Clustering using unsupervised learning using Python

Step 1: Import Libraries

We need to import the essential libraries that will help us with data manipulation, clustering, and visualization.

```
# Import necessary libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.cluster import KMeans
from sklearn.datasets import load_iris
from sklearn.preprocessing import StandardScaler
```

Step 2: Load the Dataset

We'll use the **Iris dataset**, which is available in **scikit-learn**. The dataset contains features like sepal length, sepal width, petal length, and petal width, which we will use for clustering.

```
# Load the Iris dataset
iris = load_iris()
X = iris.data  # Features (sepal length, sepal width, petal length, petal width)
y = iris.target  # True labels (species)
```

Step 3: Data Preprocessing

Before applying the K-Means algorithm, it's essential to **standardize the data** to ensure that all features are on the same scale. K-Means is sensitive to feature scaling because it uses distance metrics.

```
# Standardize the features (mean=0, variance=1)
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)
# Check the standardized data
print("Standardized data:\n", X_scaled[:5])  # Display the first 5 rows of standardized data
```

Step 4: Apply K-Means Clustering

We will apply the **K-Means algorithm** to the data. For this example, we'll assume there are 3 clusters, as there are 3 species in the Iris dataset.

```
# Apply KMeans Clustering
kmeans = KMeans(n_clusters=3, random_state=42)  # 3 clusters for 3 species
kmeans.fit(X_scaled)
# Get the cluster labels for each data point
y_kmeans = kmeans.labels_
# Print the cluster centroids and the first 10 labels
print("Cluster Centroids:\n", kmeans.cluster_centers_)
print("First 10 Cluster Labels:", y_kmeans[:10])
```

- **Cluster Centroids**: The centers of the 3 clusters.
- **Cluster Labels**: The label (cluster assignment) for each data point in the dataset.

Step 5: Visualizing the Clusters

To better understand how the algorithm has clustered the data, we will visualize the clusters using a scatter plot. We will plot the first two features: **sepal length** and **sepal width**.

```
# Add the cluster labels to the DataFrame for visualization
iris_df = pd.DataFrame(X_scaled, columns=iris.feature_names)
iris_df['cluster'] = y_kmeans
# Plot the clusters
plt.figure(figsize=(8, 6))
sns.scatterplot(x=iris_df[iris.feature_names[0]],
            y=iris_df[iris.feature_names[1]],
            hue='cluster', data=iris_df, palette="Set1")
plt.title("K-Means Clustering (Sepal Length vs Sepal Width)")
plt.xlabel(iris.feature_names[0])
plt.ylabel(iris.feature_names[1])
plt.legend(title='Cluster')
plt.show()
```

Step 6: Evaluate the Clustering

Since we know the true labels of the dataset, we can evaluate how well the clustering performed by comparing the predicted clusters with the true labels.

a) Confusion Matrix

We can use a confusion matrix to compare the predicted cluster labels with the true labels.

```
from sklearn.metrics import confusion_matrix
import seaborn as sns
# Create the confusion matrix
cm = confusion_matrix(y, y_kmeans)
# Plot the confusion matrix
plt.figure(figsize=(6, 5))
```

```
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues',
xticklabels=iris.target_names, yticklabels=iris.target_names)
plt.title("Confusion Matrix for K-Means Clustering")
plt.xlabel("Predicted Labels")
plt.ylabel("True Labels")
plt.show()
```

b) Adjusted Rand Index (ARI)

The **Adjusted Rand Index (ARI)** is a measure of similarity between the predicted clusters and the true labels. It accounts for random chance and gives a value between -1 and 1, where 1 means perfect agreement.

```
from sklearn.metrics import adjusted_rand_score
# Compute the ARI
ari = adjusted_rand_score(y, y_kmeans)
print(f"Adjusted Rand Index (ARI): {ari:.2f}")
```

Step 7: Finding the Optimal Number of Clusters (Elbow Method)

