Experiment – 8

K-means Clustering: Implementing the K-means clustering algorithm using Scikit-learn

Date Set: *The dataset contains measurements of four features of iris flowers: Sepal Length, Sepal Width, Petal Length, and Petal Width. These measurements are provided in centimeters (cm). Additionally, each data point is associated with a specific species of iris flower. Each row in the dataset represents a single observation or measurement of an iris flower. The dataset contains a total of 30 observations.*

Objective:

The objective of this lab is to learn how to perform K-means clustering using Scikit-learn library in Python.

Python Program:

Importing the libraries

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans
```

Note:

- **import matplotlib.pyplot as plt**: This line imports the **pyplot** module from the **matplotlib** library and gives it an alias **plt**. Matplotlib is a plotting library for Python, and **pyplot** provides a MATLAB-like interface to matplotlib.
- **from sklearn.cluster import KMeans**: This line imports the KMeans class from the **cluster** module of the Scikit-learn library (**sklearn**). Scikit-learn is a popular machine learning library for Python, and the **cluster** module contains various clustering algorithms, including KMeans.

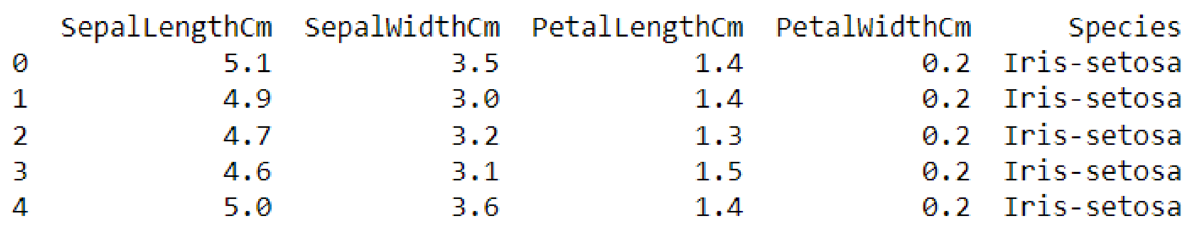
Importing the dataset

```
# Load the dataset
data = pd.read_csv(r'C:\Users\KUNAL
KUNDU\OneDrive\Desktop\GHRCEM\...\Experiment-8.csv')
# Display the first few rows of the dataset
print(data.head())
```

Note:

- **print(data.head())**: This line prints the first few rows of the DataFrame **data**. The **head()** function is used to retrieve the first few rows (by default, the first five rows) of the DataFrame.

Output:



```
   SepalLengthCm  SepalWidthCm  PetalLengthCm  PetalWidthCm      Species
0            5.1           3.5            1.4           0.2  Iris-setosa
1            4.9           3.0            1.4           0.2  Iris-setosa
2            4.7           3.2            1.3           0.2  Iris-setosa
3            4.6           3.1            1.5           0.2  Iris-setosa
4            5.0           3.6            1.4           0.2  Iris-setosa
```

Preprocess the data

```
# Extract features from the dataset
X = data.iloc[:, :-1].values

# Feature scaling (optional but recommended for K-means)
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)
```

Note:
This code is responsible for preprocessing the dataset before applying the K-means clustering algorithm. Here is the detailed breakdown of the code:

1. **Extracting Features:**
    - **X = data.iloc[:, :-1].values**: This line extracts the features from the dataset. The **iloc** function is used to select data by position. Here, **[:, :-1]** selects all rows and all columns except the last one. This assumes that the last column contains the target variable or labels, which we want to exclude from the features. The **.values** attribute converts the selected data into a NumPy array, which is suitable for further processing.
2. **Feature Scaling:**
    - **from sklearn.preprocessing import StandardScaler**: This line imports the **StandardScaler** class from the **preprocessing** module of Scikit-learn. **StandardScaler** is a preprocessing technique used to standardize features by removing the mean and scaling to unit variance.
    - **scaler = StandardScaler()**: This line creates an instance of the **StandardScaler** class. This scaler object will be used to scale the features.
    - **X_scaled = scaler.fit_transform(X)**: This line scales the features using the **fit_transform** method of the scaler object. This method computes the mean and standard deviation of each feature in the dataset (**X**) and then scales the features accordingly. The scaled features are stored in the **X_scaled** variable. This step is optional but recommended for K-means clustering, as it ensures that all features have the same scale, which can improve the performance of the algorithm.

Determine the optimal number of clusters using the Elbow Method

