_state=42)

# Visualize the raw data
plt.scatter(X[:, 0], X[:, 1], s=50, c='gray', marker='o')
plt.title("Raw Data")
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)

# Get cluster labels and centroids
labels = kmeans.labels_
centroids = kmeans.cluster_centers_

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

OUTPUT:

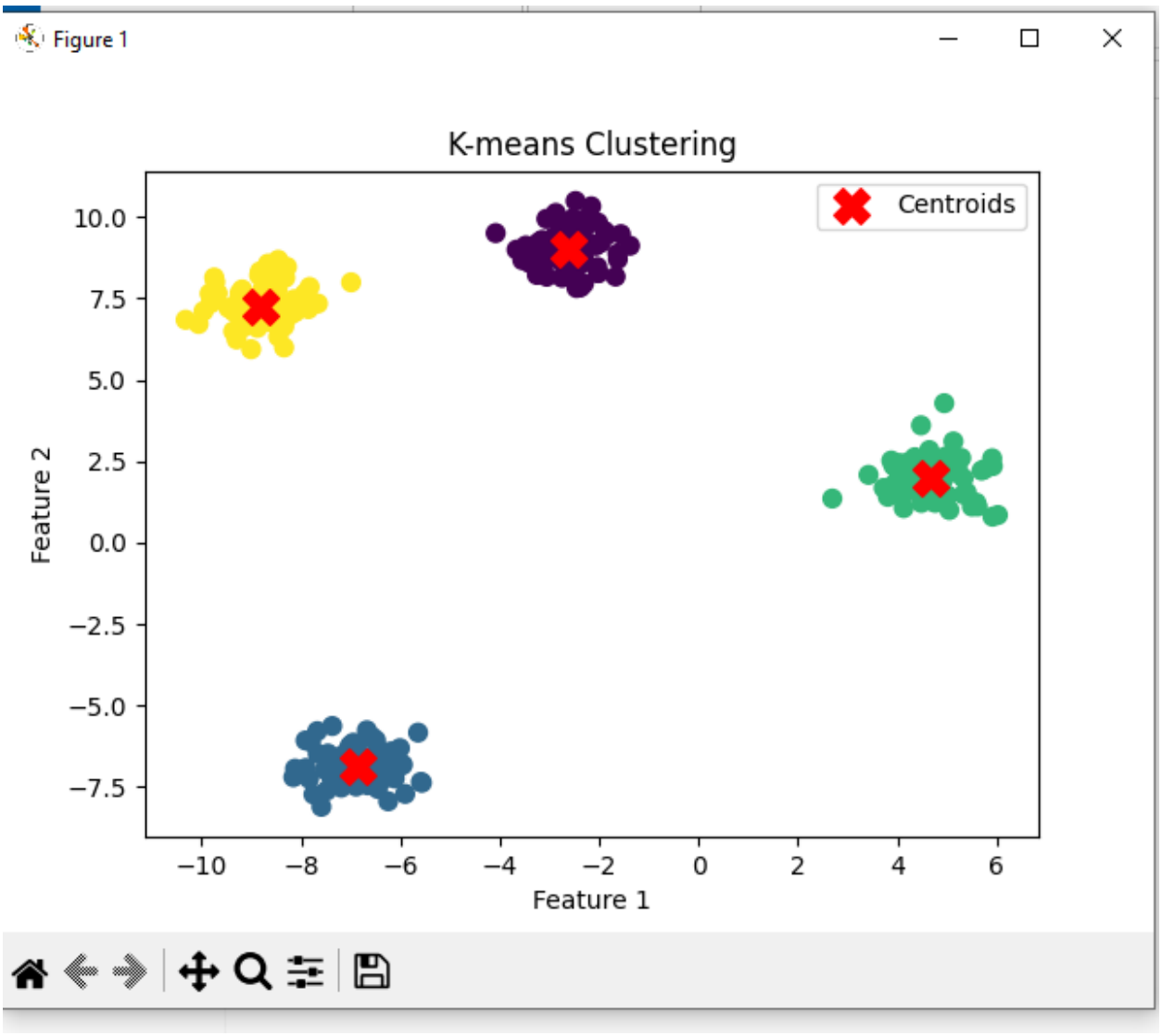


Figure 1
K-means Clustering
Centroids
10.0
7.5
5.0
2.5
0.0
−2.5
−5.0
−7.5
Feature 2
−10
−8
−6
−4
−2
0
2
4
6
Feature 1

6 . Explanation of Classification using Random Forest.

Random Forest is a versatile machine learning algorithm that can be used for both classification and regression tasks. It's an ensemble method that works by constructing a multitude of decision trees during training and outputs the class that is the majority vote from all the individual trees (in classification) or the average prediction (in regression).

Steps for Classification using Random Forest:

1. **Data Preparation**:
    - Collect and clean the data (handle missing values, categorical encoding, etc.).
    - Split the data into training and test sets (commonly 80-20 or 70-30 split).
2. **Train a Random Forest Model**:
    - Choose the number of trees (`n_estimators`) to be used in the forest.
    - Select other hyperparameters like maximum depth of trees (`max_depth`), minimum number of samples required to split a node (`min_samples_split`), and minimum samples per leaf (`min_samples_leaf`).
3. **Fit the Model to Training Data**:
    - Fit the Random Forest classifier on the training data using the `.fit()` method.
4. **Model Prediction**:
    - Use the trained Random Forest model to make predictions on the test data using the `.predict()` method.
5. **Model Evaluation**:
    - Evaluate the model's performance using various metrics such as accuracy, confusion matrix, precision, recall, F1 score, etc.
    - You can also use cross-validation to better understand the model's performance.
6. **Hyperparameter Tuning**:
    - Optionally, tune the hyperparameters like the number of trees (`n_estimators`), tree depth (`max_depth`), or other parameters to improve the model’s performance. Techniques like Grid Search or Random Search are commonly used.

