DATA SCIENCE
GROUP-3
PHASE_4 SUBMISSION

TEAM:
Sowjanya
Shrirekkha
Kavinaya
Priyanka
Deepshikaa

Feature Engineering:

Extracting the features like annual income of users.
Divide the annual income into bins or categories. This can help create a more granular view of income ranges. Like Low Income,Medium Income and High Income.

```
num_bins=3
bin_labels = ['Low Income', 'Medium Income', 'High Income']
df['IncomeCategory'] = pd.cut(df['Annual Income (k$)'], bins=num_bins, labels=bin_labels)
```

- Create 3 bins and label them as ‘low income’ , ‘medium income’,’high income’.
- Define the income category
- `pd.cut` is a function provided by the Pandas library in Python, and it is used for binning or categorizing data into discrete intervals.
- Here `df['Annual Income (k$)']` is the input array
- Num_bins is the bins and bin_lables is the labels

Applying clustering algorithms:

CODE:

```
[ ]  from sklearn.cluster import KMeans
     wcss = []
     for i in range(1, 11):
         kmeans = KMeans(n_clusters = i, init = 'k-means++', random_state = 42)
         kmeans.fit(X)
         wcss.append(kmeans.inertia_)
     plt.plot(range(1, 11), wcss)
     plt.title('The Elbow Method')
     plt.xlabel('Number of clusters')
     plt.ylabel('WCSS')
     plt.show()
```

1)Importing necessary libraries:

from sklearn.cluster import KMeans

This line imports the KMeans clustering algorithm from the scikit-learn library.

2)Initializing an empty list for the Within-Cluster-Sum-of-Squares (WCSS):

wcss = []

WCSS is a measure of the variance within the clusters. The goal is to find the number of clusters (K) that minimize the WCSS.

3)Looping through a range of values for K (number of clusters):

for i in range(1, 11):

This loop iterates from 1 to 10, trying different values for K.

4)Creating a KMeans instance for each value of K:

kmeans = KMeans(n_clusters=i, init='k-means++', random_state=42)

- `n_clusters=i`: This sets the number of clusters to the current value of `i`.

- `init='k-means++'`: This specifies the initialization method for cluster centers, which is 'k-means++'. This method initializes the cluster centroids in a smart way to speed up convergence.
- `random_state=42`: This sets a random seed for reproducibility. The same seed will produce the same results each time you run the code.

5)Fitting the KMeans model to the data:

```
kmeans.fit(X)
```

`X` should be your data, which you didn't include in the code snippet. The KMeans algorithm is applied to the data for the current value of K.

6)Calculating and storing the WCSS for each K:

```
wcss.append(kmeans.inertia_)
```

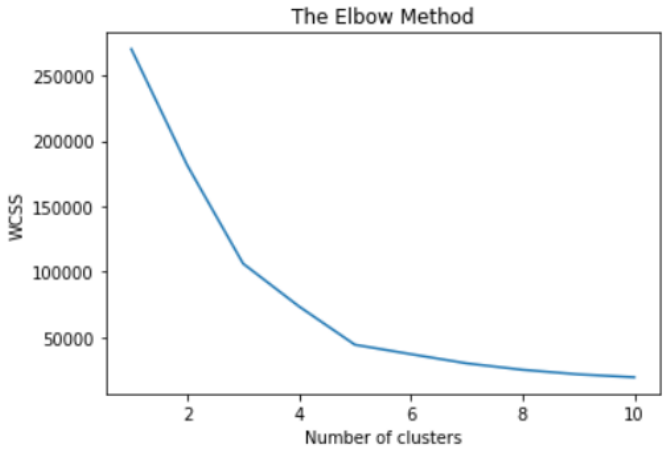
The `inertia_` attribute of the KMeans object contains the WCSS value for the current clustering. It is the sum of squared distances from each point in the dataset to its assigned cluster centroid. This value is added to the `wcss` list.