```
# Initialize an empty list to store the within-cluster sum of
squares (WCSS) for different cluster numbers
wcss = []
```

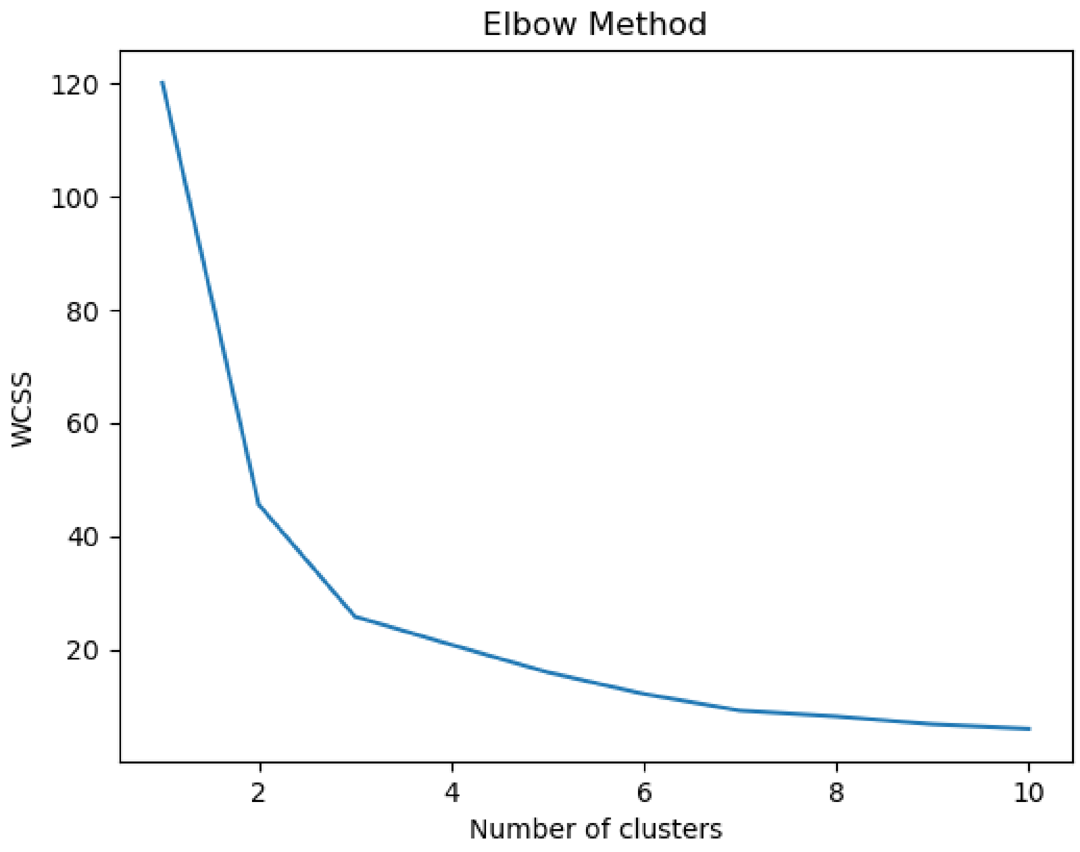
```
# Try cluster numbers from 1 to 10
for i in range(1, 11):
    kmeans    =    KMeans(n_clusters=i,    init='k-means++', random_state=42)
    kmeans.fit(X_scaled)
    # Append the WCSS to the list
    wcss.append(kmeans.inertia_)

