DIMPLE DALWANI D15C 8

DS Lab 7

Aim: To implement different clustering algorithms.

Theory:

1- K-Means Algorithm (Centroid-Based Clustering)

Goal: Partition data into *k* clusters by minimizing variance within each cluster.

Steps:

1. Initialize
    - Choose the number of clusters (k).
    - Randomly select *k* data points as initial centroids.
2. Assign Points to Nearest Centroid
    - Compute the Euclidean distance between each data point and the centroids.
    - Assign each point to the closest centroid.
3. Update Centroids
    - Compute the new centroid of each cluster (mean of all points in that cluster).
4. Repeat Until Convergence
    - Repeat Steps 2 & 3 until centroids stop changing (or max iterations reached).

Advantages

- Works well for compact, spherical clusters.
- Fast and efficient for large datasets.

Disadvantages

- Needs *k* to be predefined.
- Sensitive to outliers.

2- DBSCAN Algorithm (Density-Based Clustering)

Goal: Detect clusters based on high-density regions and mark noise points.

Steps:

1. Set Parameters:
    - eps → Maximum distance to consider a point as a neighbor.
    - min_samples → Minimum points required to form a cluster. 2.

    Select a Random Point
        - If it has at least min_samples neighbors, it becomes a core point.
        - Otherwise, it is labeled noise (temporary).
3. Expand Cluster
        - Find all density-reachable points from the core point.
        - Assign them to the same cluster.
4. Repeat for All Points

    - If a noise point later becomes reachable from a core point, it joins a cluster.

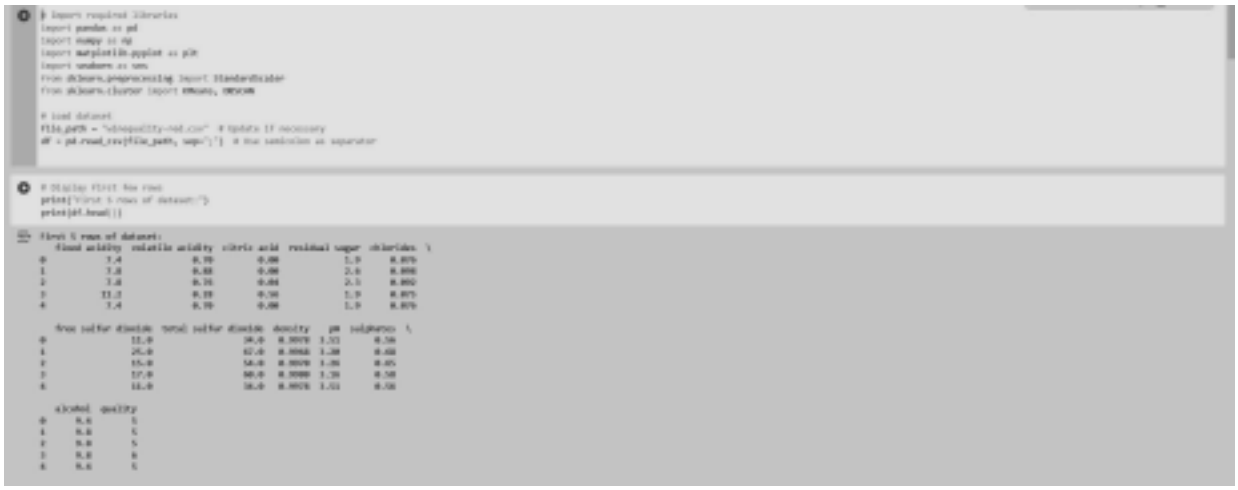
Advantages

- No need for k (finds clusters automatically).
- Can detect arbitrary shaped clusters.
- Handles outliers well.

Disadvantages

- Sensitive to eps and min_samples.
- Struggles in varying density datasets.

1-



This imports essential libraries like pandas, numpy, matplotlib, seaborn, and sklearn for clustering analysis. It loads the **wine quality dataset** from a CSV file using pd.read_csv(), specifying a semicolon (;) as the separator. The dataset consists of chemical properties of wine, such as acidity, sugar content, and alcohol percentage. The

df.head() function displays the first five rows to get an overview of the data structure. Key clustering methods, **K-Means** and **DBSCAN**, are also imported for further analysis.

