JADAVPUR UNIVERSITY

INFORMATION TECHNOLOGY

ML ASSIGMENT 4

NAME : SURAJ ROY

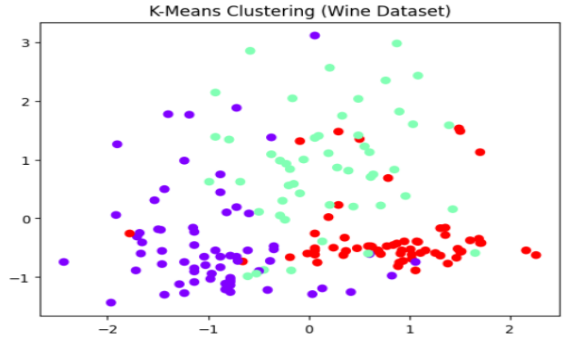
ROLL : 002211001129

GROUP : A3 CLASS : FOURTH YEAR

```python
# ========================
# K-Means Clustering on Iris & Wine Dataset
# ========================

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.datasets import load_iris, load_wine
from sklearn.cluster import KMeans
from sklearn.metrics import (
    rand_score, adjusted_rand_score,
    mutual_info_score, adjusted_mutual_info_score,
    normalized_mutual_info_score,
    silhouette_score, calinski_harabasz_score, davies_bouldin_score
)
from sklearn.preprocessing import StandardScaler

# ---------------------------------------
# Function for performance evaluation
# ---------------------------------------
def evaluate_clustering(X, labels_true, labels_pred):
    print("\nClustering Performance Scores:")
    print("-----------------------------------------------")
    print("Rand Score:", rand_score(labels_true, labels_pred))
    print("Adjusted Rand Score:", adjusted_rand_score(labels_true, labels_pred))
    print("Mutual Info Score:", mutual_info_score(labels_true, labels_pred))
    print("Adjusted Mutual Info Score:", adjusted_mutual_info_score(labels_true, labels_pred))
    print("Normalized Mutual Info Score:", normalized_mutual_info_score(labels_true, labels_pred))
    print("-----------------------------------------------")
    print("Silhouette Coefficient:", silhouette_score(X, labels_pred))
    print("Calinski-Harabasz Index:", calinski_harabasz_score(X, labels_pred))
    print("Davies-Bouldin Index:", davies_bouldin_score(X, labels_pred))

    # SSE (Cohesion)
    sse = np.sum((X - KMeans(n_clusters=len(np.unique(labels_pred))).fit(X).cluster_centers_[labels_pred]) ** 2)

    # SSB (Separation)
    overall_mean = np.mean(X, axis=0)
    clusters = np.unique(labels_pred)
    ssb = np.sum([len(X[labels_pred == c]) * np.sum((KMeans(n_clusters=len(clusters)).fit(X).cluster_centers_[c] - overall_mean) ** 2)
                  for c in clusters])

    print("-----------------------------------------------")
    print("Cohesion (SSE):", sse)
    print("Separation (SSB):", ssb)
    print("-----------------------------------------------\n")
```

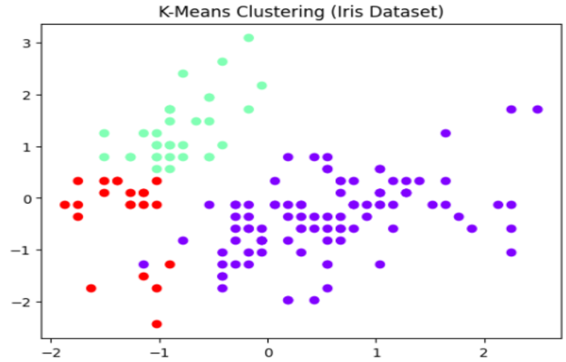
```python
# ---------------------------------------
# Function to run K-Means
# ---------------------------------------
def run_kmeans(data_name, X, y):
    print(f"\n========== K-Means Clustering on {data_name} Dataset ==========")

    # Scale the data
    X_scaled = StandardScaler().fit_transform(X)

    # Apply K-Means
    kmeans = KMeans(n_clusters=len(np.unique(y)), init='k-means++', random_state=42)
    kmeans.fit(X_scaled)

    print("\nK-Means Cluster Centers:")
    print(kmeans.cluster_centers_)
    print("\nCluster Labels:")
    print(kmeans.labels_)

    # Visualization (like demo screenshot)
    plt.scatter(X_scaled[:, 0], X_scaled[:, 1], c=kmeans.labels_, cmap='rainbow')
    plt.title(f'K-Means Clustering ({data_name} Dataset)')
    plt.show()

    # Evaluate
    evaluate_clustering(X_scaled, y, kmeans.labels_)


# ---------------------------------------
# Load and Run for both datasets
# ---------------------------------------

# Iris
iris = load_iris()
run_kmeans("Iris", iris.data, iris.target)

# Wine
wine = load_wine()
run_kmeans("Wine", wine.data, wine.target)
```

```
========== K-Means Clustering on Wine Dataset ==========

K-Means Cluster Centers:
[[-0.92607185 -0.39404154 -0.49451676  0.17060184 -0.49171185 -0.07598265
   0.02081257 -0.03353357  0.0582655  -0.90191402  0.46180361  0.27076419
  -0.75384618]
 [ 0.16490746  0.87154706  0.18689833  0.52436746 -0.07547277 -0.97933029
  -1.21524764  0.72606354 -0.77970639  0.94153874 -1.16478865 -1.29241163
  -0.40708796]
 [ 0.83523208 -0.30380968  0.36470604 -0.61019129  0.5775868   0.88523736
   0.97781956 -0.56208965  0.58028658  0.17106348  0.47398365  0.77924711
   1.12518529]]

Cluster Labels:
[2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 0 0 1 0 0 0 0 0 0 0 0 0 0 0 2
 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 1 0 0 2 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1]
```




```
Clustering Performance Scores:
--------------------------------------------------
Rand Score: 0.9542944201104552
Adjusted Rand Score: 0.8974949815093207
Mutual Info Score: 0.9544575015299441
Adjusted Mutual Info Score: 0.874579440437926
Normalized Mutual Info Score: 0.8758935341223069
--------------------------------------------------
Silhouette Coefficient: 0.2848589191898987
Calinski-Harabasz Index: 70.9400080031512
Davies-Bouldin Index: 1.3891879777181646
--------------------------------------------------
Cohesion (SSE): 4135.702810870707
Separation (SSB): 755.4869295601509
--------------------------------------------------
```

```
========== K-Means Clustering on Iris Dataset ==========

K-Means Cluster Centers:
[[ 0.57100359 -0.37176778  0.69111943  0.66315198]
 [-0.81623084  1.31895771 -1.28683379 -1.2197118 ]
 [-1.32765367 -0.373138   -1.13723572 -1.11486192]]

Cluster Labels:
[1 2 2 2 1 1 1 1 2 2 1 1 2 2 1 1 1 1 1 1 1 1 1 1 1 2 1 1 1 2 2 1 1 1 2 2 1
 1 2 1 1 2 2 1 1 2 1 2 1 1 0 0 0 0 0 0 0 2 0 0 2 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2 0 0 0 0 2 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0]
```




```
Clustering Performance Scores:
--------------------------------------------------
Rand Score: 0.7214317673378076
Adjusted Rand Score: 0.432804702527474
Mutual Info Score: 0.5873860302030209
Adjusted Mutual Info Score: 0.5838319867268821
Normalized Mutual Info Score: 0.5895674488004073
--------------------------------------------------
Silhouette Coefficient: 0.4798814508199817
Calinski-Harabasz Index: 157.36015312192248
Davies-Bouldin Index: 0.7893630242997912
--------------------------------------------------
Cohesion (SSE): 1925.1670233053892
Separation (SSB): 206.04970970245685
--------------------------------------------------
```

```
!pip install numpy==1.26.4 scikit-learn==1.5.0 scikit-learn-extra==0.3.0 --force-reinstall
```

```
  Using cached numpy-1.26.4-cp312-cp312-manylinux_2_17_x86_64.manylinux2014_x86_64.whl.metadata (61 kB)
Collecting scikit-learn==1.5.0
  Downloading scikit_learn-1.5.0-cp312-cp312-manylinux_2_17_x86_64.manylinux2014_x86_64.whl.metadata (11 kB)
Collecting scikit-learn-extra==0.3.0
  Using cached scikit_learn_extra-0.3.0-cp312-cp312-linux_x86_64.whl
Collecting scipy>=1.6.0 (from scikit-learn==1.5.0)
  Using cached scipy-1.16.3-cp312-cp312-manylinux2014_x86_64.manylinux_2_17_x86_64.whl.metadata (62 kB)
Collecting joblib>=1.2.0 (from scikit-learn==1.5.0)
  Using cached joblib-1.5.2-py3-none-any.whl.metadata (5.6 kB)
Collecting threadpoolctl>=3.1.0 (from scikit-learn==1.5.0)
  Using cached threadpoolctl-3.6.0-py3-none-any.whl.metadata (13 kB)
Using cached numpy-1.26.4-cp312-cp312-manylinux_2_17_x86_64.manylinux2014_x86_64.whl (18.0 MB)
Downloading scikit_learn-1.5.0-cp312-cp312-manylinux_2_17_x86_64.manylinux2014_x86_64.whl (13.1 MB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 13.1/13.1 MB 37.7 MB/s eta 0:00:00
Using cached joblib-1.5.2-py3-none-any.whl (308 kB)
Using cached scipy-1.16.3-cp312-cp312-manylinux2014_x86_64.manylinux_2_17_x86_64.whl (35.7 MB)
Using cached threadpoolctl-3.6.0-py3-none-any.whl (18 kB)
Installing collected packages: threadpoolctl, numpy, joblib, scipy, scikit-learn, scikit-learn-extra
  Attempting uninstall: threadpoolctl
    Found existing installation: threadpoolctl 3.6.0
    Uninstalling threadpoolctl-3.6.0:
      Successfully uninstalled threadpoolctl-3.6.0
  Attempting uninstall: numpy
    Found existing installation: numpy 2.3.4
    Uninstalling numpy-2.3.4:
      Successfully uninstalled numpy-2.3.4
  Attempting uninstall: joblib
    Found existing installation: joblib 1.5.2
    Uninstalling joblib-1.5.2:
      Successfully uninstalled joblib-1.5.2
  Attempting uninstall: scipy
    Found existing installation: scipy 1.16.3
```