# Plot the Elbow Method graph
plt.plot(range(1, 11), wcss)
plt.title('Elbow Method')
plt.xlabel('Number of clusters')
plt.ylabel('WCSS')
plt.show()
```

Note:
This code is used to determine the optimal number of clusters for K-means clustering using the Elbow Method. Here is the detailed breakdown of the code:

1. **Initialize an Empty List:**
    - **wcss = []**: This line initializes an empty list named **wcss** which will store the within-cluster sum of squares (WCSS) for different cluster numbers.
2. **Try Different Numbers of Clusters:**
    - **for i in range(1, 11):**: This loop iterates over cluster numbers from 1 to 10.
3. **Instantiate and Fit KMeans:**
    - **kmeans = KMeans(n_clusters=i, init='k-means++', random_state=42)**: This line creates a KMeans object with the specified number of clusters (**n_clusters=i**).
        - **init='k-means++'** is used to initialize centroids in a smarter way which can improve convergence speed.
        - **random_state=42** is used to ensure reproducibility of the results.
    - **kmeans.fit(X_scaled)**: This line fits the KMeans model to the scaled feature data (**X_scaled**). It assigns each data point to its nearest centroid and computes the within-cluster sum of squares (WCSS).
4. **Compute and Store WCSS:**
    - **wcss.append(kmeans.inertia_)**: After fitting the model, the WCSS (inertia) of the fitted model is computed and appended to the **wcss** list. WCSS measures the compactness of the clusters. Lower WCSS indicates better clustering.
5. **Plot the Elbow Method Graph:**
    - **plt.plot(range(1, 11), wcss)**: This line plots the number of clusters on the x-axis (from 1 to 10) and the corresponding WCSS values on the y-axis.
    - **plt.title('Elbow Method')**: Sets the title of the plot to "Elbow Method".
    - **plt.xlabel('Number of clusters')**: Sets the label for the x-axis as "Number of clusters".
    - **plt.ylabel('WCSS')**: Sets the label for the y-axis as "WCSS".
    - **plt.show()**: This line displays the plot showing the relationship between the number of clusters and the WCSS values. The "elbow" in the plot indicates the optimal number of clusters, where the rate of decrease in WCSS slows down.

This method helps in identifying a suitable number of clusters for K-means clustering.

`Output:`



Fit K-means to the dataset with the optimal number of clusters

```
# From the Elbow Method graph, we can see that the optimal number of clusters is 3
kmeans = KMeans(n_clusters=3, init='k-means++', random_state=42)
y_kmeans = kmeans.fit_predict(X_scaled)
```

Note:
This code is responsible for fitting the K-means clustering algorithm to the dataset with the optimal number of clusters, which is determined using the Elbow Method. Here is the detailed breakdown of the code:

1. **Creating KMeans Object:**
    - **kmeans = KMeans(n_clusters=3, init='k-means++', random_state=42)**: This line creates a KMeans object with the specified parameters.
        - **n_clusters=3**: Specifies the number of clusters to be used. In this case, we have determined from the Elbow Method that the optimal number of clusters is 3.

        - **init='k-means++'**: Specifies the method for initializing centroids. 'k-means++' is a smart initialization method that initializes centroids in a way that can speed up convergence.
        - **random_state=42**: Sets the random seed for reproducibility. This ensures that the random initialization of centroids is consistent across runs.
2. **Fitting and Predicting Clusters:**
    - **y_kmeans = kmeans.fit_predict(X_scaled)**: This line fits the KMeans model to the scaled feature data **X_scaled** and assigns each data point to a cluster. The **fit_predict** method computes cluster centers and predicts the cluster index for each sample. The resulting **y_kmeans** array contains the cluster labels assigned to each data point.

After executing this code snippet, **y_kmeans** will contain the cluster labels assigned by the K-means algorithm to each data point in the dataset. These cluster labels can be used for further analysis or visualization of the clustered data.

Visualize the clusters

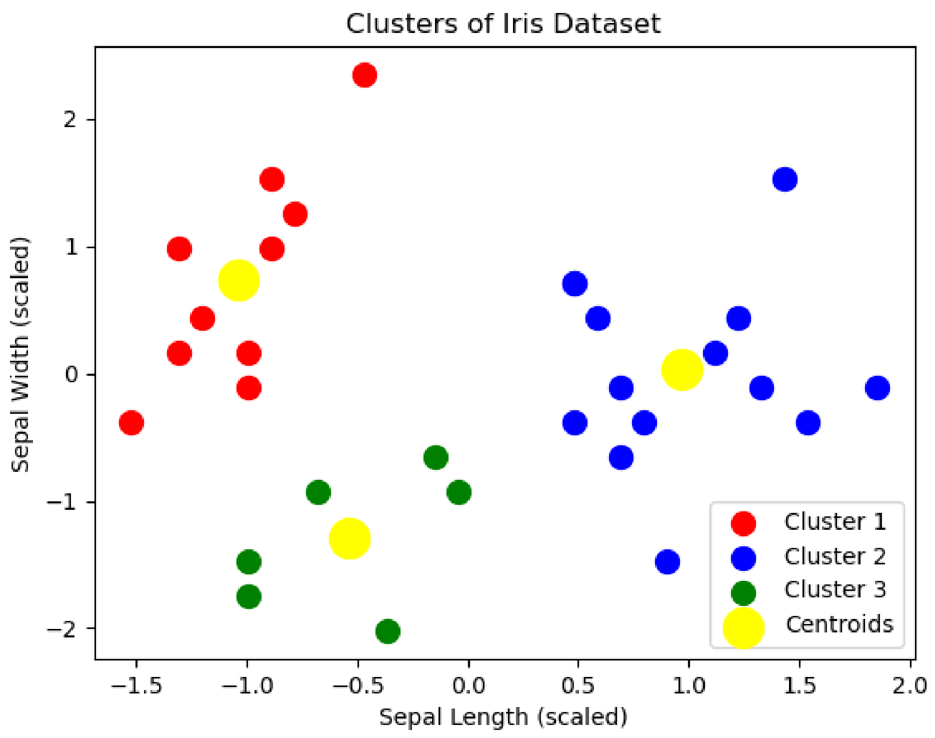
```
# Visualize the clusters on the first two features (Sepal Len
gth and Sepal Width)
plt.scatter(X_scaled[y_kmeans == 0, 0], X_scaled[y_kmeans ==
0, 1], s=100, c='red', label='Cluster 1')
plt.scatter(X_scaled[y_kmeans == 1, 0], X_scaled[y_kmeans ==
1, 1], s=100, c='blue', label='Cluster 2')
plt.scatter(X_scaled[y_kmeans == 2, 0], X_scaled[y_kmeans ==
2, 1], s=100, c='green', label='Cluster 3')