NAME: Rohan Bhimrao Patil **DATE:**

ROLL_NO: 150 **BATCH: B6**

PRACT_NAME: 6 classification using random forest.

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split
from sklearn.datasets import load_iris
from sklearn.metrics import accuracy_score, classification_report


data = load_iris()
X = data.data  # Features
y = data.target  # Target variable (class labels)

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

rf_classifier = RandomForestClassifier(n_estimators=100, random_state=42)


rf_classifier.fit(X_train, y_train)

y_pred = rf_classifier.predict(X_test)


accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy: {accuracy * 100:.2f}%")

print("Classification Report:\n", classification_report(y_test, y_pred))

print("Feature Importance:")
for feature, importance in zip(data.feature_names, rf_classifier.feature_importances_):
    print(f"{feature}: {importance:.4f}")
```

OUTPUT:

Accuracy: 100.00%
Classification Report:

	precision	recall	f1-score	support
0	1.00	1.00	1.00	19
1	1.00	1.00	1.00	13
2	1.00	1.00	1.00	13
accuracy			1.00	45
macro avg	1.00	1.00	1.00	45
weighted avg	1.00	1.00	1.00	45

Feature Importance:
sepal length (cm): 0.1041
sepal width (cm): 0.0446
petal length (cm): 0.4173
petal width (cm): 0.4340

7. Explanation of Regression Analysis using Linear Regression.

Linear Regression is one of the most basic and widely used techniques in statistics and machine learning for regression tasks. The goal of linear regression is to model the relationship between a dependent variable (target) and one or more independent variables (predictors) by fitting a linear equation to the observed data.

Linear Regression Overview:

1. **Simple Linear Regression**: Models the relationship between a single independent variable XXX and a dependent variable YYY.
   - The model is of the form: $Y=\beta0+\beta1\cdot X+\epsilon$ Where:
   - YYY is the dependent variable.
   - $\beta0$ is the intercept.
   - $\beta1$ is the coefficient of the independent variable.
   - ϵ is the error term (residuals).

2. **Multiple Linear Regression**: Extends the concept to multiple independent variables X1,X2,…,XnX_1, X_2, \dots, X_nX1,X2,…,Xn.
   - The model is of the form: $Y=\beta0+\beta1\cdot X1+\beta2\cdot X2+\cdots+\betan\cdot Xn+\epsilon$ Where:
   - YYY is the dependent variable.
   - X1,X2,…,XnX_1, X_2, \dots, X_nX1,X2,…,Xn are the independent variables.
   - $\beta0,\beta1,\ldots,\betan$ are the coefficients of the independent variables.
   - ϵ is the error term (residuals).

Steps for Regression Analysis using Linear Regression:

1. **Data Preparation**:
   - Collect and clean the data (handle missing values, categorical encoding, etc.).
   - Split the data into training and test sets (commonly 80-20 or 70-30 split).
2. **Model Training**:
   - Use the training set to fit the linear regression model using the `.fit()` method.
3. **Prediction**:
   - Use the trained model to make predictions on the test set using the `.predict()` method.
4. **Model Evaluation**:
   - Evaluate the model's performance using metrics such as Mean Squared Error (MSE), Mean Absolute Error (MAE), R2R^2R2 score, etc.
5. **Interpret the Results**:
   - Look at the coefficients to understand the influence of each predictor.
   - Check the R2R^2R2 value to assess the model's goodness of fit.

NAME: Rohan Bhimrao Patil **DATE:**

ROLL_NO: 150 **BATCH: B6**

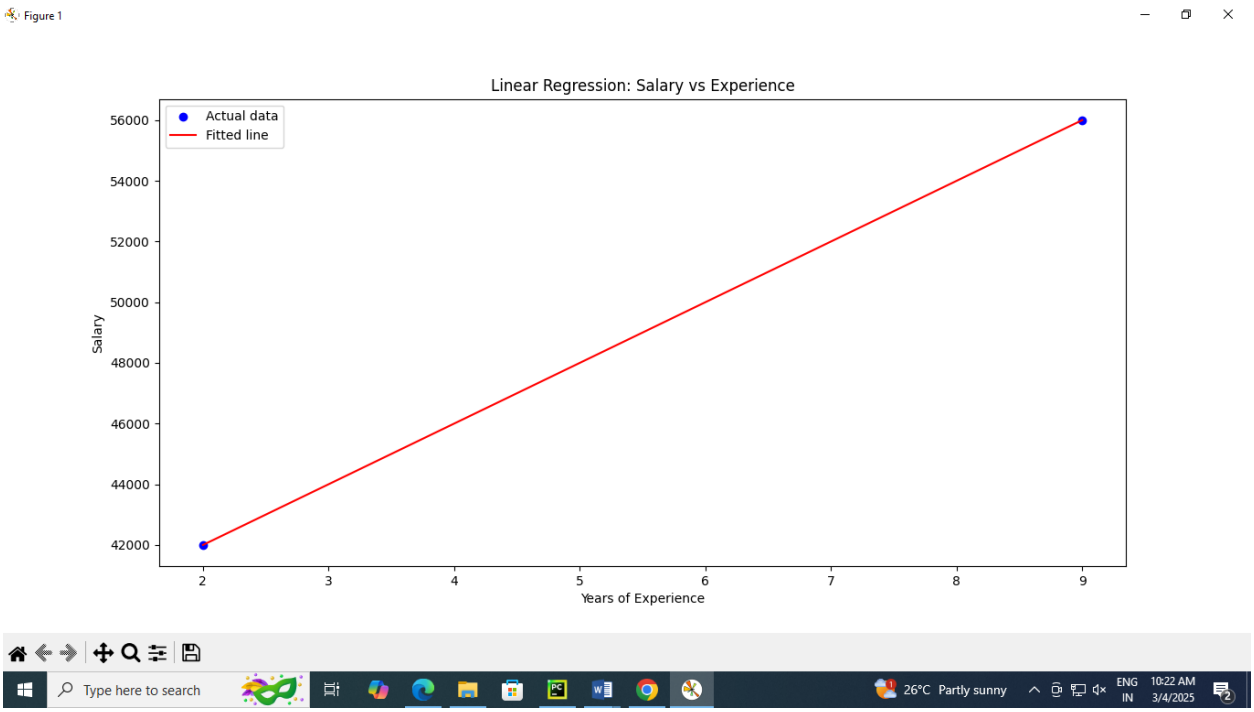
PRACT_NAME: 7 regression analysis using linear regression

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error, r2_score

data = {
   'YearsExperience': [1, 2, 3, 4, 5, 6, 7, 8, 9, 10],
   'Salary': [40000, 42000, 44000, 46000, 48000, 50000, 52000, 54000, 56000, 58000]
}

df = pd.DataFrame(data)


X = df[['YearsExperience']]  # Independent variable (Features)
y = df['Salary']  # Dependent variable (Target)


X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)


model = LinearRegression()
model.fit(X_train, y_train)


y_pred = model.predict(X_test)



mse = mean_squared_error(y_test, y_pred)
print(f"Mean Squared Error: {mse}")


r2 = r2_score(y_test, y_pred)
print(f"R-squared: {r2}")
```

```
plt.scatter(X_test, y_test, color='blue', label='Actual data')
plt.plot(X_test, y_pred, color='red', label='Fitted line')
plt.xlabel('Years of Experience')
plt.ylabel('Salary')
plt.title('Linear Regression: Salary vs Experience')
plt.legend()
plt.show()
```

```
new_data = np.array([[11]])  # Example: Predict salary for 11 years of experience
predicted_salary = model.predict(new_data)
print(f"Predicted Salary for 11 years of experience: ${predicted_salary[0]:,.2f}")
```