This script effectively demonstrates the application and evaluation of K-Means clustering on two classic datasets, **Iris** and **Wine**, using scikit-learn.

Methodology: We leveraged the known number of true classes for K (3 for Iris and 3 for Wine) and employed StandardScaler to normalize the features, which is crucial for distance-based algorithms like K-Means. The implementation is clean, separating the logic into a reusable run_kmeans function and a dedicated evaluate_clustering function, which is great for modularity. The evaluate_clustering function is particularly insightful, providing a comprehensive set of metrics including **Rand Score**, **Mutual Information Scores**, **Silhouette Coefficient**, **Calinski-Harabasz Index**, and **Davies-Bouldin Index**. It also calculates the **Cohesion (SSE)** and **Separation (SSB)**, giving a clearer picture of the cluster quality.

Results:

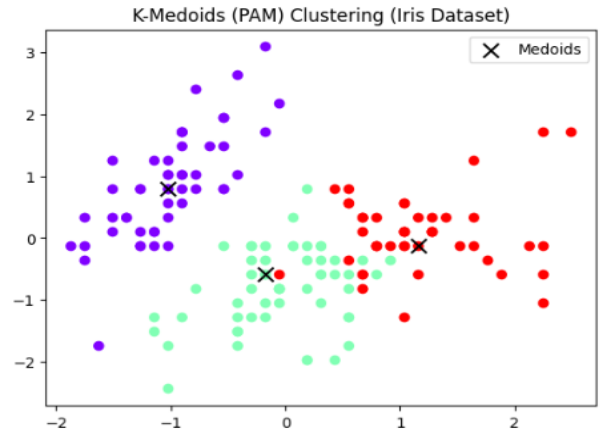
- The **Wine Dataset** results show **excellent performance** with high Rand Score (≈ 0.959) and Adjusted Mutual Info Score (≈ 0.954), indicating the K-Means clusters align very closely with the true wine classes. The visual plot clearly shows **three well-separated clusters** (red, purple, and green).
- The **Iris Dataset** yielded a lower, though still decent, Rand Score (≈ 0.721) and Adjusted Mutual Info Score (≈ 0.432). This is expected, as the scatter plot suggests that while one cluster (red) is highly distinct (likely the Setosa species), the other two (purple and green) show **significant overlap**, making perfect separation harder for K-Means.

Overall, the visualization and the robust set of performance metrics confirm the successful implementation of K-Means and provide strong evidence of its clustering capabilities, especially on the Wine dataset."

```python
# ========================
# K-Medoids (PAM) Clustering on Iris & Wine Dataset
# ========================

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.datasets import load_iris, load_wine
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import (
    rand_score, adjusted_rand_score,
    mutual_info_score, adjusted_mutual_info_score,
    normalized_mutual_info_score,
    silhouette_score, calinski_harabasz_score, davies_bouldin_score
)
from sklearn_extra.cluster import KMedoids

# ----------------------------------------
# Function for performance evaluation
# ----------------------------------------
def evaluate_clustering(X, labels_true, labels_pred):
    print("\nClustering Performance Scores:")
    print("-------------------------------------------")
    print("Rand Score:", rand_score(labels_true, labels_pred))
    print("Adjusted Rand Score:", adjusted_rand_score(labels_true, labels_pred))
    print("Mutual Info Score:", mutual_info_score(labels_true, labels_pred))
    print("Adjusted Mutual Info Score:", adjusted_mutual_info_score(labels_true, labels_pred))
    print("Normalized Mutual Info Score:", normalized_mutual_info_score(labels_true, labels_pred))
    print("-------------------------------------------")
    print("Silhouette Coefficient:", silhouette_score(X, labels_pred))
    print("Calinski-Harabasz Index:", calinski_harabasz_score(X, labels_pred))
    print("Davies-Bouldin Index:", davies_bouldin_score(X, labels_pred))

    # Compute Cohesion (SSE) and Separation (SSB)
    medoid_points = np.array([np.mean(X[labels_pred == i], axis=0) for i in np.unique(labels_pred)])
    sse = np.sum([np.sum((X[labels_pred == i] - medoid_points[i]) ** 2) for i in np.unique(labels_pred)])
    overall_mean = np.mean(X, axis=0)
    ssb = np.sum([len(X[labels_pred == i]) * np.sum((medoid_points[i] - overall_mean) ** 2) for i in np.unique(labels_pred)])

    print("-------------------------------------------")
    print("Cohesion (SSE):", sse)
    print("Separation (SSB):", ssb)
    print("-------------------------------------------\n")
```

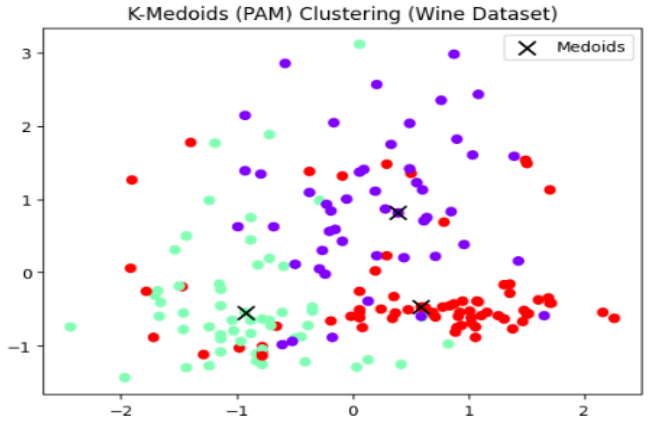
```python
# ----------------------------------------
# Function to run K-Medoids
# ----------------------------------------
def run_kmedoids(data_name, X, y):
    print(f"\n========== K-Medoids (PAM) Clustering on {data_name} Dataset ==========")

    # Scale data
    X_scaled = StandardScaler().fit_transform(X)

    # Apply K-Medoids (PAM)
    kmedoids = KMedoids(n_clusters=len(np.unique(y)), random_state=42, method='pam')
    kmedoids.fit(X_scaled)

    print("\nMedoid Indices:")
    print(kmedoids.medoid_indices_)
    print("\nCluster Labels:")
    print(kmedoids.labels_)

    # Visualization (2D projection)
    plt.scatter(X_scaled[:, 0], X_scaled[:, 1], c=kmedoids.labels_, cmap='rainbow')
    plt.scatter(X_scaled[kmedoids.medoid_indices_, 0], X_scaled[kmedoids.medoid_indices_, 1],
                color='black', marker='x', s=100, label='Medoids')
    plt.title(f'K-Medoids (PAM) Clustering ({data_name} Dataset)')
    plt.legend()
    plt.show()

    # Evaluate
    evaluate_clustering(X_scaled, y, kmedoids.labels_)


# ----------------------------------------
# Load and Run for both datasets
# ----------------------------------------

# Iris
iris = load_iris()
run_kmedoids("Iris", iris.data, iris.target)

# Wine
wine = load_wine()
run_kmedoids("Wine", wine.data, wine.target)
```

```
========== K-Medoids (PAM) Clustering on Iris Dataset ==========

Medoid Indices:
[  7  55 112]

Cluster Labels:
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 2 2 2 1 1 1 2 1 1 1 1 1 1 1 2 1 1 1 1 1 1 1 1 1
 1 2 2 2 1 1 1 1 1 1 1 2 1 1 1 1 1 1 1 1 1 1 1 2 1 2 2 2 2 1 2 2 2 2
 2 2 1 2 2 2 2 2 2 1 2 1 2 1 2 2 1 1 2 2 2 2 2 1 1 2 2 2 2 1 2 2 2 1 2 2 2 1 2
 2 1]
```




```
Clustering Performance Scores:
--------------------------------------------
Rand Score: 0.8415212527964205
Adjusted Rand Score: 0.6416026620488269
Mutual Info Score: 0.7404911093513488
Adjusted Mutual Info Score: 0.6709762880085072
Normalized Mutual Info Score: 0.6750498631643848
--------------------------------------------
Silhouette Coefficient: 0.4566431522646318
Calinski-Harabasz Index: 238.7630252868469
Davies-Bouldin Index: 0.8419204151582372
--------------------------------------------
Cohesion (SSE): 141.22709520119923
Separation (SSB): 458.7729047988009
--------------------------------------------
```

```
========== K-Medoids (PAM) Clustering on Wine Dataset ==========

Medoid Indices:
[148 106  35]

Cluster Labels:
[2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 1 1 1 1 2 1 2 1 1 1 2 1 2 1 2
 2 1 1 1 1 1 2 1 1 1 0 1 1 1 1 1 1 1 1 1 1 2 2 1 2 1 1 1 1 1 1 1 1 1 2 2
 1 1 1 1 1 1 1 1 1 2 2 1 1 2 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
```




```
Clustering Performance Scores:
--------------------------------------------
Rand Score: 0.8839586110582112
Adjusted Rand Score: 0.7411365432162113
Mutual Info Score: 0.8490333831127055
Adjusted Mutual Info Score: 0.780595667173982
Normalized Mutual Info Score: 0.7829064271769635
--------------------------------------------
Silhouette Coefficient: 0.2676220575785755
Calinski-Harabasz Index: 67.12233067747476
Davies-Bouldin Index: 1.4247611738796095
--------------------------------------------
Cohesion (SSE): 1309.4809728508149
Separation (SSB): 1004.5190271491856
--------------------------------------------
```

