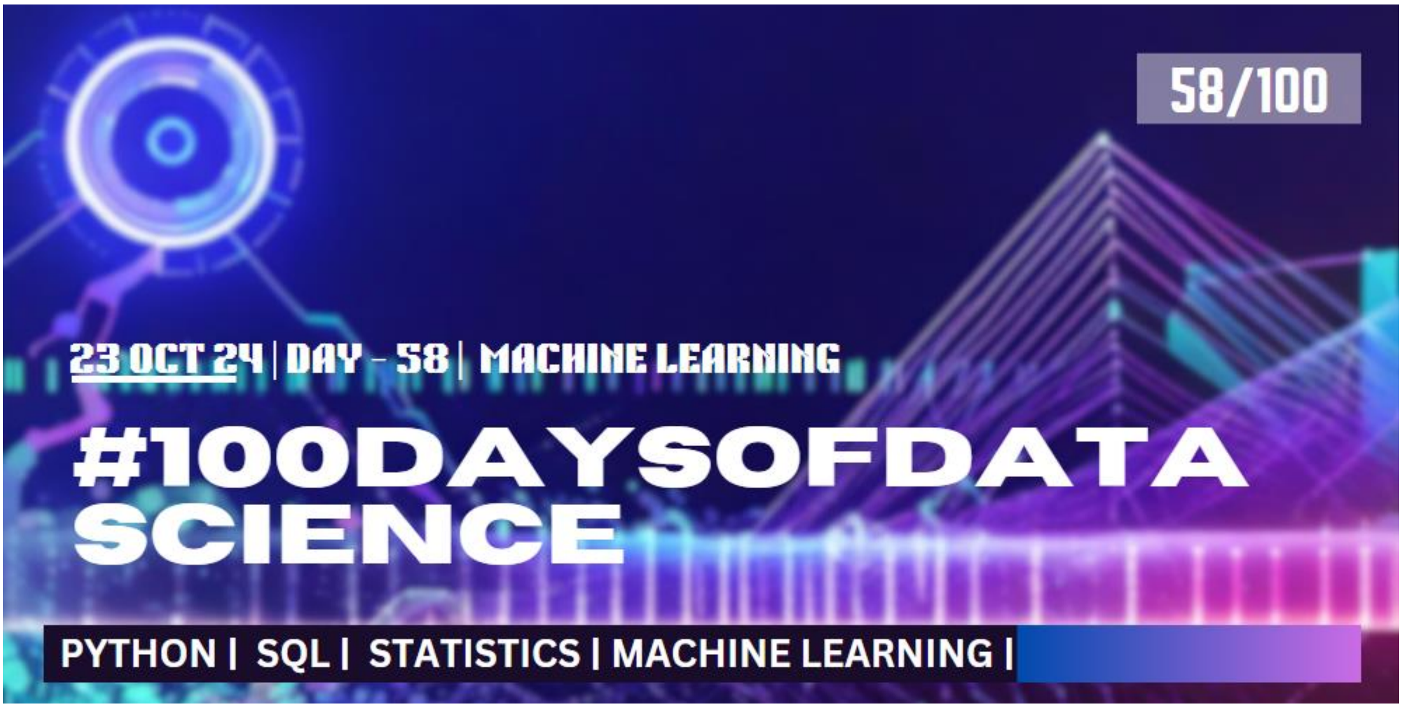
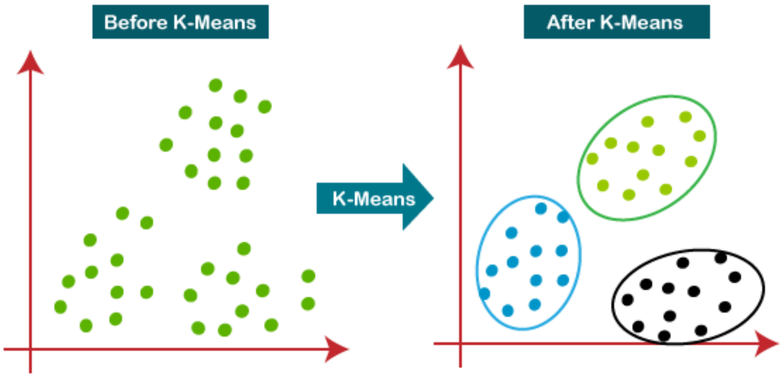


K-Means Clustering



- **K-Means Clustering** is a popular unsupervised learning algorithm used to partition a dataset into **K distinct, non-overlapping clusters**. The algorithm aims to group similar data points together, minimizing intra-cluster variance and maximizing inter-cluster variance.