Output:



8. Explanation of Association Rule Mining using Apriori.

Association Rule Mining using Apriori

Association Rule Mining is a fundamental technique in data mining used to find interesting relationships (associations) between variables in large datasets. It's widely used in market basket analysis, where you discover patterns such as "If a customer buys item A, they are likely to buy item B."

The **Apriori Algorithm** is one of the most popular algorithms used for Association Rule Mining. It works by identifying frequent itemsets in a dataset and then deriving association rules from these itemsets. The rules are of the form:

- **Rule**: A→BA \rightarrow BA→B
  - Meaning: If item AAA is purchased, then item BBB is likely to be purchased as well.

Key Concepts in Association Rule Mining:

1. **Support**: The support of an itemset is the proportion of transactions in the database that contain the itemset.
   - Formula:
     Support(A)=Number of transactions containing ATotal number of transactions\text{Support}(A) = \frac{\text{Number of transactions containing A}}{\text{Total number of transactions}}Support(A)=Total number of transactionsNumber of transactions containing A
2. **Confidence**: Confidence is a measure of how often the rule has been found to be true.
   - Formula:
     Confidence(A→B)=Support(A∪B)Support(A)\text{Confidence}(A \rightarrow B) = \frac{\text{Support}(A \cup B)}{\text{Support}(A)}Confidence(A→B)=Support(A)Support(A∪B)
3. **Lift**: Lift is a measure of how much more likely item BBB is bought when item AAA is bought, compared to when BBB is bought without AAA.
   - Formula:
     Lift(A→B)=Support(A∪B)Support(A)·Support(B)\text{Lift}(A \rightarrow B) = \frac{\text{Support}(A \cup B)}{\text{Support}(A) \cdot \text{Support}(B)}Lift(A→B)=Support(A)·Support(B)Support(A∪B)
   - Lift > 1 indicates a strong association.

Steps in the Apriori Algorithm:

1. **Generate Candidate Itemsets**: Start by generating all possible itemsets of size 1 (i.e., individual items), then size 2, and so on.
2. **Prune Itemsets**: At each step, eliminate itemsets that have a support lower than the minimum threshold. Only retain itemsets that meet the minimum support.
3. **Generate Rules**: From the frequent itemsets, generate association rules by considering all possible combinations of item subsets, ensuring they meet the minimum confidence threshold.

NAME: Rohan Bhimrao Patil **DATE:**

ROLL_NO: 150 **BATCH: B6**

PRACT_NAME: 8 association rule mining using apriori.

```
import pandas as pd
from mlxtend.frequent_patterns import apriori, association_rules

data = {'Milk': [1, 1, 0, 1, 1],
      'Bread': [1, 1, 1, 1, 0],
      'Butter': [0, 1, 1, 1, 1],
      'Cheese': [1, 0, 1, 1, 1]}

df = pd.DataFrame(data)

frequent_itemsets = apriori(df, min_support=0.6, use_colnames=True)


rules = association_rules(frequent_itemsets, metric="confidence", min_threshold=0.7)


print("Frequent Itemsets:")
print(frequent_itemsets)


print("\nAssociation Rules:")
print(rules)
```

OUTPUT:

Frequent Itemsets:

	support	itemsets
0	0.8	(Milk)
1	0.8	(Bread)
2	0.8	(Butter)
3	0.8	(Cheese)
4	0.6	(Milk, Bread)
5	0.6	(Milk, Butter)
6	0.6	(Cheese, Milk)
7	0.6	(Butter, Bread)
8	0.6	(Cheese, Bread)
9	0.6	(Cheese, Butter)