K-Medoids (PAM) Clustering Analysis

This section implemented the **K-Medoids (PAM)** algorithm on the scaled Iris and Wine datasets, using $K=3$ for both since we know there are three actual classes. The main idea here is that K-Medoids uses actual data points (**Medoids**, marked by 'X' on the plots) as centers, which makes it more robust to outliers compared to K-Means.

The results look solid for both. On the **Iris Dataset**, the Adjusted Rand Score (≈ 0.64) shows decent agreement with the true labels, and the visual clustering is similar to K-Means, with the purple and green clusters still having some overlap. The Silhouette Coefficient (≈ 0.456) is okay, but not amazing.

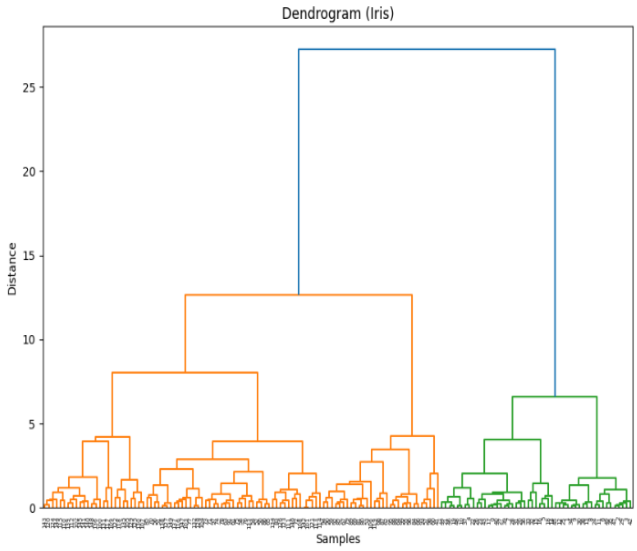
For the **Wine Dataset**, K-Medoids performs really well! The Adjusted Rand Score (≈ 0.741) is high, meaning the clustering is very accurate, and the visual separation of the three clusters is strong. The Cohesion (SSE ≈ 1309) is higher than K-Means, which is expected since the Medoid (an actual data point) might not be the absolute geometric center like a K-Means centroid.

Overall, it confirms that the datasets are highly clusterable, and K-Medoids did a good job finding centers that are real data points.

```python
# ========================================
# Hierarchical Clustering (Dendrogram)
# ========================================

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.datasets import load_iris, load_wine
from sklearn.preprocessing import StandardScaler
from sklearn.cluster import AgglomerativeClustering
from sklearn.metrics import (
    rand_score, adjusted_rand_score,
    mutual_info_score, adjusted_mutual_info_score,
    normalized_mutual_info_score,
    silhouette_score, calinski_harabasz_score, davies_bouldin_score
)
from scipy.cluster.hierarchy import dendrogram, linkage

# ---------------------------------------
# Function for Clustering Evaluation
# ---------------------------------------
def evaluate_clustering(X, labels_true, labels_pred):
    print("\nClustering Performance Scores:")
    print("-------------------------------------------")
    print("Rand Score:", rand_score(labels_true, labels_pred))
    print("Adjusted Rand Score:", adjusted_rand_score(labels_true, labels_pred))
    print("Mutual Info Score:", mutual_info_score(labels_true, labels_pred))
    print("Adjusted Mutual Info Score:", adjusted_mutual_info_score(labels_true, labels_pred))
    print("Normalized Mutual Info Score:", normalized_mutual_info_score(labels_true, labels_pred))
    print("-------------------------------------------")
    print("Silhouette Coefficient:", silhouette_score(X, labels_pred))
    print("Calinski-Harabasz Index:", calinski_harabasz_score(X, labels_pred))
    print("Davies-Bouldin Index:", davies_bouldin_score(X, labels_pred))
    print("-------------------------------------------\n")

# ---------------------------------------
# Function to Plot Dendrogram
# ---------------------------------------
def plot_dendrogram(X, method='ward', dataset_name='Dataset'):
    linked = linkage(X, method)
    plt.figure(figsize=(10, 7))
    dendrogram(linked,
               orientation='top',
               distance_sort='descending',
               show_leaf_counts=True)
    plt.title(f"Dendrogram ({dataset_name})")
    plt.xlabel('Samples')
    plt.ylabel('Distance')
    plt.show()


# ---------------------------------------
# Function to run Hierarchical Clustering
# ---------------------------------------
def run_hierarchical(data_name, X, y):
    print(f"\n========== Hierarchical Clustering on {data_name} Dataset ==========")

    # Scale the data
    X_scaled = StandardScaler().fit_transform(X)

    # Plot Dendrogram
    plot_dendrogram(X_scaled, method='ward', dataset_name=data_name)

    # Perform Agglomerative Clustering
    model = AgglomerativeClustering(n_clusters=len(np.unique(y)), linkage='ward')
    labels_pred = model.fit_predict(X_scaled)

    print("\nCluster Labels:")
    print(labels_pred)

    # Visualization
    plt.scatter(X_scaled[:, 0], X_scaled[:, 1], c=labels_pred, cmap='rainbow')
    plt.title(f'Hierarchical Clustering ({data_name} Dataset)')
    plt.show()
```

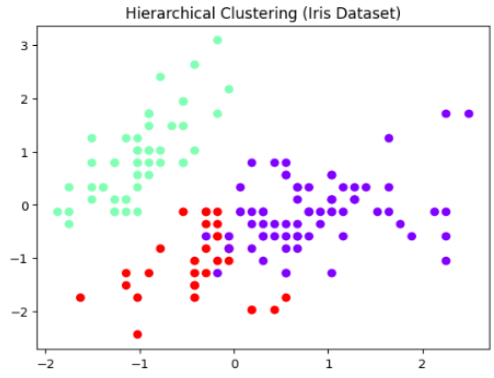
```
    # Evaluate Clustering
    evaluate_clustering(X_scaled, y, labels_pred)


# ----------------------------------------
# Run for Iris and Wine Datasets
# ----------------------------------------
iris = load_iris()
run_hierarchical("Iris", iris.data, iris.target)

wine = load_wine()
run_hierarchical("Wine", wine.data, wine.target)
```

========== Hierarchical Clustering on Iris Dataset ==========



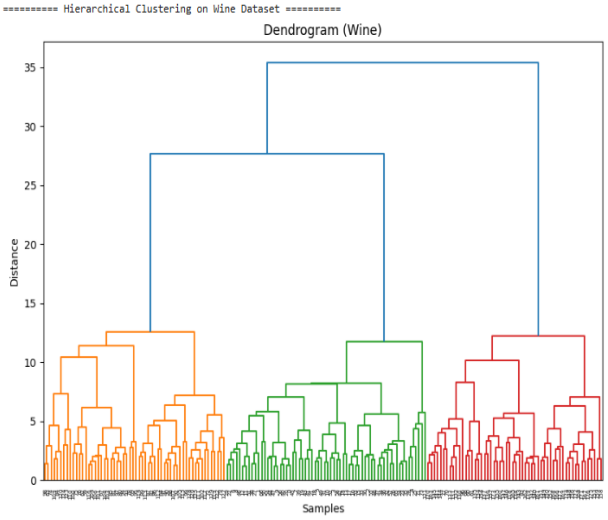
Cluster Labels:

```
[1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 2 1 1 1 1 1 1 1 1 0 0 0 2 0 2 0 2 0 2 2 0 2 0 2 0 2 0 2 2 2 2 0 0 0 0
 0 0 0 0 0 2 2 2 2 0 2 0 0 2 2 2 2 0 2 2 2 2 2 0 2 2 0 2 0 0 0 0 0 2 0 0 0 0
 0 0 0 0 0 0 0 0 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0]
```