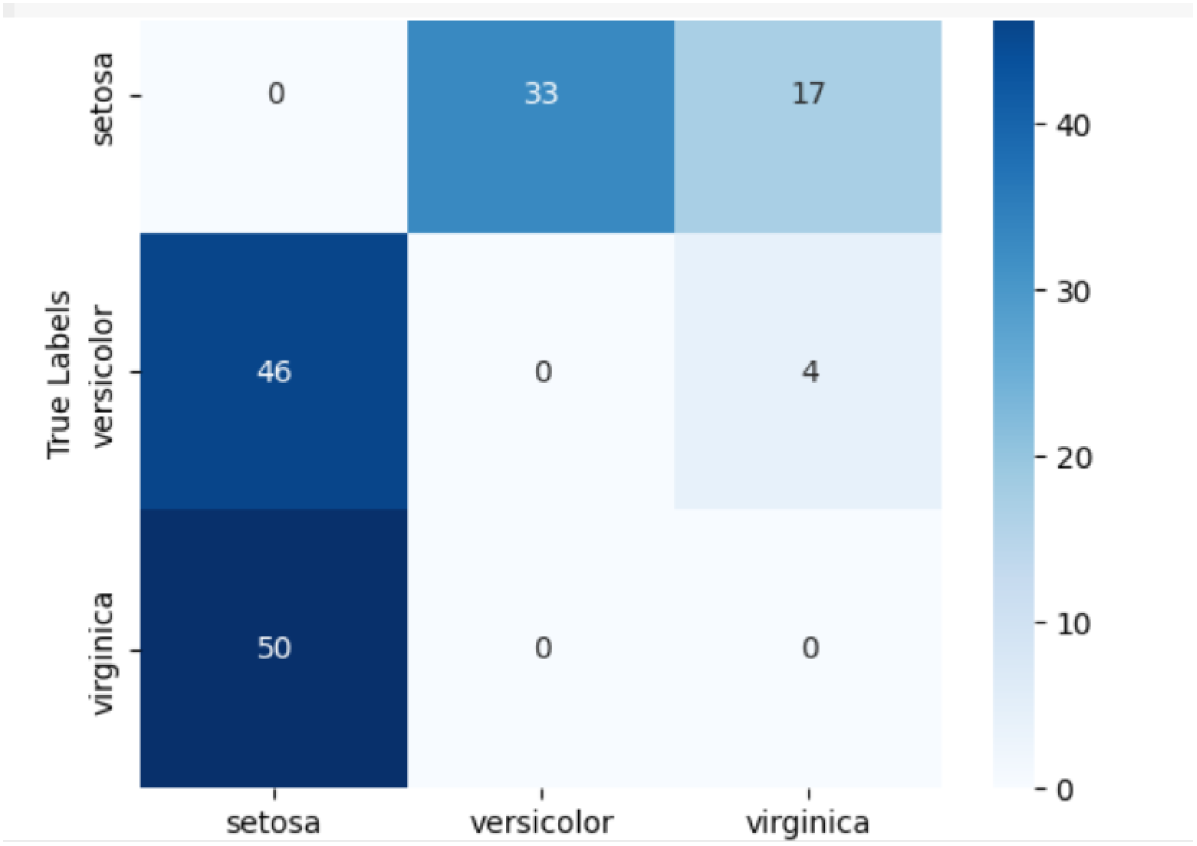
If you do not know the optimal number of clusters in advance, you can use the **Elbow Method** to determine the ideal number of clusters. The idea is to plot the **inertia** (within-cluster sum of squared distances) against different values of k and look for the "elbow" point, where inertia starts decreasing more slowly.

```
# Elbow Method to find optimal k
inertia = []
k_range = range(1, 11)
for k in k_range:
    kmeans = KMeans(n_clusters=k, random_state=42)
    kmeans.fit(X_scaled)
    inertia.append(kmeans.inertia_)
```

```
# Plot the elbow graph
plt.figure(figsize=(8, 6))
plt.plot(k_range, inertia, marker='o')
plt.title("Elbow Method for Optimal k")
plt.xlabel("Number of Clusters (k)")
plt.ylabel("Inertia")
plt.show()
```

- Look for the **elbow point**, where the rate of decrease in inertia slows down. The k corresponding to this point is the optimal number of clusters.

OUTPUT:



Step 8: Conclusion

This completes the steps for performing K-Means clustering on the Iris dataset using Python. Below is a summary of what we accomplished:

1. **Loaded the Iris dataset.**

2. **Standardized the data** to ensure proper scaling for K-Means.
3. **Applied K-Means clustering** and visualized the results.
4. **Evaluated the clustering** using confusion matrix and ARI.
5. **Determined the optimal number of clusters** using the Elbow Method.