Association Rules:

	antecedents	consequents	antecedent support	...	jaccard	certainty	kulczynski
0	(Milk)	(Bread)	0.8	...	0.6	-0.25	0.75
1	(Bread)	(Milk)	0.8	...	0.6	-0.25	0.75
2	(Milk)	(Butter)	0.8	...	0.6	-0.25	0.75
3	(Butter)	(Milk)	0.8	...	0.6	-0.25	0.75
4	(Cheese)	(Milk)	0.8	...	0.6	-0.25	0.75
5	(Milk)	(Cheese)	0.8	...	0.6	-0.25	0.75
6	(Butter)	(Bread)	0.8	...	0.6	-0.25	0.75
7	(Bread)	(Butter)	0.8	...	0.6	-0.25	0.75
8	(Cheese)	(Bread)	0.8	...	0.6	-0.25	0.75
9	(Bread)	(Cheese)	0.8	...	0.6	-0.25	0.75
10	(Cheese)	(Butter)	0.8	...	0.6	-0.25	0.75
11	(Butter)	(Cheese)	0.8	...	0.6	-0.25	0.75

[12 rows x 14 columns]

9. Visualize the result of the clustering and compare.

Visualizing the results of clustering is an essential step in understanding the structure of the data and assessing the performance of the clustering algorithm. In this case, we can use **K-Means clustering** as an example, but the approach can be applied to other clustering algorithms like **DBSCAN**, **Agglomerative Clustering**, etc.

To visualize the clustering results and compare different clustering algorithms, we can follow these steps:

1. K-Means Clustering Example

Let's first demonstrate how to perform K-Means clustering on a dataset and visualize the results. Afterward, we'll compare it to other clustering techniques (e.g., Agglomerative Clustering or DBSCAN).

Steps to Visualize Clustering:

1. **Generate or load a dataset**: For simplicity, we'll generate a synthetic dataset (e.g., `make_blobs`).
2. **Perform K-Means clustering**.
3. **Visualize the clusters**.
4. **Compare the results**: We'll compare K-Means with another clustering algorithm, such as **Agglomerative Clustering**.

Explanation:

1. **Dataset**: We use `make_blobs` to generate a synthetic dataset with 300 samples and 4 centers (clusters).
2. **Standardization**: Standardize the dataset to zero mean and unit variance using `StandardScaler`. This is important because many clustering algorithms, including K-Means, perform better when the data is scaled.
3. **K-Means Clustering**: We perform K-Means clustering with 4 clusters.
4. **Agglomerative Clustering**: We perform Agglomerative Clustering (a hierarchical clustering method) to compare with K-Means.
5. **Visualization**: We visualize the clustering results for both K-Means and Agglomerative Clustering using scatter plots.

NAME: Rohan Bhimrao Patil **DATE:**

ROLL_NO: 150 **BATCH: B6**

Practical No 9 :- Visualize the result of the clustering and compare.

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

# Generate synthetic dataset
X, y = make_blobs(n_samples=300, centers=3, cluster_std=1.0, random_state=42)

# Apply K-Means clustering
kmeans = KMeans(n_clusters=3, random_state=42)
kmeans_labels = kmeans.fit_predict(X)

# Apply DBSCAN clustering
dbscan = DBSCAN(eps=0.8, min_samples=5)
dbscan_labels = dbscan.fit_predict(X)

# Plot the results
fig, axes = plt.subplots(1, 2, figsize=(12, 5))

# K-Means Clustering
axes[0].scatter(X[:, 0], X[:, 1], c=kmeans_labels, cmap='viridis', marker='o', edgecolor='k')
axes[0].scatter(kmeans.cluster_centers_[:, 0], kmeans.cluster_centers_[:, 1], c='red', marker='X', s=200,
label="Centroids")
axes[0].set_title("K-Means Clustering")
axes[0].legend()

# DBSCAN Clustering
axes[1].scatter(X[:, 0], X[:, 1], c=dbscan_labels, cmap='viridis', marker='o', edgecolor='k')
axes[1].set_title("DBSCAN Clustering")