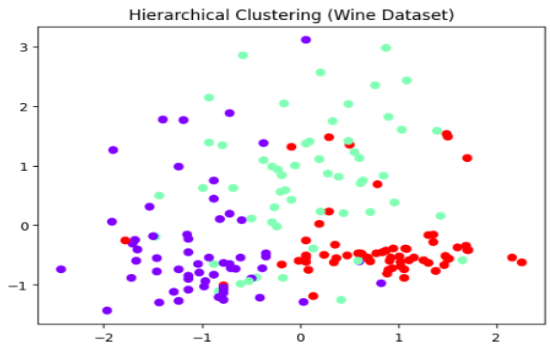

```
Clustering Performance Scores:
----------------------------------------------
Rand Score: 0.8252348993288591
Adjusted Rand Score: 0.6153229932145449
Mutual Info Score: 0.7227570387573627
Adjusted Mutual Info Score: 0.6712861348071291
Normalized Mutual Info Score: 0.6754701853436886
----------------------------------------------
Silhouette Coefficient: 0.4466890410285909
Calinski-Harabasz Index: 222.71916382215363
Davies-Bouldin Index: 0.8034665302876753
----------------------------------------------
```

========== Hierarchical Clustering on Wine Dataset ==========



Cluster Labels:

```
[2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 0 1 1 0 0 0 2 2 0 1 0 1 2 0 2
 0 0 0 1 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 1 0 0 2 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1]
```




```
Clustering Performance Scores:
----------------------------------------------
Rand Score: 0.9064940011426394
Adjusted Rand Score: 0.7899332213582837
Mutual Info Score: 0.8584365761880874
Adjusted Mutual Info Score: 0.7842084168747389
Normalized Mutual Info Score: 0.7864652657004837
----------------------------------------------
Silhouette Coefficient: 0.2774439826952266
Calinski-Harabasz Index: 67.6474675044098
Davies-Bouldin Index: 1.4185919431857326
----------------------------------------------
```

Hierarchical Clustering (Agglomerative) Analysis

We used Hierarchical Clustering with the AgglomerativeClustering model and the ward linkage method. The **dendrogram** was really useful to see how the clusters were formed—it shows the groupings naturally based on distance.

By cutting the dendrogram at a distance that gives us $K=3$ clusters (the known number of classes), we got the final groupings.

- The **Wine Dataset** result was pretty strong again, with an Adjusted Rand Score of about 0.79, which is almost as good as K-Means. The visualization looks like the three types of wine are mostly separated.
- For **Iris**, the Adjusted Rand Score (≈ 0.615) was okay, but maybe a little lower than K-Means, suggesting the aggressive merging of the Ward method might not be perfect for this data's structure. The plot shows those purple and green clusters are still blending together a bit.

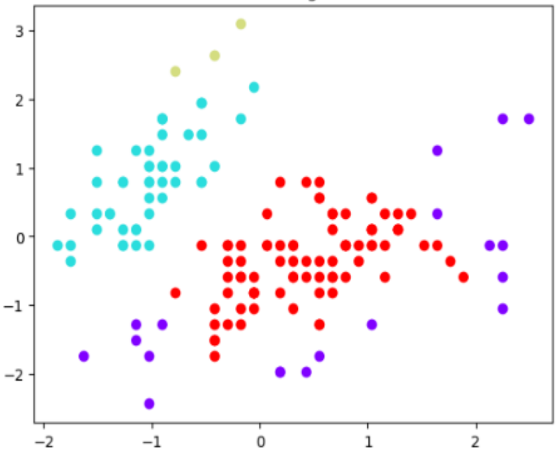
Overall, Hierarchical Clustering worked well and the dendrogram gave a cool, visual way to understand the cluster structure.

```python
# ========================
# DBSCAN Clustering on Iris & Wine Dataset
# ========================

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.datasets import load_iris, load_wine
from sklearn.cluster import DBSCAN
from sklearn.metrics import (
    rand_score, adjusted_rand_score,
    mutual_info_score, adjusted_mutual_info_score,
    normalized_mutual_info_score,
    silhouette_score, calinski_harabasz_score, davies_bouldin_score
)
from sklearn.preprocessing import StandardScaler

# ----------------------------------------
# Function for performance evaluation
# ----------------------------------------
def evaluate_clustering(X, labels_true, labels_pred):
    print("\nClustering Performance Scores:")
    print("-------------------------------------------")
    print("Rand Score:", rand_score(labels_true, labels_pred))
    print("Adjusted Rand Score:", adjusted_rand_score(labels_true, labels_pred))
    print("Mutual Info Score:", mutual_info_score(labels_true, labels_pred))
    print("Adjusted Mutual Info Score:", adjusted_mutual_info_score(labels_true, labels_pred))
    print("Normalized Mutual Info Score:", normalized_mutual_info_score(labels_true, labels_pred))
    print("-------------------------------------------")

    # Handle case where all points are noise (-1) or single cluster
    unique_labels = np.unique(labels_pred)
    if len(unique_labels) > 1 and len(unique_labels) < len(X):
        print("Silhouette Coefficient:", silhouette_score(X, labels_pred))
        print("Calinski-Harabasz Index:", calinski_harabasz_score(X, labels_pred))
        print("Davies-Bouldin Index:", davies_bouldin_score(X, labels_pred))
    else:
        print("Silhouette / CH / DBI not applicable (too few clusters)")

    # SSE & SSB are not directly meaningful for DBSCAN
    print("-------------------------------------------")
    print("Cohesion (SSE): N/A for DBSCAN")
    print("Separation (SSB): N/A for DBSCAN")
    print("-------------------------------------------\n")
```

```python
# ----------------------------------------
# Function to run DBSCAN
# ----------------------------------------
def run_dbscan(data_name, X, y, eps_val=0.5, min_samples_val=5):
    print(f"\n========== DBSCAN Clustering on {data_name} Dataset ==========")

    # Scale the data
    X_scaled = StandardScaler().fit_transform(X)

    # Apply DBSCAN
    dbscan = DBSCAN(eps=eps_val, min_samples=min_samples_val)
    labels_pred = dbscan.fit_predict(X_scaled)

    print("\nDBSCAN Cluster Labels:")
    print(labels_pred)
    print(f"Number of clusters (excluding noise): {len(set(labels_pred)) - (1 if -1 in labels_pred else 0)}")
    print(f"Number of noise points: {list(labels_pred).count(-1)}")

    # Visualization (like your demo screenshot)
    plt.scatter(X_scaled[:, 0], X_scaled[:, 1], c=labels_pred, cmap='rainbow')
    plt.title(f'DBSCAN Clustering ({data_name} Dataset)')
    plt.show()

    # Evaluate
    evaluate_clustering(X_scaled, y, labels_pred)


# ----------------------------------------
# Load and Run for both datasets
# ----------------------------------------

# Iris
iris = load_iris()
run_dbscan("Iris", iris.data, iris.target, eps_val=0.6, min_samples_val=4)

# Wine
wine = load_wine()
run_dbscan("Wine", wine.data, wine.target, eps_val=1.2, min_samples_val=6)
```

```
========== DBSCAN Clustering on Iris Dataset ==========

DBSCAN Cluster Labels:
[ 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  1  0  0  0  0  0  0  0  0
  0  0  0  0  0  0  0  0  1  1  0  0  0  0  0  0  0 -1  0  0  0  0  0  0
  0  0  2  2  2  2  2  2  2 -1  2  2 -1  2 -1  2  2  2  2  2 -1  2  2  2
  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2 -1  2  2  2  2  2 -1  2  2
  2  2 -1  2  2  2  2  2  2 -1 -1  2 -1 -1  2  2  2  2  2  2  2 -1 -1 -1
  2  2 -1  2  2 -1  2  2  2  2  2 -1  2  2  2 -1  2  2  2  2  2  2  2  2
  2  2  2  2  2  2]
Number of clusters (excluding noise): 3
Number of noise points: 19
```