7)Plotting the WCSS values for different values of K:

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

- `plt.plot(range(1, 11), wcss)`: This line creates a line plot with the number of clusters on the x-axis (from 1 to 10) and the corresponding WCSS values on the y-axis.
- `plt.title('The Elbow Method')`: This sets the title of the plot to 'The Elbow Method'.
- `plt.xlabel('Number of clusters')`: This labels the x-axis.
- `plt.ylabel('WCSS')`: This labels the y-axis.
- `plt.show()`: This displays the plot.

A plot displaying the WCSS for various values of K is the end result. The ideal number of clusters is usually found at the "elbow" position in the plot. The elbow point denotes the point at which the WCSS rate of decline slows down, suggesting that there are fewer benefits to reducing variance within clusters. By using the code, you may select the right number of clusters for your K-means clustering algorithm and visually identify this point.



Training the K-Means model on the dataset:

```
kmeans = KMeans(n_clusters = 5, init = 'k-means++', random_state = 42)
y_kmeans = kmeans.fit_predict(X)
```

1)Creating a KMeans instance with 5 clusters:

kmeans = KMeans(n_clusters=5, init='k-means++', random_state=42)

The following parameters are being used to create a KMeans object in this instance:

- The argument n_clusters=5 indicates that you wish to form 5 clusters. Every cluster will own a unique centroid.
- The initialization method for cluster centroids is changed to 'k-means++' by setting init='k-means++'. This is a more sophisticated initialization method.
- random_state=42: This ensures reproducibility by setting a random seed, so when you execute the code again, you will receive the same outcomes.

2)Clustering the data and obtaining cluster labels:

y_kmeans = kmeans.fit_predict(X)

- kmeans.fit_predict(X): This line places each data point in one of the five clusters after fitting the KMeans model to your data X. The cluster labels for every data point in X are contained in

the y_kmeans array that is produced. Every component of y_kmeans represents a data point and denotes the cluster to which it belongs.

We have now applied K-means clustering to your data with 5 clusters by running these lines of code, and y_kmeans now holds the cluster labels for each data point in X. These cluster labels can be applied to a number of activities, such predicting outcomes based on the clusters or examining and visualising the structure of the data.

Interpretation and Visualization:

CODE:

Visualising the clusters

```
[ ]  plt.scatter(X[y_kmeans == 0, 0], X[y_kmeans == 0, 1], s = 100, c = 'red', label = 'Cluster 1')
     plt.scatter(X[y_kmeans == 1, 0], X[y_kmeans == 1, 1], s = 100, c = 'blue', label = 'Cluster 2')
     plt.scatter(X[y_kmeans == 2, 0], X[y_kmeans == 2, 1], s = 100, c = 'green', label = 'Cluster 3')
     plt.scatter(X[y_kmeans == 3, 0], X[y_kmeans == 3, 1], s = 100, c = 'cyan', label = 'Cluster 4')
     plt.scatter(X[y_kmeans == 4, 0], X[y_kmeans == 4, 1], s = 100, c = 'magenta', label = 'Cluster 5')
     plt.scatter(kmeans.cluster_centers_[:, 0], kmeans.cluster_centers_[:, 1], s = 300, c = 'yellow', label = 'Centroids')
     plt.title('Clusters of customers')
     plt.xlabel('Annual Income (k$)')
     plt.ylabel('Spending Score (1-100)')
     plt.legend()
     plt.show()
```

1)Scatter Plot for Data Points in Each Cluster:

plt.scatter(X[y_kmeans == 0, 0], X[y_kmeans == 0, 1], s=100, c='red', label='Cluster 1')
plt.scatter(X[y_kmeans == 1, 0], X[y_kmeans == 1, 1], s=100, c='blue', label='Cluster 2')
plt.scatter(X[y_kmeans == 2, 0], X[y_kmeans == 2, 1], s=100, c='green', label='Cluster 3')
plt.scatter(X[y_kmeans == 3, 0], X[y_kmeans == 3, 1], s=100, c='cyan', label='Cluster 4')
plt.scatter(X[y_kmeans == 4, 0], X[y_kmeans == 4, 1], s=100, c='magenta', label='Cluster 5')

For every cluster, scatter plots are produced by these lines. Plotting data points in various colours according to their associated cluster is accomplished using the plt.scatter function. The markers' size and colour are set by the s and c parameters, respectively. A legend is created using the label, and each cluster is represented by a distinct colour.