plt.show()
```

Output:

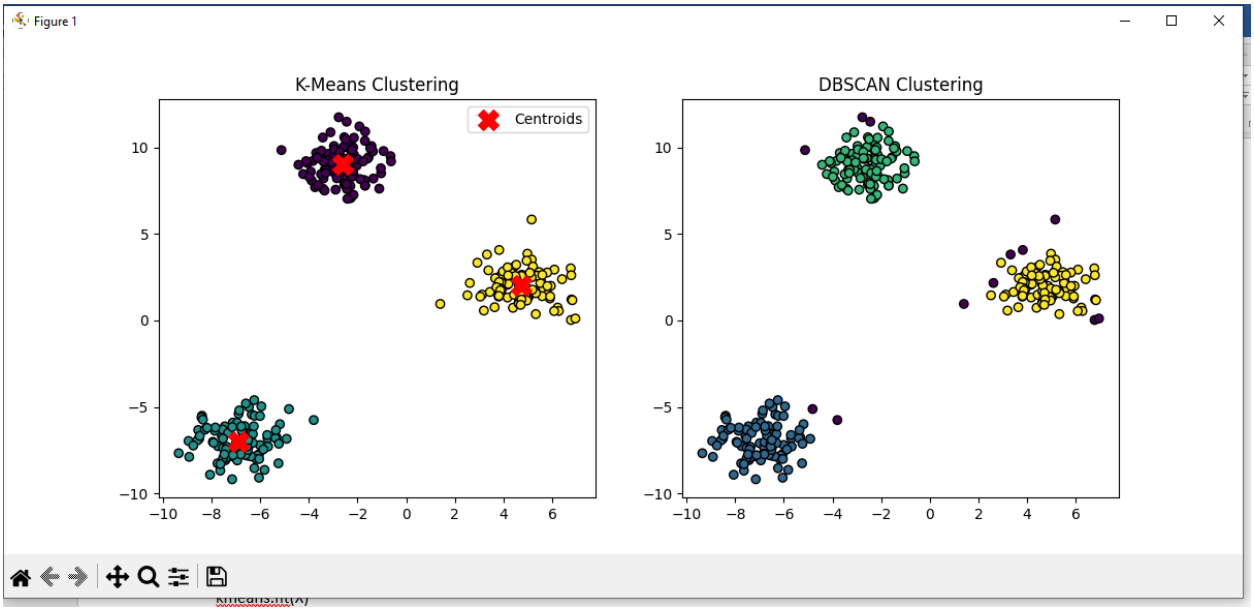


Figure 1
K-Means Clustering
Centroids
DBSCAN Clustering
10
5
0
−5
−10
−10
−8
−6
−4
−2
0
2
4
6

10. Explanation of Visualize the correlation matrix using a pseudocolor plot.

Visualizing the correlation matrix using a pseudocolor plot is an effective way to understand the relationships between different variables in a dataset. A correlation matrix shows the pairwise correlations between features, and using a pseudocolor plot (like a heatmap) can provide an intuitive view of these relationships.

Steps to Visualize the Correlation Matrix Using a Pseudocolor Plot:

1. **Compute the correlation matrix**: Calculate the correlation coefficients between all pairs of features in the dataset.
2. **Plot the correlation matrix**: Use a heatmap (pseudocolor plot) to visualize the correlation matrix.

Correlation Coefficient:

- **Pearson's correlation coefficient** is most commonly used. It measures the linear relationship between two variables.
  - Values range from -1 to 1:
  - **1**: Perfect positive correlation.
  - **-1**: Perfect negative correlation.
  - **0**: No linear relationship.

Advantages of Visualizing the Correlation Matrix:

- **Quick Insights**: Heatmaps give an immediate understanding of which features are highly correlated, which can be useful for feature selection, dimensionality reduction (e.g., PCA), or identifying multicollinearity.
- **Data Exploration**: It helps in exploring relationships between variables, particularly for understanding dependencies.

NAME: Rohan Bhimrao Patil **DATE:**

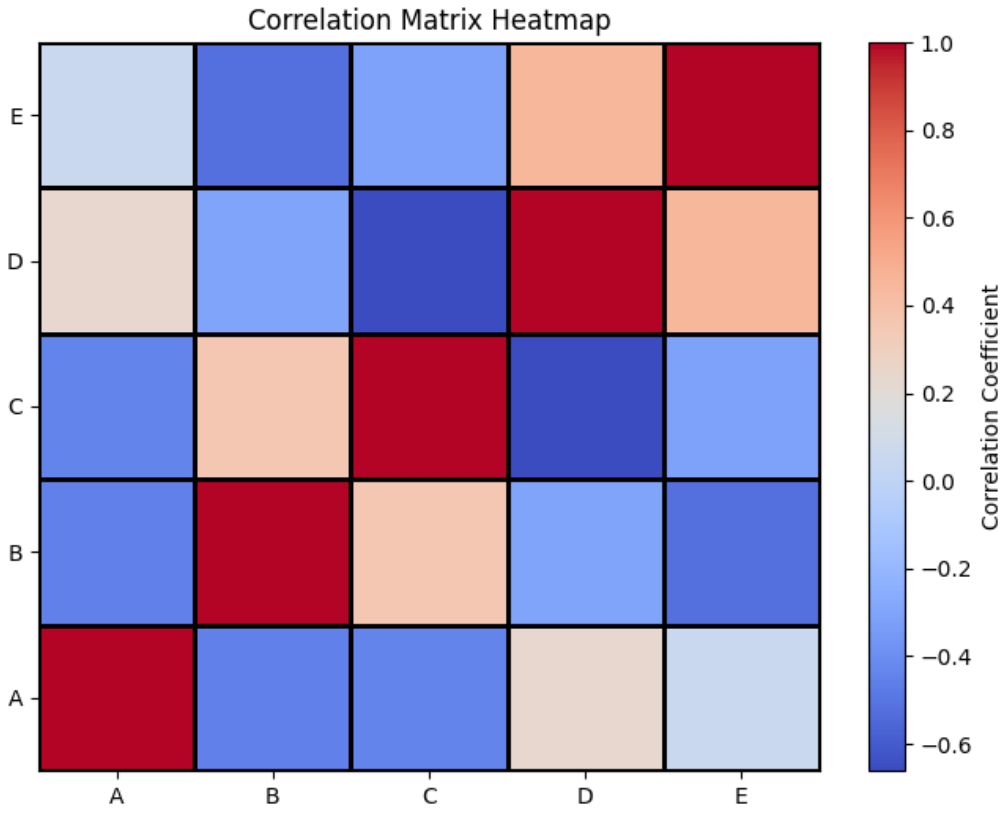
ROLL_NO: 150 **BATCH: B6**

Practical No 10:- Visualize the correlation matrix using a pseudo color plot

---

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

# Generate a random dataset
np.random.seed(42)
data = pd.DataFrame(np.random.rand(10, 5), columns=['A', 'B', 'C', 'D', 'E'])

# Compute the correlation matrix
corr_matrix = data.corr()

# Create a pseudocolor plot (heatmap)
plt.figure(figsize=(8, 6))
plt.pcolormesh(corr_matrix, cmap='coolwarm', edgecolors='k')
plt.colorbar(label='Correlation Coefficient')

# Add labels at the center of each grid
plt.xticks(np.arange(0.5, len(corr_matrix.columns), 1), corr_matrix.columns)
plt.yticks(np.arange(0.5, len(corr_matrix.index), 1), corr_matrix.index)
plt.title('Correlation Matrix Heatmap')

plt.show()
```

Output:


Correlation Matrix Heatmap
E
D
C
B
A
A
B
C
D
E
1.0
0.8
0.6
0.4
0.2
0.0
−0.2
−0.4
−0.6
Correlation Coefficient

11. Explanation of Use of degrees distribution of a network.

Use of Degree Distribution in a Network

The **degree distribution** of a network is a key concept in network theory, particularly in the study of graph structures. It describes the frequency of each possible degree (number of connections or edges) in the network. In a graph, the degree of a node is simply the number of edges connected to it.