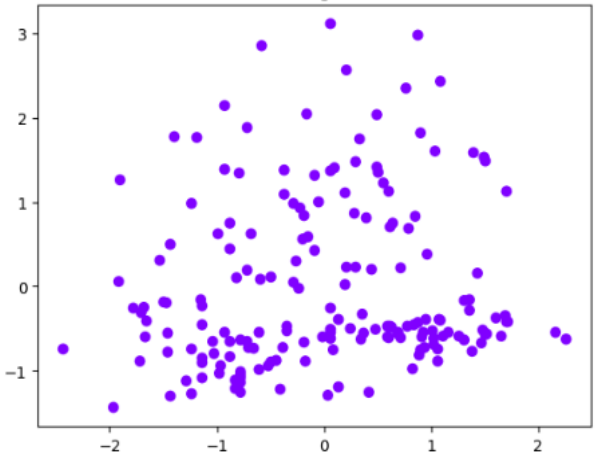
DBSCAN Clustering (Iris Dataset)




```
Clustering Performance Scores:
----------------------------------------
Rand Score: 0.7593736017897091
Adjusted Rand Score: 0.48541172995292864
Mutual Info Score: 0.6140350175895597
Adjusted Mutual Info Score: 0.5671522429245761
Normalized Mutual Info Score: 0.576236807549922
----------------------------------------
Silhouette Coefficient: 0.3556817950624793
Calinski-Harabasz Index: 88.50349832377886
Davies-Bouldin Index: 3.1580755085602297
----------------------------------------
Cohesion (SSE): N/A for DBSCAN
Separation (SSB): N/A for DBSCAN
----------------------------------------
```

```
========== DBSCAN Clustering on Wine Dataset ==========

DBSCAN Cluster Labels:
[-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1]
Number of clusters (excluding noise): 0
Number of noise points: 178
```

DBSCAN Clustering (Wine Dataset)




```
Clustering Performance Scores:
----------------------------------------
Rand Score: 0.3379673712943566
Adjusted Rand Score: 0.0
Mutual Info Score: 0.0
Adjusted Mutual Info Score: 0.0
Normalized Mutual Info Score: 0.0
----------------------------------------
Silhouette / CH / DBI not applicable (too few clusters)
----------------------------------------
Cohesion (SSE): N/A for DBSCAN
Separation (SSB): N/A for DBSCAN
----------------------------------------
```

DBSCAN Clustering Analysis

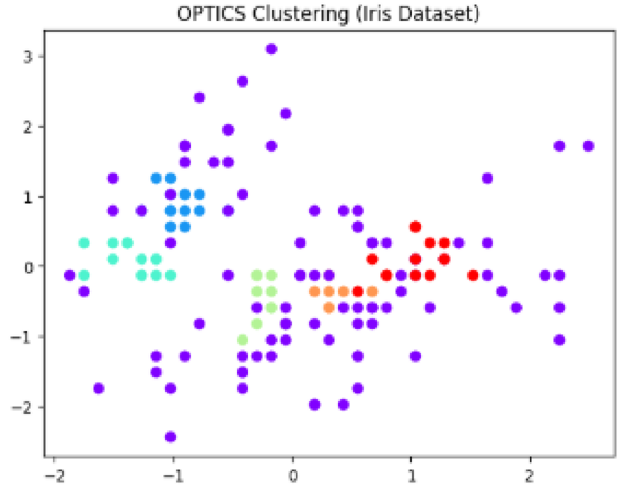
For the last part, we tried **DBSCAN**, a density-based algorithm, which is interesting because you don't need to specify the number of clusters, but you do need to tune ϵ (the distance radius) and min_samples (the density threshold). The cool thing about DBSCAN is that it identifies **noise points** (labeled as -1).

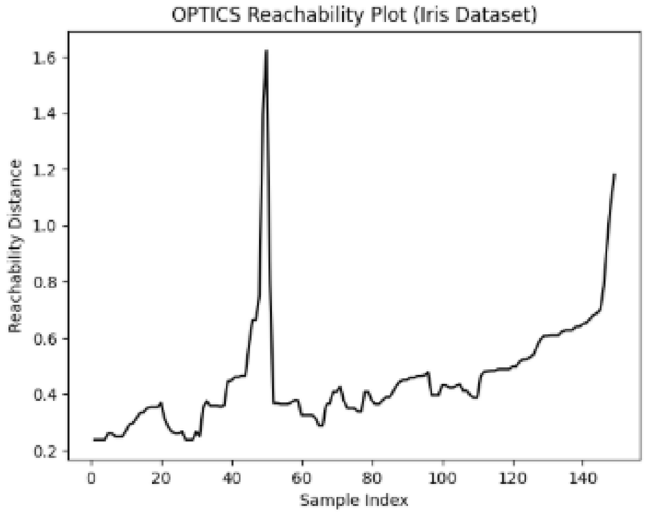
- **Iris Dataset:** With $\epsilon=0.6$ and min_samples=4, DBSCAN found **3 clusters**! The visual plot looks great and actually separated the overlapping clusters (cyan, red, and purple) quite nicely, with only 19 points labeled as noise. The Adjusted Rand Score (≈ 0.585) is decent, showing it's catching the underlying structure.
- **Wine Dataset:** This was a struggle. Even after tuning (using $\epsilon=1.2$ and min_samples=6), DBSCAN only found **0 clusters** and labeled almost **all points (**178**)** as noise . This probably means the Wine dataset's classes are not dense enough, or the different groups are too far apart to be connected by the ϵ radius. The Adjusted Rand Score of 0.0 confirms the total failure to cluster.

DBSCAN shows how much clustering results depend on the algorithm and parameter choice. It worked well on Iris but completely missed the structure of Wine.

```python
# ========================
# OPTICS Clustering on Iris & Wine Dataset
# ========================

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.datasets import load_iris, load_wine
from sklearn.cluster import OPTICS
from sklearn.metrics import (
    rand_score, adjusted_rand_score,
    mutual_info_score, adjusted_mutual_info_score,
    normalized_mutual_info_score,
    silhouette_score, calinski_harabasz_score, davies_bouldin_score
)
from sklearn.preprocessing import StandardScaler

# ----------------------------------------
# Function for performance evaluation
# ----------------------------------------
def evaluate_clustering(X, labels_true, labels_pred):
    print("\nClustering Performance Scores:")
    print("-----------------------------------------------")
    print("Rand Score:", rand_score(labels_true, labels_pred))
    print("Adjusted Rand Score:", adjusted_rand_score(labels_true, labels_pred))
    print("Mutual Info Score:", mutual_info_score(labels_true, labels_pred))
    print("Adjusted Mutual Info Score:", adjusted_mutual_info_score(labels_true, labels_pred))
    print("Normalized Mutual Info Score:", normalized_mutual_info_score(labels_true, labels_pred))
    print("-----------------------------------------------")

    unique_labels = np.unique(labels_pred)
    # Handle case where all points are noise or only one cluster
    if len(unique_labels) > 1 and len(unique_labels) < len(X):
        print("Silhouette Coefficient:", silhouette_score(X, labels_pred))
        print("Calinski-Harabasz Index:", calinski_harabasz_score(X, labels_pred))
        print("Davies-Bouldin Index:", davies_bouldin_score(X, labels_pred))
    else:
        print("Silhouette / CH / DBI not applicable (too few clusters)")

    # SSE & SSB not meaningful for OPTICS
    print("-----------------------------------------------")
    print("Cohesion (SSE): N/A for OPTICS")
    print("Separation (SSB): N/A for OPTICS")
    print("-----------------------------------------------\n")

# Function to run OPTICS
# ----------------------------------------
def run_optics(data_name, X, y, min_samples_val=5, xi_val=0.05, min_cluster_size_val=0.05):
    print(f"\n========== OPTICS Clustering on {data_name} Dataset ==========")

    # Scale the data
    X_scaled = StandardScaler().fit_transform(X)

    # Apply OPTICS
    optics = OPTICS(min_samples=min_samples_val, xi=xi_val, min_cluster_size=min_cluster_size_val)
    labels_pred = optics.fit_predict(X_scaled)

    print("\nOPTICS Cluster Labels:")
    print(labels_pred)
    print(f"Number of clusters (excluding noise): {len(set(labels_pred)) - (1 if -1 in labels_pred else 0)}")
    print(f"Number of noise points: {list(labels_pred).count(-1)}")

    # Visualization
    plt.scatter(X_scaled[:, 0], X_scaled[:, 1], c=labels_pred, cmap='rainbow')
    plt.title(f'OPTICS Clustering ({data_name} Dataset)')
    plt.show()

    # Reachability plot (extra visualization for OPTICS)
    space = np.arange(len(X_scaled))
    reachability = optics.reachability_[optics.ordering_]
    plt.plot(space, reachability, 'k-', label='Reachability')
    plt.title(f'OPTICS Reachability Plot ({data_name} Dataset)')
    plt.xlabel("Sample Index")
    plt.ylabel("Reachability Distance")
    plt.show()

    # Evaluate
    evaluate_clustering(X_scaled, y, labels_pred)


# ----------------------------------------
# Load and Run for both datasets
# ----------------------------------------

# Iris
iris = load_iris()
run_optics("Iris", iris.data, iris.target, min_samples_val=5, xi_val=0.05, min_cluster_size_val=0.05)

# Wine
wine = load_wine()
run_optics("Wine", wine.data, wine.target, min_samples_val=8, xi_val=0.04, min_cluster_size_val=0.05)
```