2-

This checks for missing values in a **wine quality dataset** and ensures data integrity before further analysis. It first prints the column names to verify the dataset structure. Then, it checks for missing values using df.isnull().sum(), revealing that all columns have zero missing values. Despite this, the df.dropna() function is used as a precaution to remove any potential missing rows. Since no missing values exist, the dataset remains unchanged. This step ensures clean data for further processing, such as feature scaling or model training.

3-

This prepares data for clustering by selecting two numerical features, **"alcohol"** and **"density"**, from the dataset. These features are stored in a new DataFrame data to ensure relevant attributes are used. Next, **StandardScaler()** is applied to normalize the data, improving clustering performance by standardizing feature values to have a mean of **zero** and a standard deviation of **one**. The transformed data, stored in data_scaled, ensures that both features contribute equally during clustering. This step is crucial for distance-based clustering methods like K-Means.

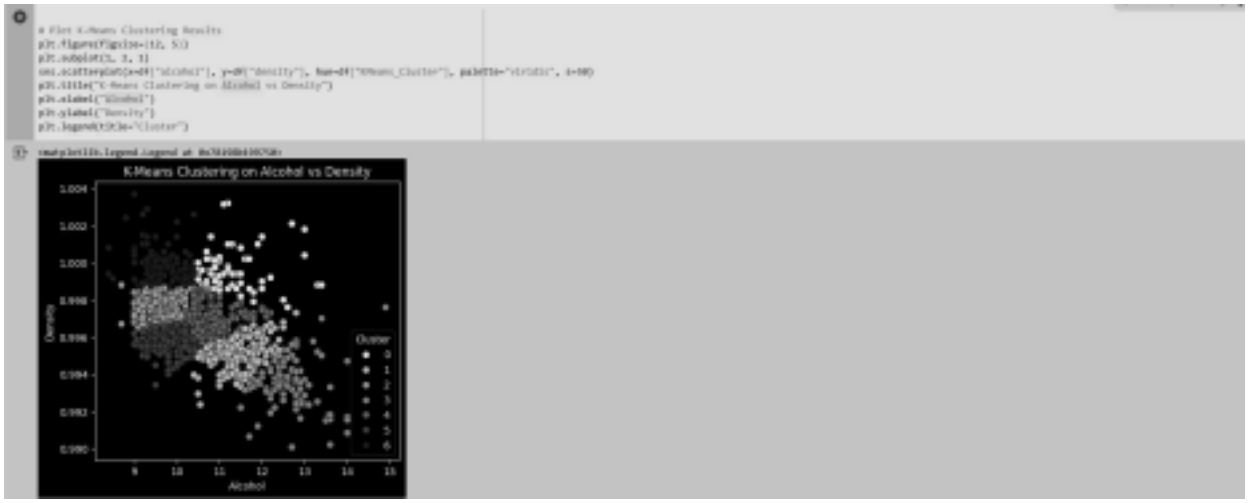
4-

```
[ ] # Apply K-Means clustering
    kmeans = KMeans(n_clusters=7, random_state=42, n_init=10)
    df["KMeans_Cluster"] = kmeans.fit_predict(data_scaled)

[ ] # Apply DBSCAN clustering
    dbscan = DBSCAN(eps=0.5, min_samples=7)
    df["DBSCAN_Cluster"] = dbscan.fit_predict(data_scaled)
```

This applies two clustering algorithms, **K-Means** and **DBSCAN**, to the scaled dataset. First, **K-Means** is used with **7 clusters**, a fixed random state for reproducibility, and **10 initializations** to ensure better clustering. The predicted cluster labels are stored in the "KMeans_Cluster" column of the DataFrame. Next, **DBSCAN** is applied with an **epsilon (eps) of 0.5** and a minimum of **7 samples per cluster**, identifying dense regions in the data. The DBSCAN cluster labels are stored in the "DBSCAN_Cluster" column, allowing comparison between the two methods.