Degree distribution helps us understand the structure of the network, revealing whether it has properties such as:

1. **Scale-free behavior**: A few nodes (hubs) have very high degrees, while the majority have low degrees. This is typical of real-world networks like the internet or social networks.
2. **Randomness**: If the degree distribution follows a Poisson distribution, the network is more likely to be random.
3. **Community structure**: In networks with communities, degree distribution can help detect whether nodes in different communities have different degrees.

Key Terms**:**

1. **Node Degree**: The number of edges connected to a node.
2. **Degree Distribution**: The probability distribution of node degrees in a graph.
3. **Average Degree**: The average number of edges per node.

Steps to Calculate Degree Distribution**:**

1. **Calculate the degree of each node**: This can be done by simply counting the number of edges incident to each node.
2. **Plot the degree distribution**: Typically, a histogram is used to show the frequency of each degree.

Python Example: Visualizing Degree Distribution

We will use the **NetworkX** library in Python, which is specifically designed for creating and analyzing complex networks, and **Matplotlib** for plotting.

Steps:

1. **Generate or load a network** (e.g., random graph, social network, etc.).
2. **Calculate the degree of each node**.
3. **Plot the degree distribution**.

NAME: Rohan Bhimrao Patil **DATE:**

ROLL_NO: 150 **BATCH: B6**

PRACT_NAME:11 use of degree distribution of a network.

```
import networkx as nx
import matplotlib.pyplot as plt

G = nx.erdos_renyi_graph(n=100, p=0.05)  # A random graph with 100 nodes and a 5% chance of edge creation

degree_sequence = [G.degree(node) for node in G.nodes()]

degree_count = {}
for degree in degree_sequence:
   degree_count[degree] = degree_count.get(degree, 0) + 1

degrees = list(degree_count.keys())  # List of degrees
counts = list(degree_count.values())  # Corresponding counts of nodes with each degree


plt.figure(figsize=(8, 6))
plt.bar(degrees, counts, color='b')
plt.xlabel('Degree (k)')
plt.ylabel('Number of Nodes (Count)')
plt.title('Degree Distribution of the Network')
plt.show()

plt.figure(figsize=(8, 6))
plt.loglog(degrees, counts, marker='o', color='r')
plt.xlabel('Log(Degree)')
plt.ylabel('Log(Count)')
plt.title('Log-Log Degree Distribution')
plt.show()
```

OUTPUT:

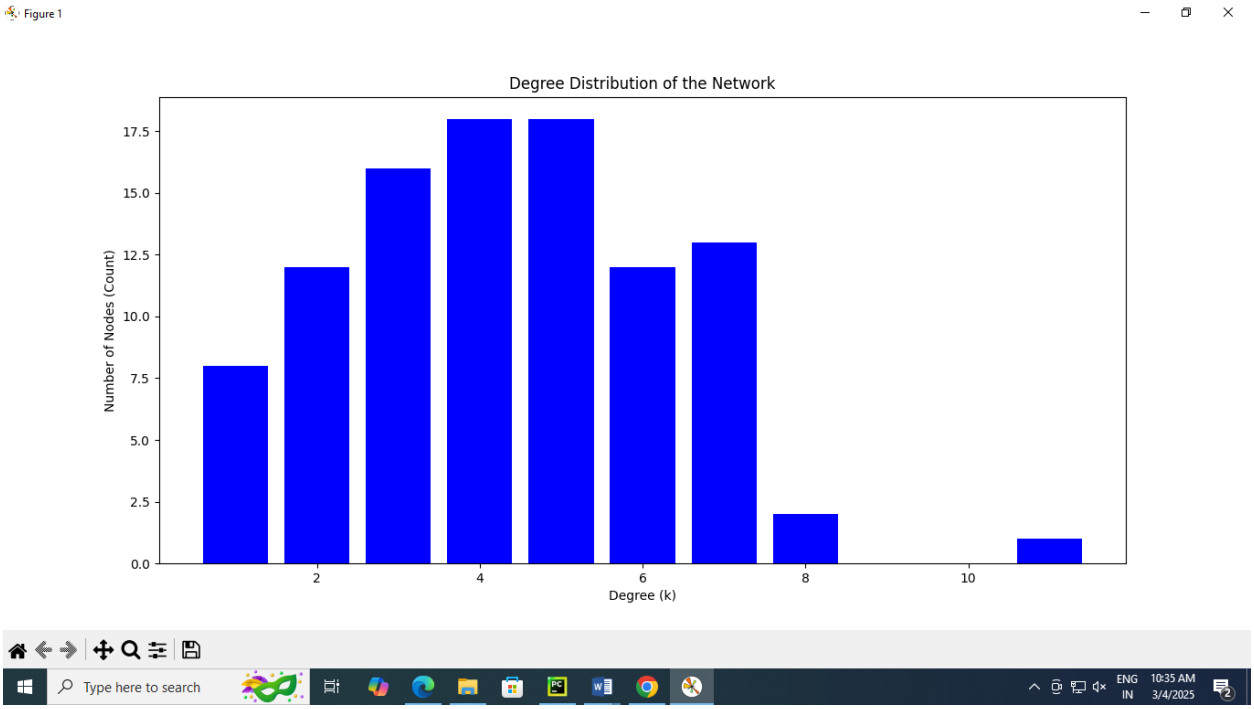


Figure 1
Degree Distribution of the Network
Number of Nodes (Count)
Degree (k)
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
2
4
6
8
10
Type here to search
ENG
IN
10:35 AM
3/4/2025