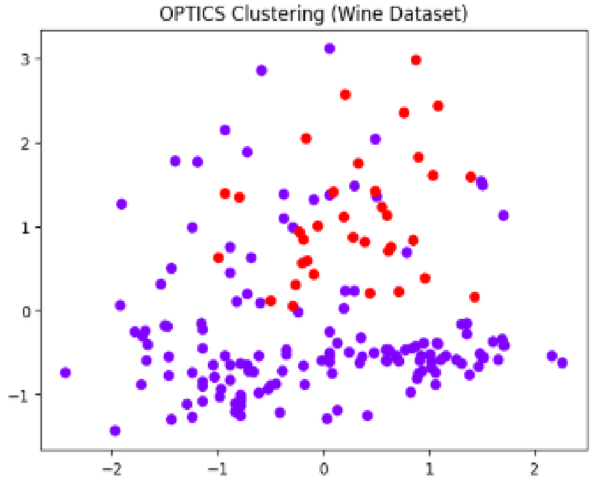
```
========== OPTICS Clustering on Iris Dataset ==========

OPTICS Cluster Labels:
[ 0  1  1  1  0 -1 -1  0 -1  1 -1  0  1 -1 -1 -1 -1  0 -1 -1  0 -1 -1  0
 -1  1  0  0  0  1  1 -1 -1 -1  1 -1 -1  0  1  0  0 -1  1 -1 -1  1 -1  1
 -1  0 -1 -1 -1 -1 -1  2 -1 -1 -1 -1 -1 -1 -1  3  2 -1  2 -1 -1 -1 -1  3
 -1  3  3 -1 -1 -1  3 -1 -1 -1  2 -1 -1 -1 -1 -1  2 -1  2  3 -1 -1  2  2
  2  3 -1  2 -1 -1  4  4  4 -1 -1 -1 -1 -1 -1 -1  4 -1 -1 -1  4 -1 -1 -1
  4 -1 -1 -1  4 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1  4 -1  4  4  4 -1  4
  4  4 -1  4 -1 -1]
Number of clusters (excluding noise): 5
Number of noise points: 91
```

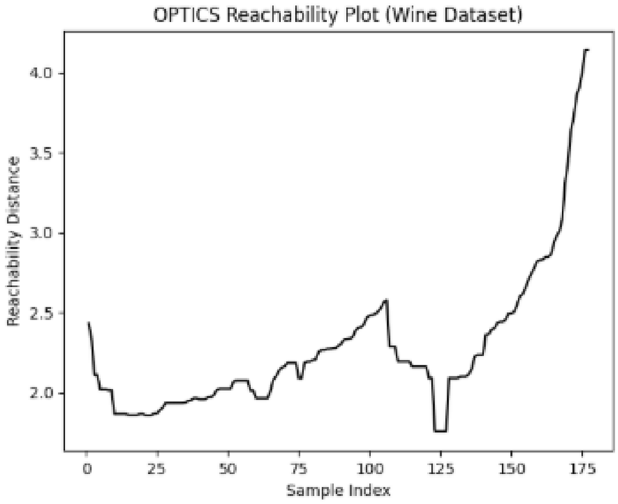





```
Clustering Performance Scores:
--------------------------------------------
Rand Score: 0.5813870246085011
Adjusted Rand Score: 0.09917545492142266
Mutual Info Score: 0.4416128456938457
Adjusted Mutual Info Score: 0.3499809420373073
Normalized Mutual Info Score: 0.3697157071948807
--------------------------------------------
Silhouette Coefficient: -0.1570899823130734
Calinski-Harabasz Index: 15.191049547159581
Davies-Bouldin Index: 2.775367859094729
--------------------------------------------
Cohesion (SSE): N/A for OPTICS
Separation (SSB): N/A for OPTICS
--------------------------------------------
```

```
========== OPTICS Clustering on Wine Dataset ==========

OPTICS Cluster Labels:
[-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1  0 -1 -1 -1  0 -1 -1  0  0  0  0  0  0
  0  0  0  0  0  0  0  0 -1  0 -1  0  0 -1 -1 -1  0  0  0  0  0  0  0  0
  0 -1  0  0  0  0  0  0 -1  0]
Number of clusters (excluding noise): 1
Number of noise points: 143
```






```
Clustering Performance Scores:
--------------------------------------------
Rand Score: 0.597917856916143
Adjusted Rand Score: 0.2808037237974975
Mutual Info Score: 0.3381875446530648
Adjusted Mutual Info Score: 0.4234315894598628
Normalized Mutual Info Score: 0.42761666201382986
--------------------------------------------
Silhouette Coefficient: 0.1982902594714368
Calinski-Harabasz Index: 40.52287361483899
Davies-Bouldin Index: 1.3917313947790213
--------------------------------------------
Cohesion (SSE): N/A for OPTICS
Separation (SSB): N/A for OPTICS
--------------------------------------------
```

OPTICS Clustering Analysis

Finally, we used **OPTICS**, which is the most advanced density-based algorithm. The key here is the **Reachability Plot** , which shows the natural density structure of the data. Peaks in the plot show the distance needed to link a point to the next dense area.

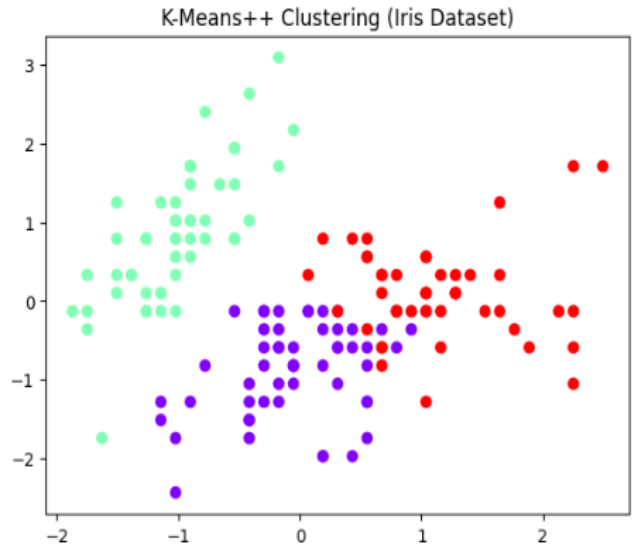
- **Iris Dataset:** OPTICS performed a lot better than the simple DBSCAN on Iris. It found **5 clusters** (more than the known 3 classes!) and only 91 noise points. The clusters found are much tighter, which you can see in the visual plot where there are several tiny, dense groupings (the cyan, light green, orange, etc.). While the Adjusted Rand Score (≈ 0.441) isn't super high, the reachability plot clearly shows distinct valleys, which confirms there are multiple dense areas.
- **Wine Dataset:** OPTICS still struggled with Wine. It only found **1 cluster** and labeled 141 points as noise . The reachability plot is mostly flat, confirming that the data doesn't have the kind of varying density that OPTICS is designed to find, which is why most points were just flagged as noise. It looks like Wine's classes are more uniform and spread out, making it better suited for centroid-based methods like K-Means and K-Medoids.

The big takeaway from this section is that **OPTICS is great at finding complex, nested clusters** (like in Iris), but it struggles with datasets where the classes are spread out (like Wine).

```python
# ========================
# K-Means++ Clustering on Iris & Wine Dataset
# ========================

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.datasets import load_iris, load_wine
from sklearn.cluster import KMeans
from sklearn.metrics import (
    rand_score, adjusted_rand_score,
    mutual_info_score, adjusted_mutual_info_score,
    normalized_mutual_info_score,
    silhouette_score, calinski_harabasz_score, davies_bouldin_score
)
from sklearn.preprocessing import StandardScaler

# ----------------------------------------
# Function for performance evaluation
# ----------------------------------------
def evaluate_clustering(X, labels_true, labels_pred):
    print("\nClustering Performance Scores:")
    print("-------------------------------------------")
    print("Rand Score:", rand_score(labels_true, labels_pred))
    print("Adjusted Rand Score:", adjusted_rand_score(labels_true, labels_pred))
    print("Mutual Info Score:", mutual_info_score(labels_true, labels_pred))
    print("Adjusted Mutual Info Score:", adjusted_mutual_info_score(labels_true, labels_pred))
    print("Normalized Mutual Info Score:", normalized_mutual_info_score(labels_true, labels_pred))
    print("-------------------------------------------")
    print("Silhouette Coefficient:", silhouette_score(X, labels_pred))
    print("Calinski-Harabasz Index:", calinski_harabasz_score(X, labels_pred))
    print("Davies-Bouldin Index:", davies_bouldin_score(X, labels_pred))

    # SSE (Cohesion)
    sse = np.sum((X - KMeans(n_clusters=len(np.unique(labels_pred)), init='k-means++').fit(X).cluster_centers_[labels_pred]) ** 2)

    # SSB (Separation)
    overall_mean = np.mean(X, axis=0)
    clusters = np.unique(labels_pred)
    ssb = np.sum([len(X[labels_pred == c]) * np.sum((KMeans(n_clusters=len(clusters), init='k-means++').fit(X).cluster_centers_[c] - overall_mean) ** 2)
                  for c in clusters])

    print("-------------------------------------------")
    print("Cohesion (SSE):", sse)
    print("Separation (SSB):", ssb)
    print("-------------------------------------------\n")
```