- **Steps of the K-Means Algorithm**:
  1. **Initialization**: Choose K initial centroids randomly from the dataset.
  2. **Cluster Assignment**: Assign each data point to the nearest centroid using a distance measure, typically Euclidean distance.
  3. **Centroid Update**: For each cluster, calculate the mean of the data points assigned to it and update the centroid.
  4. **Repeat**: Repeat the assignment and update steps until the centroids no longer change, or a specified number of iterations is reached.

- **K Elbow Method**:
  The **K Elbow Method** is a heuristic used to determine the optimal number of clusters (K) in K-Means clustering. This method helps overcome the challenge of pre-specifying K, which is a common drawback of K-Means.
  - **How it Works**:
    1. Run K-Means with a range of K values (e.g., K = 1 to 10).
    2. For each K, compute the **sum of squared distances (SSE)** between data points and their assigned cluster centroids.

3. Plot the SSE against K values.
4. The **elbow point** on the curve (where the SSE starts to level off) is considered the optimal K. This is where adding more clusters does not significantly improve clustering performance.

- **Intuition**: The elbow represents the point where increasing K further provides diminishing returns in terms of reducing the SSE. Prior to this point, adding more clusters greatly improves clustering, while after this point, the improvement is minimal.
- **Example**: Suppose you plot the SSE for K values from 1 to 10. The plot shows a sharp decrease in SSE for K = 1 to 4, but after K = 4, the curve flattens. This suggests that K = 4 is the optimal number of clusters for your data.

- **Advantages**:
  - **Simplicity**: K-Means is easy to implement and computationally efficient.
  - **Scalability**: Works well with large datasets.
  - **Speed**: Converges relatively quickly, making it suitable for real-time applications.
  - **K Elbow Method**: Helps overcome the challenge of choosing K, offering a visual approach to selecting the optimal number of clusters.

- **Disadvantages**:
  - **Choice of K**: Although the elbow method helps, the process can still be subjective, especially in cases where the elbow is not clearly defined.
  - **Sensitive to Initialization**: Different initial centroids can lead to different results.
  - **Sensitive to Outliers**: Outliers can distort centroid calculations and negatively affect clustering performance.
  - **Assumes Spherical Clusters**: K-Means works best when clusters are spherical and evenly sized, which might not hold for all datasets.
  - **Requires Normalized Data**: The algorithm relies on Euclidean distance, which means it performs better when features are normalized.

- **Applications**:
  - **Image Compression**: K-Means can be used in image segmentation by clustering similar pixels together.
  - **Customer Segmentation**: Businesses can use K-Means to segment customers based on purchasing behavior.
  - **Anomaly Detection**: K-Means can identify outliers in datasets, helping in fraud detection or network security.
  - **Document Clustering**: K-Means is commonly used in text mining to cluster documents by topic or content.

- **Limitations**:
  - **Scalability Issues**: With very large datasets, K-Means might become computationally expensive unless optimized techniques like mini-batch K-Means are used.
  - **Shape of Clusters**: It may struggle with non-spherical clusters or clusters of varying density.
  - **Outliers**: The algorithm can be skewed by outliers that affect the calculation of centroids.

In conclusion, **K-Means** is a widely used clustering algorithm, and the **K Elbow Method** provides a valuable heuristic for selecting the number of clusters. While K-Means is efficient and effective for many applications, careful consideration of parameters like K and sensitivity to outliers is essential for achieving optimal results.