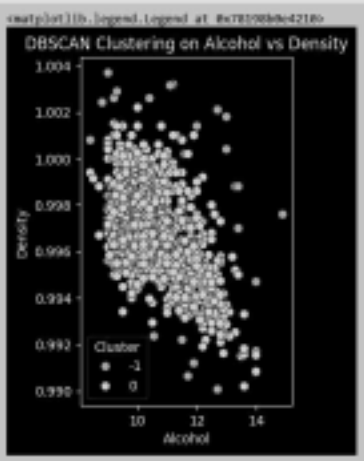
5-



The image shows a scatter plot depicting the results of K-Means clustering applied to a dataset with "Alcohol" on the x-axis and "Density" on the y-axis. Each point represents a data sample, color-coded according to its assigned cluster using the "viridis" color palette. The clustering algorithm has identified seven distinct groups (clusters labeled 0-6), as indicated in the legend. The distribution of points suggests that samples with similar Alcohol and Density values are grouped together, forming clear patterns. The plot provides insights into how these two features influence clustering, revealing regions of high and low density for different alcohol levels.

6-

```
# Plot DBSCAN Clustering Results
plt.subplot(1, 2, 2)
sns.scatterplot(x=df["alcohol"], y=df["density"], hue=df["DBSCAN_Cluster"], palette="coolwarm", s=60)
plt.title("DBSCAN Clustering on Alcohol vs Density")
plt.xlabel("Alcohol")
plt.ylabel("Density")
plt.legend(title="Cluster")
```



This applies DBSCAN clustering to a dataset and visualizes the results in a scatter plot, where "Alcohol" is on the x-axis and "Density" is on the y-axis. The points are colored based on their cluster assignments, using the "coolwarm" palette. In the resulting plot, most points belong to Cluster 0 (red), while some outliers (noise points) are assigned to Cluster -1 (blue). Unlike K-Means, DBSCAN effectively identifies dense regions and marks sparse points as noise. The visualization highlights how DBSCAN clusters high-density regions while leaving low-density points unclassified. This helps in detecting outliers and meaningful patterns in the data.

7-

```
# Show plots
plt.tight_layout()
plt.show()
```

```
<Figure size 640x480 with 0 Axes>
```

```
# Display cluster counts
print("\nK-Means Cluster Counts:")
print(df["KMeans_Cluster"].value_counts())
```

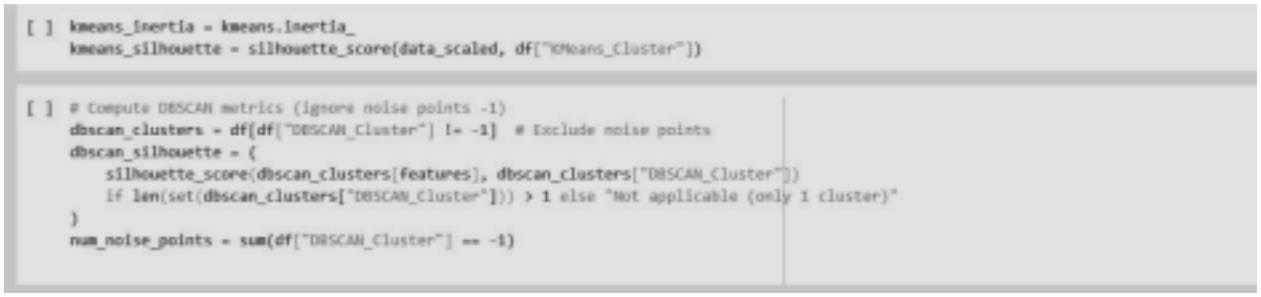
```
K-Means Cluster Counts:
KMeans_Cluster
2    437
5    301
1    269
4    242
6    147
3    123
0     80
Name: count, dtype: int64
```

```
print("\nDBSCAN Cluster Counts:")
print(df["DBSCAN_Cluster"].value_counts())
```

```
DBSCAN Cluster Counts:
DBSCAN_Cluster
 0    1578
-1      29
Name: count, dtype: int64
```