# Plot the centroids of the clusters
plt.scatter(kmeans.cluster_centers_[:, 0], kmeans.cluster_cen
ters_[:, 1], s=300, c='yellow', label='Centroids')

plt.title('Clusters of Iris Dataset')
plt.xlabel('Sepal Length (scaled)')
plt.ylabel('Sepal Width (scaled)')
plt.legend()
plt.show()
```

Note:
This code generates a scatter plot where each data point is colored according to its cluster assignment, and the centroids of the clusters are marked with yellow markers. It provides a visual representation of how the data points are grouped into clusters based on their Sepal Length and Sepal Width features. Here is the detailed breakdown of the code:

1. **Visualizing Data Points for Each Cluster:**
    - **plt.scatter(X_scaled[y_kmeans == 0, 0], X_scaled[y_kmeans == 0, 1], s=100, c='red', label='Cluster 1')**: This line creates a scatter plot of data points belonging to Cluster 1.
        - **X_scaled[y_kmeans == 0, 0]** selects the scaled Sepal Length values of data points belonging to Cluster 1.

    - **X_scaled[y_kmeans == 0, 1]** selects the scaled Sepal Width values of data points belonging to Cluster 1.
    - **s=100** sets the size of the markers to 100.
    - **c='red'** sets the color of the markers to red.
    - **label='Cluster 1'** provides a label for the data points in the legend.
  - Similar lines follow for Clusters 2 and 3, with different colors and labels.
2. **Visualizing Centroids:**
  - **plt.scatter(kmeans.cluster_centers_[:, 0], kmeans.cluster_centers_[:, 1], s=300, c='yellow', label='Centroids')**: This line creates scatter plot markers for the centroids of each cluster.
    - **kmeans.cluster_centers_[:, 0]** selects the scaled Sepal Length values of the centroids.
    - **kmeans.cluster_centers_[:, 1]** selects the scaled Sepal Width values of the centroids.
    - **s=300** sets the size of the centroid markers to 300.
    - **c='yellow'** sets the color of the centroid markers to yellow.
    - **label='Centroids'** provides a label for the centroids in the legend.
3. **Plot Customization:**
  - **plt.title('Clusters of Iris Dataset')**: Sets the title of the plot to "Clusters of Iris Dataset".
  - **plt.xlabel('Sepal Length (scaled)')**: Sets the label for the x-axis as "Sepal Length (scaled)".
  - **plt.ylabel('Sepal Width (scaled)')**: Sets the label for the y-axis as "Sepal Width (scaled)".
  - **plt.legend()**: Displays the legend on the plot.
  - **plt.show()**: Displays the plot.

Output:


Clusters of Iris Dataset
Sepal Width (scaled)
Sepal Length (scaled)
Cluster 1
Cluster 2
Cluster 3
Centroids