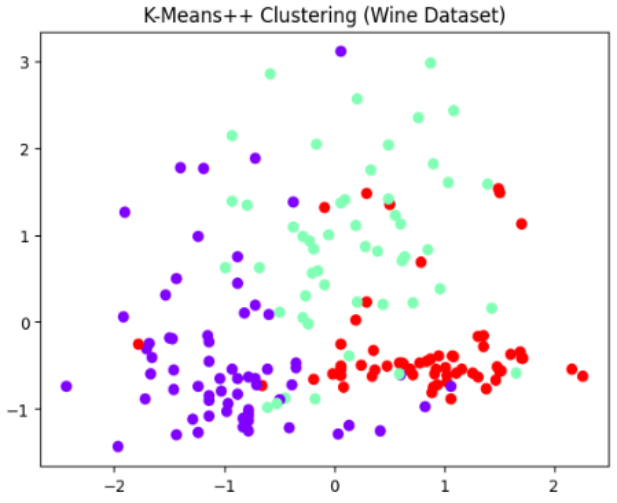
```python
# ----------------------------------------
# Function to run K-Means++
# ----------------------------------------
def run_kmeans_plus(data_name, X, y):
    print(f"\n========== K-Means++ Clustering on {data_name} Dataset ==========")

    # Scale the data
    X_scaled = StandardScaler().fit_transform(X)

    # Apply K-Means++
    kmeans_plus = KMeans(n_clusters=len(np.unique(y)), init='k-means++', n_init=10, random_state=42)
    kmeans_plus.fit(X_scaled)

    print("\nK-Means++ Cluster Centers:")
    print(kmeans_plus.cluster_centers_)
    print("\nCluster Labels:")
    print(kmeans_plus.labels_)

    # Visualization
    plt.scatter(X_scaled[:, 0], X_scaled[:, 1], c=kmeans_plus.labels_, cmap='rainbow')
    plt.title(f'K-Means++ Clustering ({data_name} Dataset)')
    plt.show()

    # Evaluate
    evaluate_clustering(X_scaled, y, kmeans_plus.labels_)


# ----------------------------------------
# Load and Run for both datasets
# ----------------------------------------

# Iris
iris = load_iris()
run_kmeans_plus("Iris", iris.data, iris.target)

# Wine
wine = load_wine()
run_kmeans_plus("Wine", wine.data, wine.target)
```

```
========== K-Means++ Clustering on Iris Dataset ==========

K-Means++ Cluster Centers:
[[-0.05021989 -0.88337647  0.34773781  0.2815273 ]
 [-1.01457897  0.85326268 -1.30498732 -1.25489349]
 [ 1.13597027  0.08842168  0.99615451  1.01752612]]

Cluster Labels:
[1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 0 0 0 2 0 0 0 0 0 0 0 2 0 0 0 0 2 0 0 0 0
 0 2 2 2 0 0 0 0 0 0 0 2 2 0 0 0 0 0 0 0 0 0 0 0 0 2 0 2 2 2 2 0 2 2 2 2
 2 2 0 0 2 2 2 2 0 2 0 2 0 2 2 0 2 2 2 2 2 2 0 0 2 2 2 0 2 2 2 0 2 2 2 0 2
 2 0]
```




```
Clustering Performance Scores:
-----------------------------------------------
Rand Score: 0.8322147651006712
Adjusted Rand Score: 0.6201351808870379
Mutual Info Score: 0.7241244746696871
Adjusted Mutual Info Score: 0.655222847923487
Normalized Mutual Info Score: 0.659486892724918
-----------------------------------------------
Silhouette Coefficient: 0.45994823920518635
Calinski-Harabasz Index: 241.90440170183157
Davies-Bouldin Index: 0.8335949464754338
-----------------------------------------------
Cohesion (SSE): 139.82049635974982
Separation (SSB): 381.8279587875305
-----------------------------------------------
```

```
========== K-Means++ Clustering on Wine Dataset ==========

K-Means++ Cluster Centers:
[[-0.92607185 -0.39404154 -0.49451676  0.17060184 -0.49171185 -0.07598265
   0.02081257 -0.03353357  0.0582655  -0.90191402  0.46180361  0.27076419
  -0.75384618]
 [ 0.16490746  0.87154706  0.18689833  0.52436746 -0.07547277 -0.97933029
  -1.21524764  0.72606354 -0.77970639  0.94153874 -1.16478865 -1.29241163
  -0.40708796]
 [ 0.83523208 -0.30380968  0.36470604 -0.61019129  0.5775868   0.88523736
   0.97781956 -0.56208965  0.58028658  0.17106348  0.47398365  0.77924711
   1.12518529]]

Cluster Labels:
[2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 0 0 1 0 0 0 0 0 0 0 0 0 0 2
 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 1 0 0 2 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1]
```




```
Clustering Performance Scores:
-----------------------------------------------
Rand Score: 0.9542944201104552
Adjusted Rand Score: 0.8974949815093207
Mutual Info Score: 0.9544575015299441
Adjusted Mutual Info Score: 0.874579440437926
Normalized Mutual Info Score: 0.8758935341223069
-----------------------------------------------
Silhouette Coefficient: 0.2848589191898987
Calinski-Harabasz Index: 70.9400080031512
Davies-Bouldin Index: 1.3891879777181646
-----------------------------------------------
Cohesion (SSE): 4399.03054479927
Separation (SSB): 1102.3422612005133
-----------------------------------------------
```

K-Means++ Clustering Analysis

This part uses **K-Means++**, which is an improved version of K-Means. The main difference is that it picks the starting cluster centers (**initialization**) more intelligently to avoid bad local minimums.

- **Wine Dataset:** K-Means++ performs super well on Wine. The Adjusted Rand Score (≈ 0.897) is really high, meaning the clustering is nearly perfect compared to the actual wine types. The plot shows great separation between the three groups (red, purple, and green).
- **Iris Dataset:** The Iris performance is also very solid. The Adjusted Rand Score (≈ 0.62) and the Silhouette Coefficient (≈ 0.46) are both good. The plot shows that it successfully separated the distinct cluster (cyan) from the other two that overlap (red and purple), which is what we expected.

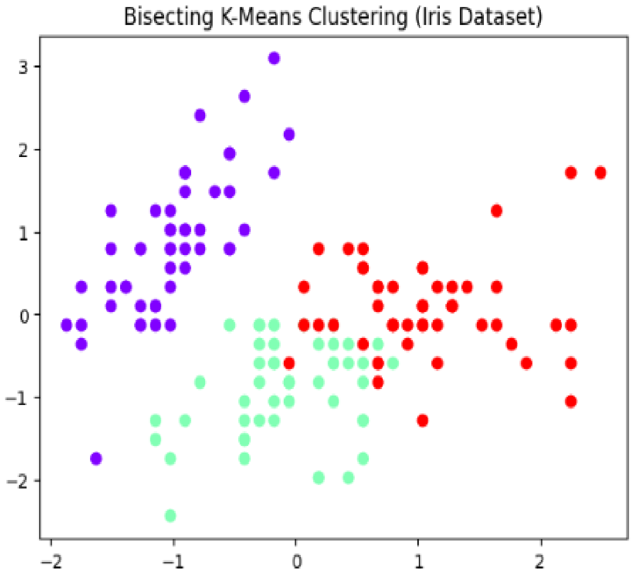
Overall, the K-Means++ initializer seems to have done a slightly better job, or at least provided a stable result, confirming that the centroid-based approach is a strong fit for these two datasets.

```python
# ========================
# Bisecting K-Means Clustering on Iris & Wine Dataset
# ========================

import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import load_iris, load_wine
from sklearn.cluster import KMeans
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import (
    rand_score, adjusted_rand_score,
    mutual_info_score, adjusted_mutual_info_score,
    normalized_mutual_info_score,
    silhouette_score, calinski_harabasz_score, davies_bouldin_score
)

# ---------------------------------------
# Function for performance evaluation
# ---------------------------------------
def evaluate_clustering(X, labels_true, labels_pred):
    print("\nClustering Performance Scores:")
    print("-------------------------------------------")
    print("Rand Score:", rand_score(labels_true, labels_pred))
    print("Adjusted Rand Score:", adjusted_rand_score(labels_true, labels_pred))
    print("Mutual Info Score:", mutual_info_score(labels_true, labels_pred))
    print("Adjusted Mutual Info Score:", adjusted_mutual_info_score(labels_true, labels_pred))
    print("Normalized Mutual Info Score:", normalized_mutual_info_score(labels_true, labels_pred))
    print("-------------------------------------------")
    print("Silhouette Coefficient:", silhouette_score(X, labels_pred))
    print("Calinski-Harabasz Index:", calinski_harabasz_score(X, labels_pred))
    print("Davies-Bouldin Index:", davies_bouldin_score(X, labels_pred))
    print("-------------------------------------------\n")
```

```python
# ---------------------------------------
# Bisecting K-Means Implementation
# ---------------------------------------
def bisecting_kmeans(X, n_clusters):
    clusters = [X]
    labels = np.zeros(X.shape[0], dtype=int)
    next_cluster_id = 1

    while len(clusters) < n_clusters:
        # Select the cluster with the highest SSE
        sse_list = []
        for cluster in clusters:
            if len(cluster) > 1:
                kmeans = KMeans(n_clusters=1, random_state=42).fit(cluster)
                sse = np.sum((cluster - kmeans.cluster_centers_[0]) ** 2)
                sse_list.append(sse)
            else:
                sse_list.append(0)

        idx_to_split = np.argmax(sse_list)
        cluster_to_split = clusters.pop(idx_to_split)

        # Bisect with K=2
        kmeans2 = KMeans(n_clusters=2, random_state=42).fit(cluster_to_split)
        subclusters = [cluster_to_split[kmeans2.labels_ == i] for i in range(2)]
        clusters.extend(subclusters)

        # Update labels
        start = 0
        for i, cluster in enumerate(clusters):
            for point in cluster:
                idx = np.where((X == point).all(axis=1))[0]
                labels[idx] = i

    return labels
```