Notebook

October 21, 2024

[91]:
```python
### Importing Libraries
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import plotly.express as px
import plotly.figure_factory as ff
from sklearn.metrics import confusion_matrix, accuracy_score,␣
↪classification_report
from sklearn.metrics import mean_squared_error, r2_score
import warnings
warnings.filterwarnings('ignore')
```

[93]:
```python
### Import the Dataset
df = pd.read_csv(r"C:\Users\Zahid.Shaikh\100days\58\Mall_Customers.
↪csv",header=0)
df.head()
```

[93]:

	CustomerID	Gender	Age	Annual Income (k$)	Spending Score (1-100)
0	1	Male	19	15	39
1	2	Male	21	15	81
2	3	Female	20	16	6
3	4	Female	23	16	77
4	5	Female	31	17	40

[95]:
```python
df.shape ### Checking Shape
```

[95]: (200, 5)

[97]:
```python
df.describe() ### Get information of the Dataset
```

[97]:

	CustomerID	Age	Annual Income (k$)	Spending Score (1-100)
count	200.000000	200.000000	200.000000	200.000000
mean	100.500000	38.850000	60.560000	50.200000
std	57.879185	13.969007	26.264721	25.823522
min	1.000000	18.000000	15.000000	1.000000
25%	50.750000	28.750000	41.500000	34.750000
50%	100.500000	36.000000	61.500000	50.000000

```
75%    150.250000  49.000000           78.000000                73.000000
max    200.000000  70.000000          137.000000                99.000000
```

[99]:
```python
df.columns ### Checking Columns
```

[99]:
```
Index(['CustomerID', 'Gender', 'Age', 'Annual Income (k$)',
       'Spending Score (1-100)'],
      dtype='object')
```

[101]:
```python
df.info() ### Checking Information About a DataFrame
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 200 entries, 0 to 199
Data columns (total 5 columns):
 #   Column                  Non-Null Count  Dtype
---  ------                  --------------  -----
 0   CustomerID              200 non-null    int64
 1   Gender                  200 non-null    object
 2   Age                     200 non-null    int64
 3   Annual Income (k$)      200 non-null    int64
 4   Spending Score (1-100)  200 non-null    int64
dtypes: int64(4), object(1)
memory usage: 7.9+ KB
```

[103]:
```python
df.isnull().sum() ### Checking Null Values in the Data
```

[103]:
```
CustomerID                0
Gender                    0
Age                       0
Annual Income (k$)        0
Spending Score (1-100)    0
dtype: int64
```

[105]:
```python
df1 = pd.DataFrame.copy(df)
df1.shape
```

[105]:
```
(200, 5)
```

[107]:
```python
for i in df1.columns:
    print({i:df1[i].unique()}) ### Checking Unique values in each columns
```

```
{'CustomerID': array([  1,   2,   3,   4,   5,   6,   7,   8,   9,  10,  11,
12,  13,
        14,  15,  16,  17,  18,  19,  20,  21,  22,  23,  24,  25,  26,
        27,  28,  29,  30,  31,  32,  33,  34,  35,  36,  37,  38,  39,
        40,  41,  42,  43,  44,  45,  46,  47,  48,  49,  50,  51,  52,
        53,  54,  55,  56,  57,  58,  59,  60,  61,  62,  63,  64,  65,
        66,  67,  68,  69,  70,  71,  72,  73,  74,  75,  76,  77,  78,
        79,  80,  81,  82,  83,  84,  85,  86,  87,  88,  89,  90,  91,
```

```
         92,  93,  94,  95,  96,  97,  98,  99, 100, 101, 102, 103, 104,
        105, 106, 107, 108, 109, 110, 111, 112, 113, 114, 115, 116, 117,
        118, 119, 120, 121, 122, 123, 124, 125, 126, 127, 128, 129, 130,
        131, 132, 133, 134, 135, 136, 137, 138, 139, 140, 141, 142, 143,
        144, 145, 146, 147, 148, 149, 150, 151, 152, 153, 154, 155, 156,
        157, 158, 159, 160, 161, 162, 163, 164, 165, 166, 167, 168, 169,
        170, 171, 172, 173, 174, 175, 176, 177, 178, 179, 180, 181, 182,
        183, 184, 185, 186, 187, 188, 189, 190, 191, 192, 193, 194, 195,
        196, 197, 198, 199, 200], dtype=int64)}
{'Gender': array(['Male', 'Female'], dtype=object)}
{'Age': array([19, 21, 20, 23, 31, 22, 35, 64, 30, 67, 58, 24, 37, 52, 25, 46,
54,
       29, 45, 40, 60, 53, 18, 49, 42, 36, 65, 48, 50, 27, 33, 59, 47, 51,
       69, 70, 63, 43, 68, 32, 26, 57, 38, 55, 34, 66, 39, 44, 28, 56, 41],
      dtype=int64)}
{'Annual Income (k$)': array([ 15,  16,  17,  18,  19,  20,  21,  23,  24,  25,
28,  29,  30,
        33,  34,  37,  38,  39,  40,  42,  43,  44,  46,  47,  48,  49,
        50,  54,  57,  58,  59,  60,  61,  62,  63,  64,  65,  67,  69,
        70,  71,  72,  73,  74,  75,  76,  77,  78,  79,  81,  85,  86,
        87,  88,  93,  97,  98,  99, 101, 103, 113, 120, 126, 137],
      dtype=int64)}
{'Spending Score (1-100)': array([39, 81,  6, 77, 40, 76, 94,  3, 72, 14, 99,
15, 13, 79, 35, 66, 29,
       98, 73,  5, 82, 32, 61, 31, 87,  4, 92, 17, 26, 75, 36, 28, 65, 55,
       47, 42, 52, 60, 54, 45, 41, 50, 46, 51, 56, 59, 48, 49, 53, 44, 57,
       58, 43, 91, 95, 11,  9, 34, 71, 88,  7, 10, 93, 12, 97, 74, 22, 90,
       20, 16, 89,  1, 78, 83, 27, 63, 86, 69, 24, 68, 85, 23,  8, 18],
      dtype=int64)}
```