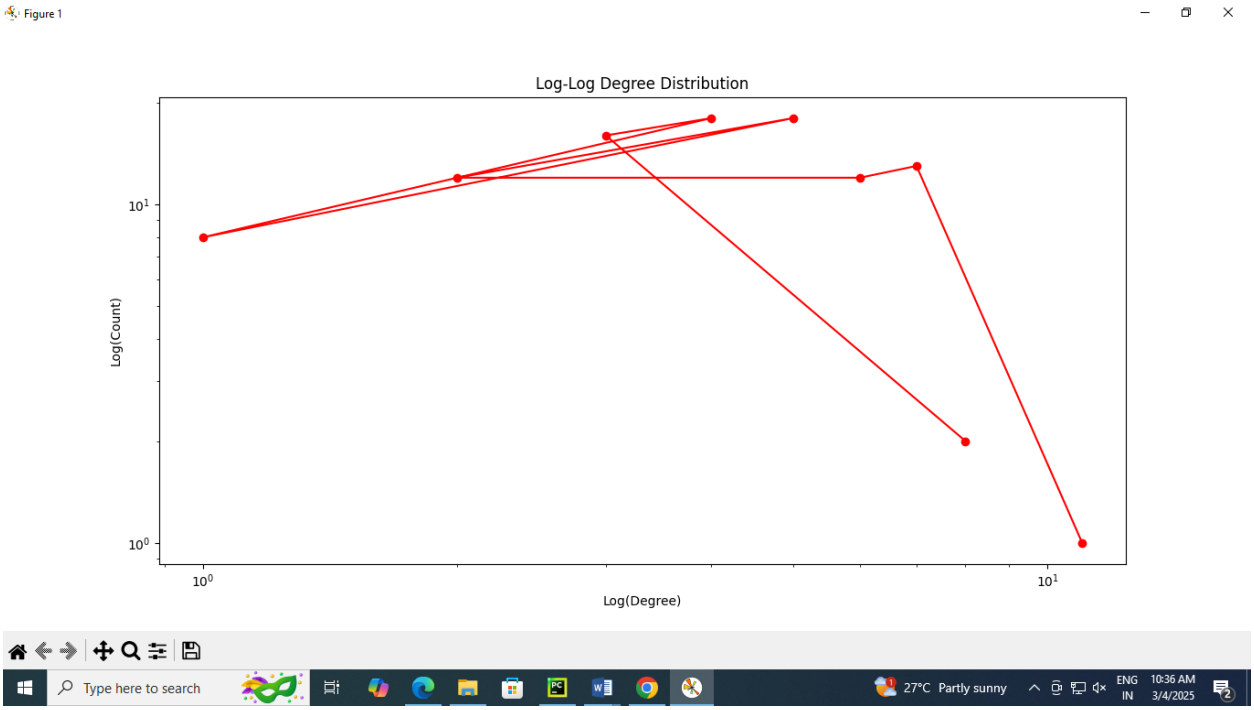


Figure 1
Log-Log Degree Distribution
Log(Count)
Log(Degree)
10^0
10^1
10^0
10^1
Type here to search
27°C Partly sunny
ENG
IN
10:36 AM
3/4/2025

12. Explanation of Graph visualization of a network using maximum, minimum, median, first quartile and third quartile.

Graph Visualization of a Network Using Degree Statistics

When analyzing a network, it's useful to visualize nodes based on statistical properties of their degrees, such as the **maximum**, **minimum**, **median**, **first quartile (Q1)**, and **third quartile (Q3)**. These statistics can help in understanding how nodes are distributed in terms of their connectivity.

For this task, we will:

1. **Calculate the degree** of each node in the network.
2. **Determine degree statistics**: Minimum, maximum, median, Q1 (first quartile), and Q3 (third quartile).
3. **Visualize the network** by adjusting the node size or color based on these degree statistics.

Steps:

1. **Create a network**: You can use any graph, but we'll use a random graph for illustration.
2. **Calculate the degree of each node**: This gives us how many edges are connected to each node.
3. **Compute the degree statistics**: Minimum, maximum, median, Q1, and Q3 of node degrees.
4. **Visualize the network**: Adjust node properties (e.g., size, color) based on the degree statistics.

Use Cases:

- **Network Analysis**: By visualizing the network with degree statistics, you can get a sense of which nodes are central (hubs) and which are peripheral (isolated).
- **Community Detection**: Understanding node degrees helps in detecting potential community structures within the network.
- **Identifying Critical Nodes**: Nodes with high degrees may represent key components of the network and are often targeted in robustness analysis or optimization tasks.

Additional Customizations:

- **Color**: You can modify the node colors based on degree or other centrality measures (e.g., betweenness centrality).
- **Edge Weight**: If the network has weighted edges, you can adjust the width or color of the edges based on the edge weights.