2)Plotting Cluster Centroids:

plt.scatter(kmeans.cluster_centers_[:, 0], kmeans.cluster_centers_[:, 1], s=300, c='yellow', label='Centroids')

This line plots the cluster centroids (center points of each cluster) as large yellow markers. The `kmeans.cluster_centers_` attribute provides the coordinates of the centroids.

3)Adding Titles and Labels:

plt.title('Clusters of customers')
plt.xlabel('Annual Income (k$)')
plt.ylabel('Spending Score (1-100)')

These lines set the title and labels for the x and y axes of the plot.

4)Displaying the Legend:

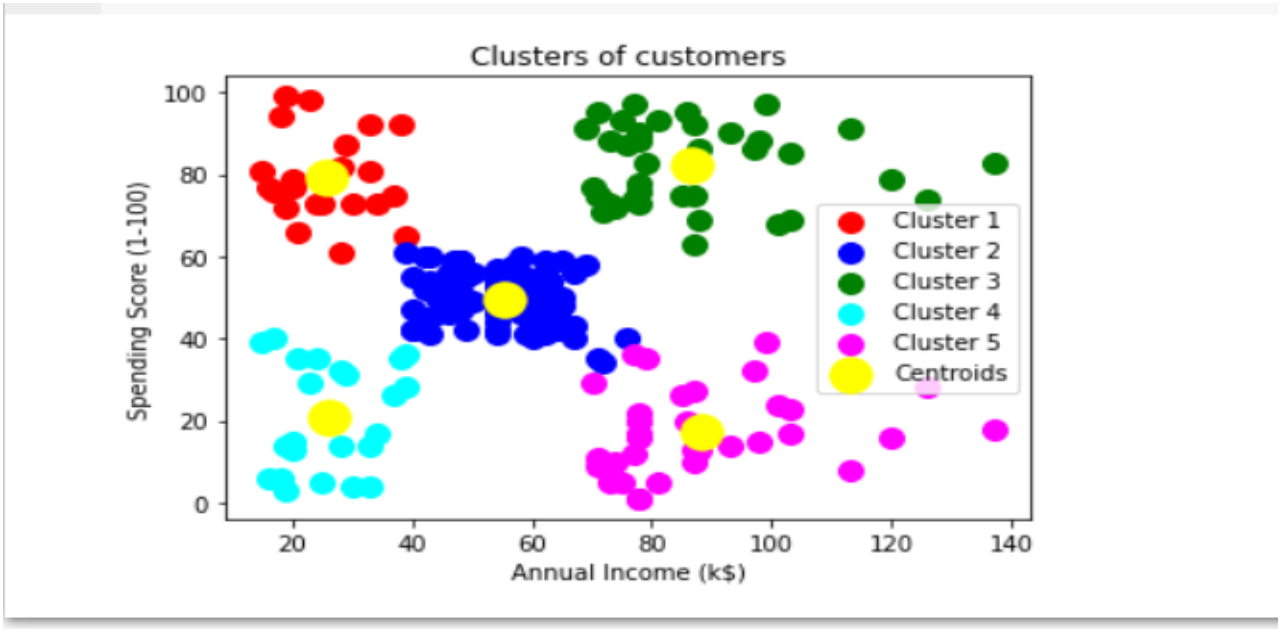
plt.legend()

This line displays the legend on the plot, indicating which color corresponds to each cluster and the centroids.

5)Displaying the Plot:

plt.show()

This line shows the plot with the clustered data points and centroids.


Clusters of customers
Spending Score (1-100)
Annual Income (k$)
Cluster 1
Cluster 2
Cluster 3
Cluster 4
Cluster 5
Centroids
0
20
40
60
80
100
120
140