[109]:
```
### Finding numerical variables
colname_num = [var for var in df1.columns if df1[var].dtype!='O']
print('There are {} numerical variables\n'.format(len(colname_num)))
print('The numerical variables are :', colname_num)
```

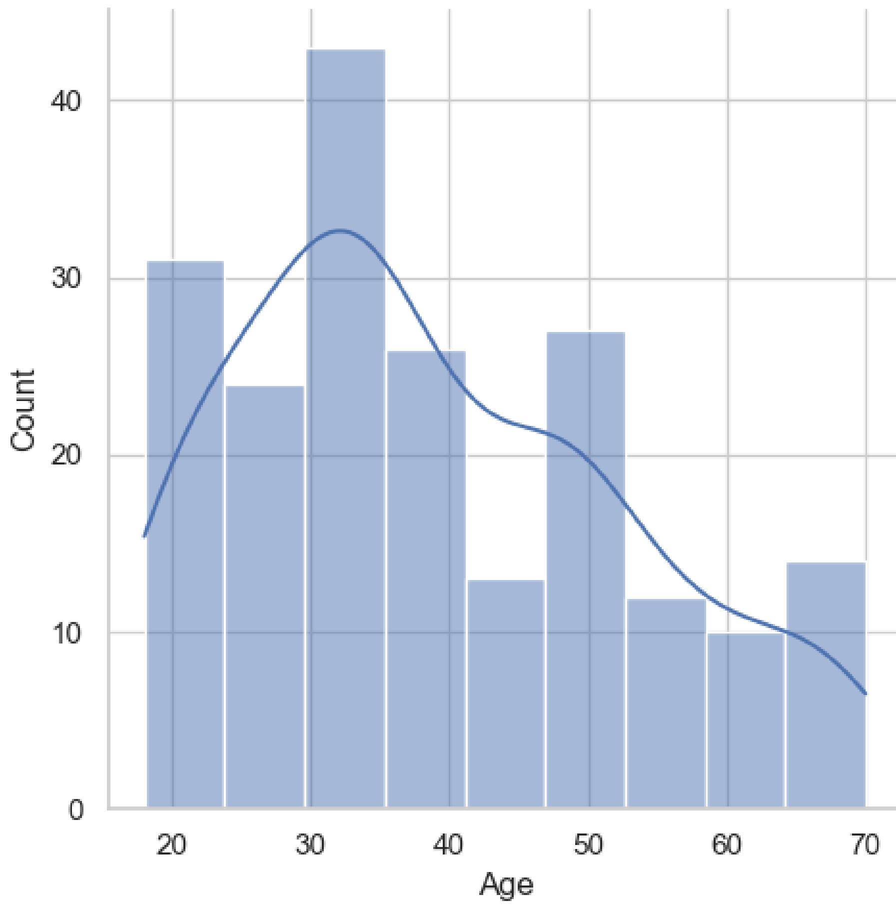
```
There are 4 numerical variables

The numerical variables are : ['CustomerID', 'Age', 'Annual Income (k$)',
'Spending Score (1-100)']
```