NAME: Rohan Bhimrao Patil **DATE:**

ROLL_NO: 150 **BATCH: B6**

PRACT_NAME: 12.graph visualization of a network using maximum,minimum,median,first quartile,and third quartile.

```
import networkx as nx
import matplotlib.pyplot as plt
import numpy as np

G = nx.erdos_renyi_graph(n=100, p=0.1)  # A random graph with 100 nodes and a 10% chance of edge creation


degree_sequence = [G.degree(node) for node in G.nodes()]


max_degree = np.max(degree_sequence)
min_degree = np.min(degree_sequence)
median_degree = np.median(degree_sequence)
q1 = np.percentile(degree_sequence, 25)
q3 = np.percentile(degree_sequence, 75)

print(f"Maximum Degree: {max_degree}")
print(f"Minimum Degree: {min_degree}")
print(f"Median Degree: {median_degree}")
print(f"First Quartile (Q1): {q1}")
print(f"Third Quartile (Q3): {q3}")

node_colors = []
for degree in degree_sequence:
    if degree == max_degree:
        node_colors.append('red')  # High-degree node (maximum degree)
    elif degree == min_degree:
        node_colors.append('blue')  # Low-degree node (minimum degree)
    elif degree <= q1:
        node_colors.append('green')  # Nodes in the first quartile
    elif degree <= median_degree:
        node_colors.append('yellow')  # Nodes up to the median degree
    elif degree <= q3:
        node_colors.append('orange')  # Nodes up to the third quartile
    else:
        node_colors.append('purple')  # Nodes greater than the third quartile


plt.figure(figsize=(10, 8))
```

```
pos = nx.spring_layout(G, seed=42)  # Positions for all nodes
nx.draw(G, pos, with_labels=True, node_size=300, node_color=node_colors, font_size=10,
font_weight='bold', edge_color='gray')


import matplotlib.lines as mlines

max_label = mlines.Line2D([], [], color='red', marker='o', markersize=10, label=f"Max Degree
({max_degree})")
min_label = mlines.Line2D([], [], color='blue', marker='o', markersize=10, label=f"Min Degree
({min_degree})")
q1_label = mlines.Line2D([], [], color='green', marker='o', markersize=10, label=f"Q1 (≤ {q1})")
median_label = mlines.Line2D([], [], color='yellow', marker='o', markersize=10, label=f"Median (≤
{median_degree})")
q3_label = mlines.Line2D([], [], color='orange', marker='o', markersize=10, label=f"Q3 (≤ {q3})")
above_q3_label = mlines.Line2D([], [], color='purple', marker='o', markersize=10, label=f"Above Q3")

plt.legend(handles=[max_label, min_label, q1_label, median_label, q3_label, above_q3_label],
loc="upper left")


plt.title("Network Visualization with Degree Statistics Coloring")
plt.show()
```

OUTPUT:

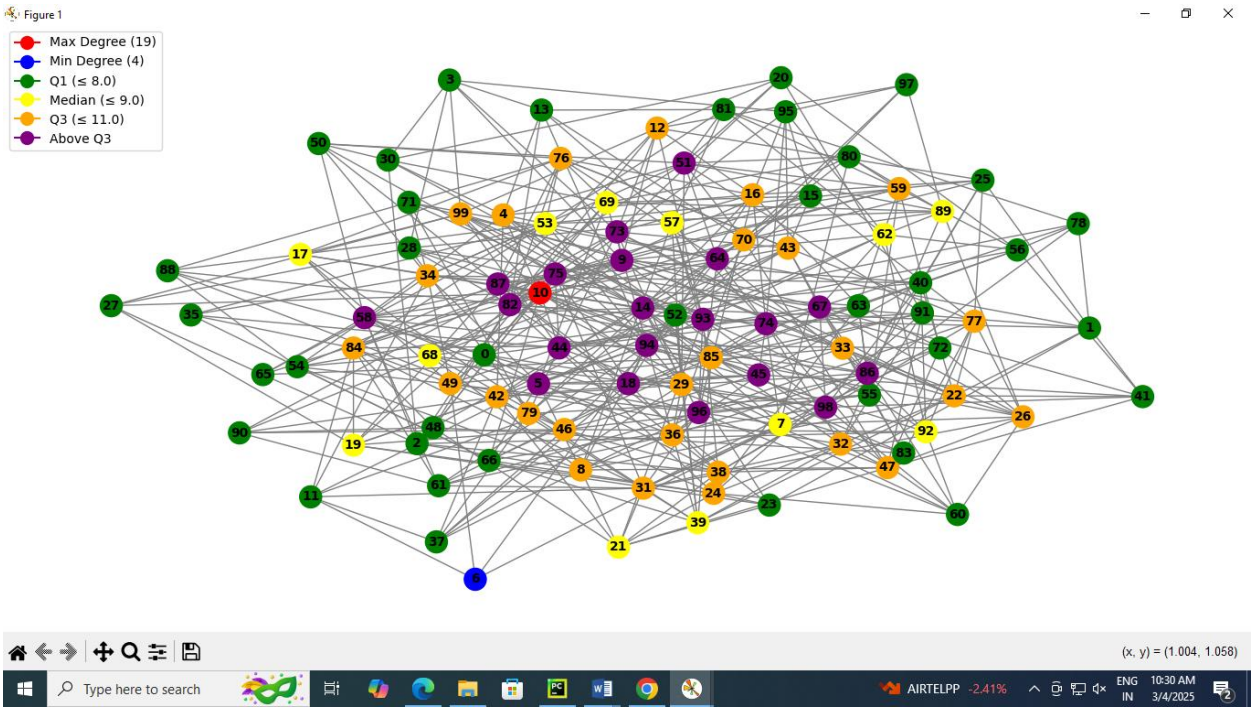


Figure 1
Max Degree (19)
Min Degree (4)
Q1 (≤ 8.0)
Median (≤ 9.0)
Q3 (≤ 11.0)
Above Q3
(x, y) = (1.004, 1.058)
Type here to search
AIRTELPP -2.41%
ENG
IN
10:30 AM
3/4/2025