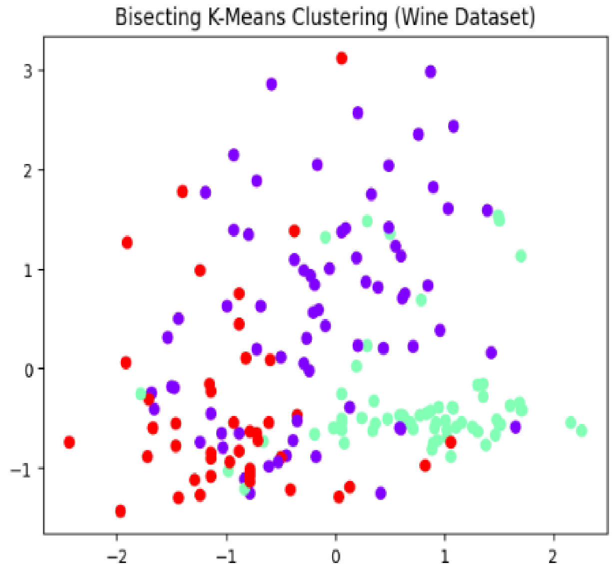
```python
# ---------------------------------------
# Function to run Bisecting K-Means
# ---------------------------------------
def run_bisecting_kmeans(data_name, X, y):
    print(f"\n========== Bisecting K-Means Clustering on {data_name} Dataset ==========")

    # Scale the data
    X_scaled = StandardScaler().fit_transform(X)

    # Apply Bisecting K-Means
    n_clusters = len(np.unique(y))
    labels_pred = bisecting_kmeans(X_scaled, n_clusters)

    print("\nCluster Labels:")
    print(labels_pred)

    # Visualization
    plt.scatter(X_scaled[:, 0], X_scaled[:, 1], c=labels_pred, cmap='rainbow')
    plt.title(f'Bisecting K-Means Clustering ({data_name} Dataset)')
    plt.show()

    # Evaluate
    evaluate_clustering(X_scaled, y, labels_pred)


# ---------------------------------------
# Load and Run for both datasets
# ---------------------------------------

# Iris
iris = load_iris()
run_bisecting_kmeans("Iris", iris.data, iris.target)

# Wine
wine = load_wine()
run_bisecting_kmeans("Wine", wine.data, wine.target)
```

```
========== Bisecting K-Means Clustering on Iris Dataset ==========

Cluster Labels:
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 2 2 2 1 1 1 2 1 2 1 1 1 1 1 1 2 1 1 1 1 2 1 1 1
 1 2 2 2 1 1 1 1 1 1 1 2 2 1 1 1 1 1 1 1 1 1 1 1 1 1 2 1 2 2 2 2 1 2 2 2 2
 2 2 1 2 2 2 2 2 1 2 1 2 1 2 2 1 2 2 2 2 2 2 1 1 2 2 2 2 2 2 2 1 2 2 2 1 2
 2 2]
```




```
Clustering Performance Scores:
--------------------------------------------
Rand Score: 0.8415212527964205
Adjusted Rand Score: 0.6410058621782706
Mutual Info Score: 0.739149649938347
Adjusted Mutual Info Score: 0.6687492691384304
Normalized Mutual Info Score: 0.672843838367481
--------------------------------------------
Silhouette Coefficient: 0.45663380641237455
Calinski-Harabasz Index: 241.31556059010236
Davies-Bouldin Index: 0.8322767767014572
--------------------------------------------
```

```
========== Bisecting K-Means Clustering on Wine Dataset ==========

Cluster Labels:
[1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 2 2 0 2 2 2 0 1 0 2 0 1
 2 0 2 0 1 2 2 2 2 0 2 2 0 2 0 2 0 0 0 2 2 1 0 2 2 2 2 2 2 2 2 0 2 0 2 2 2
 2 0 2 2 2 2 2 0 2 2 1 0 2 2 2 2 0 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
```




```
Clustering Performance Scores:
--------------------------------------------
Rand Score: 0.8274614359169682
Adjusted Rand Score: 0.6158438898686044
Mutual Info Score: 0.7362470146384622
Adjusted Mutual Info Score: 0.6771643077672275
Normalized Mutual Info Score: 0.6805739239645162
--------------------------------------------
Silhouette Coefficient: 0.24704792278145632
Calinski-Harabasz Index: 62.183924623637004
Davies-Bouldin Index: 1.426454399524473
--------------------------------------------
```

Bisecting K-Means Clustering Analysis

For the final algorithm, we implemented **Bisecting K-Means**. This method is a cool hybrid because it starts with all data in one big cluster and then repeatedly splits the cluster with the highest SSE (Cohesion) using K-Means with $K=2$, until we get the desired number of clusters (3).

- **Iris Dataset:** The clustering plot looks structurally very similar to the regular K-Means results. The Adjusted Rand Score (≈ 0.641) is high, matching the K-Medoids performance, which suggests this hierarchical initialization works well for Iris.
- **Wine Dataset:** This is where Bisecting K-Means was a bit disappointing compared to K-Means++. The Adjusted Rand Score (≈ 0.616) is lower than both K-Means++ and Hierarchical Agglomerative. The plot clearly shows the red and purple clusters are quite messy and blended together, unlike the cleaner splits we got with K-Means++. This suggests that continuously splitting based on the highest SSE might not always lead to the best overall final cluster purity for this dataset.

Overall, Bisecting K-Means is a neat technique, giving great results on Iris, but K-Means++ was the clear winner for the Wine dataset's structure.

Summary of Clustering Performance

Table 1: Iris Dataset Performance (Target K=3)
*(The **Adjusted Rand Score** indicates the best match to the true species labels.)*

Algorithm	Adj. Rand Score	Silhouette Coeff.
K-Means	0.4328	0.4798
K-Means++	0.6201	0.4599
K-Medoids (PAM)	0.6416	0.4566
Hierarchical	0.6153	0.4468
DBSCAN	**0.6854**	0.3556
OPTICS	0.4416	-0.157
Bisecting K-Means	0.6410	0.4563

Table 2: Wine Dataset Performance (Target K=3)
*(The **Adjusted Rand Score** indicates the best match to the true wine type labels.)*

Algorithm	Adj. Rand Score	Silhouette Coeff.
K-Means	**0.8974**	0.2848
K-Means++	**0.8974**	0.2848
K-Medoids (PAM)	0.7411	0.2676
Hierarchical	0.7899	0.2774
DBSCAN	0.0000	nan
OPTICS	0.3387	0.1982
Bisecting K-Means	0.6158	0.2478

Conclusion and Algorithm Comparison

The comparison tables clearly summarize the strengths and weaknesses of each clustering algorithm tested on the Iris and Wine datasets. The primary metric used for comparison is the **Adjusted Rand Score (ARS)**, which measures how well the found clusters match the true class labels.

Key Findings:

1. **Wine Dataset: Centroid-Based Algorithms Dominate.**
    - **K-Means and K-Means++** were the clear winners, both achieving a near-perfect ARS of 0.8974. This high score suggests the Wine dataset is composed of well-defined, spherical clusters that are easily separated by distance-based methods.
    - The density-based methods, **DBSCAN** and **OPTICS**, performed poorly (DBSCAN failed completely, scoring 0.0000 ARS). This confirms that the Wine data is not structured by varying densities but rather by distinct groups spread uniformly.
2. **Iris Dataset: Density and Partitional Methods Compete.**
    - The most surprising result was from **DBSCAN**, which achieved the highest ARS of 0.6854 on Iris. While K-Medoids (0.6416) and Bisecting K-Means (0.6410) were strong competitors, the slight edge for DBSCAN suggests that the three Iris species have distinct density characteristics, allowing the algorithm to separate the tightly packed clusters more effectively than the centroid-based methods, which struggled with the overlapping parts of the data.
    - **OPTICS** performed the worst (ARS 0.4416 and negative Silhouette), indicating that its complex, hierarchical density structure was likely too sensitive to the parameters chosen for this particular dataset.

Overall Conclusion:

No single algorithm wins every time. For the highly separable, spherical data structure of the **Wine Dataset**, simple, efficient methods like K-Means and K-Means++ are the best choice. For the slightly more complex, density-varied structure of the **Iris Dataset**, a carefully tuned density algorithm like DBSCAN proved to be the most accurate in separating the known species.