[111]:
```
### Finding categorical variables
colname_cat = [var for var in df1.columns if df1[var].dtype=='O']
print('There are {} categorical variables\n'.format(len(colname_cat)))
print('The categorical variables are :', colname_cat)
```

```
There are 1 categorical variables

The categorical variables are : ['Gender']
```

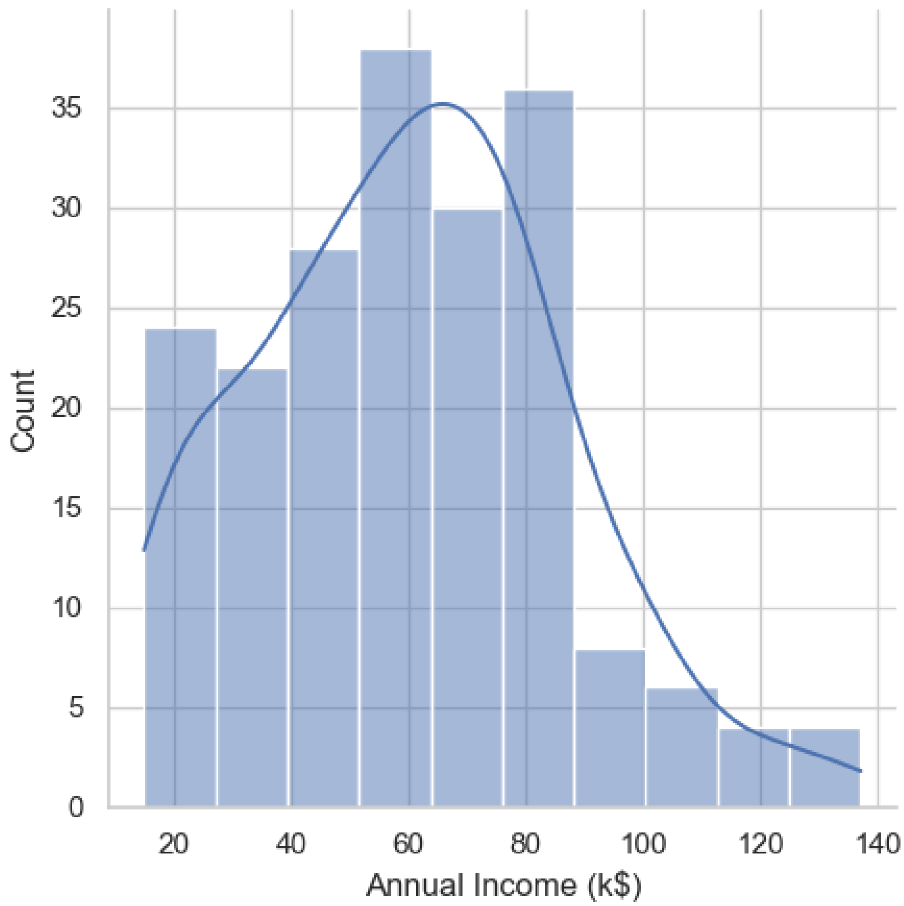
[113]:
```python
### Distribution of age
sns.displot(x='Age', data=df1, kde=True)
```

[113]: <seaborn.axisgrid.FacetGrid at 0x1c9306811c0>



[138]:
```python
### Distribution of income
sns.displot(x='Annual Income (k$)', data=df1, kde=True)
```

[138]: <seaborn.axisgrid.FacetGrid at 0x1c93833e360>



[ ]:
```
### Distribution of score
sns.displot(x='Score', data=df1, kde=True)
```

[ ]:
```
# distribution of categorical variable
print(df1['Gender'].value_counts())
sns.countplot(x='Gender', data=df1)
```

[ ]:
```
df2 = df1.copy()
df2.shape
```

[ ]:
```
### Feature sleection for the model
#Considering only 2 features (Annual income and Spending Score) and no Label␣
↪available
X = df2.iloc[:, [3,4]].values
X
```