This shows the results of two clustering algorithms applied to a dataset: K-Means and DBSCAN. The K-Means algorithm divided the data into 7 clusters (labeled 0 to 6), with cluster 2 having the most points (457) and cluster 0 the fewest (80). In contrast, DBSCAN identified only two groups: cluster 0 with 1570 points and cluster -1 with 29 points, where -1 represents noise or outliers. This suggests K-Means created more granular groupings, while DBSCAN categorized most data into a single cluster, flagging a small portion as anomalies. The tight_layout() and show() commands indicate the results were visualized, though the plot itself is not displayed here. The output highlights the differing behaviors of the algorithms in handling the dataset.

8-



```
[ ] kmeans_inertia = kmeans.inertia_
    kmeans_silhouette = silhouette_score(data_scaled, df["KMeans_Cluster"])

[ ] # Compute DBSCAN metrics (ignore noise points -1)
    dbscan_clusters = df[df["DBSCAN_Cluster"] != -1]  # Exclude noise points
    dbscan_silhouette = (
        silhouette_score(dbscan_clusters[features], dbscan_clusters["DBSCAN_Cluster"])
        if len(set(dbscan_clusters["DBSCAN_Cluster"])) > 1 else "Not applicable (only 1 cluster)"
    )
    num_noise_points = sum(df["DBSCAN_Cluster"] == -1)
```

This is for evaluating the performance of K-Means and DBSCAN clustering algorithms. For K-Means, the inertia(sum of squared distances of samples to their nearest cluster center) and silhouette score (measuring cluster cohesion and separation) are computed directly using the scaled data and cluster labels. For DBSCAN, noise points (labeled `-1`) are excluded before calculating the silhouette score, which is only computed if there are at least two clusters; otherwise, it returns "Not applicable." The code also counts the number of noise points identified by DBSCAN. The differences in evaluation methods highlight K-Means' reliance on predefined clusters and DBSCAN's noise-handling capability, with metrics tailored to each algorithm's unique characteristics. A syntax error (missing parenthesis) is also visible in the DBSCAN silhouette score calculation.

9-

```
[ ]  # Print cluster evaluation metrics
     print("\n🔹 **Cluster Evaluation Metrics** 🔹")
     print(f"🔹 K-Means Inertia (WCSS): {kmeans_inertia:.2f}")
     print(f"🔹 K-Means Silhouette Score: {kmeans_silhouette:.4f}")
     print(f"🔹 DBSCAN Silhouette Score: {dbscan_silhouette}")
     print(f"🔹 DBSCAN Noise Points: {num_noise_points}")
```

```
🔹 **Cluster Evaluation Metrics** 🔹
🔹 K-Means Inertia (WCSS): 532.73
🔹 K-Means Silhouette Score: 0.3575
🔹 DBSCAN Silhouette Score: Not applicable (only 1 cluster)
🔹 DBSCAN Noise Points: 29
```

This displays clustering evaluation metrics for K-Means and DBSCAN. K-Means has an inertia (sum of squared distances) of **532.73** and a silhouette score of **0.3575**, indicating moderate cluster separation. DBSCAN's silhouette score is **not applicable** because it formed only one cluster (excluding noise), with **29 points** labeled as noise. The results show K-Means performed structured clustering, while DBSCAN treated most data as a single group with outliers. The metrics highlight the algorithms' differing behaviors on the dataset.

Conclusion

K-Means is an efficient clustering algorithm that assigns data points into a predefined number of clusters based on centroid minimization. It performs well with well-separated and spherical clusters but struggles with arbitrary shapes.. DBSCAN, on the other hand, is density-based and can identify noise points, making it more robust for complex cluster structures. However, DBSCAN is sensitive to parameter tuning (**eps, min_samples**) and may label some points as noise instead of assigning them to clusters. In this experiment, K-Means produced balanced clusters, while DBSCAN identified noise points but captured non-linear structures better. The choice between these algorithms depends on the dataset characteristics and the need for predefined clusters versus flexible, shape-adaptive clustering.