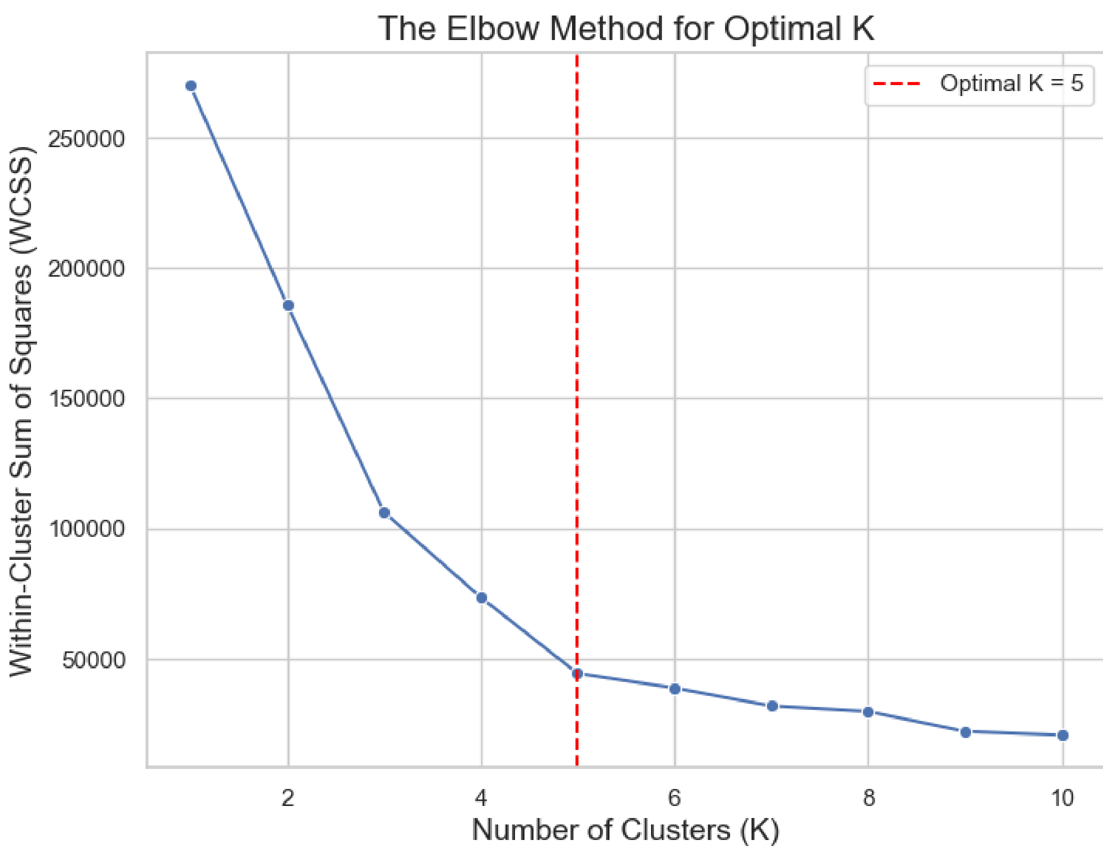
[120]:
```
# Importing the KMeans clustering algorithm from scikit-learn
from sklearn.cluster import KMeans
```

```
# List to store the Within-Cluster Sum of Squares (WCSS) values for each number␣
↪of clusters
wcss = []

# Looping over a range of possible number of clusters (from 1 to 10)
for i in range(1, 11):
    # Creating a KMeans object with 'i' clusters, using the k-means++␣
↪initialization method, and setting a random state for reproducibility
    kmeans = KMeans(n_clusters=i, init='k-means++', random_state=0)

    # Fitting the KMeans model to the dataset 'X' (assumed to be predefined)
    kmeans.fit(X)

    # Appending the inertia (WCSS value) of the current model to the list
    # Inertia represents the sum of squared distances of samples to their␣
↪closest cluster center, measuring the compactness of the clusters
    wcss.append(kmeans.inertia_)

# The 'wcss' list now contains the inertia values for 1 to 10 clusters, useful␣
↪for determining the optimal number of clusters using the elbow method
wcss
```

[120]:
```
[269981.28,
 185917.14253928524,
 106348.37306211119,
 73679.78903948836,
 44448.45544793371,
 38858.9599751439,
 31969.426550235476,
 29858.483597603947,
 22209.851608025547,
 20786.936692059156]
```

[121]:
```
# Visualizing the Elbow method using Seaborn
sns.set(style="whitegrid")  # Setting a style for the seaborn plot
plt.figure(figsize=(8,6))  # Setting the figure size

# Plot the WCSS values
sns.lineplot(x=range(1, 11), y=wcss, marker='o', color='b')

# Adding titles and labels
plt.title('The Elbow Method for Optimal K', fontsize=16)
plt.xlabel('Number of Clusters (K)', fontsize=14)
plt.ylabel('Within-Cluster Sum of Squares (WCSS)', fontsize=14)

# Adding a vertical line to indicate the optimal value of K (5)
```

```
plt.axvline(x=5, color='red', linestyle='--', label='Optimal K = 5')

# Show the plot with the optimal K value marked
plt.legend()
plt.show()

# Print the optimal K value
optimal_k = 5
print(f"The optimal value of K is: {optimal_k}")
```




```
The optimal value of K is: 5
```

[123]:
```
# Import necessary libraries
from sklearn.cluster import KMeans

# Step 1: Model Initialization
# We initialize the KMeans model with 5 clusters as we determined the optimal K␣
↪to be 5
# 'init' parameter is set to 'k-means++' for smart initialization of centroids␣
↪to speed up convergence
```

```
# 'random_state' ensures reproducibility of the results across different runs
kmeans_model = KMeans(n_clusters=5, init='k-means++', random_state=0)

# Step 2: Fitting the model and predicting the clusters
# The fit_predict() method is a combination of two actions:
#   a) It fits the KMeans model to the input data (X)
#   b) It assigns each data point in X to one of the 5 clusters by predicting␣
↪the cluster labels
# 'X' is the dataset we are working with, which contains the features for␣
↪clustering
y_kmeans = kmeans_model.fit_predict(X)
y_kmeans
```

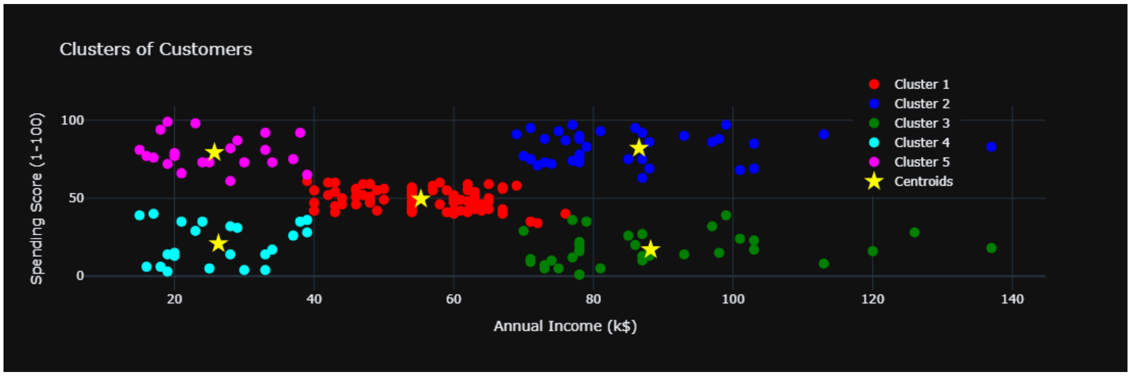
[123]:

```
array([3, 4, 3, 4, 3, 4, 3, 4, 3, 4, 3, 4, 3, 4, 3, 4, 3, 4, 3, 4, 3, 4,
       3, 4, 3, 4, 3, 4, 3, 4, 3, 4, 3, 4, 3, 4, 3, 4, 3, 4, 3, 0,
       3, 4, 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, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 2, 1, 0, 1, 2, 1, 2, 1,
       0, 1, 2, 1, 2, 1, 2, 1, 2, 1, 0, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1,
       2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1,
       2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1,
       2, 1])
```

[132]:

```
import plotly.graph_objects as go

# Creating a scatter plot for each cluster using Plotly
fig = go.Figure()

# Add Cluster 1
fig.add_trace(go.Scatter(
    x=X[y_kmeans == 0, 0], y=X[y_kmeans == 0, 1],
    mode='markers',
    marker=dict(size=10, color='red'),
    name='Cluster 1'
))

# Add Cluster 2
fig.add_trace(go.Scatter(
    x=X[y_kmeans == 1, 0], y=X[y_kmeans == 1, 1],
    mode='markers',
    marker=dict(size=10, color='blue'),
    name='Cluster 2'
))

# Add Cluster 3
fig.add_trace(go.Scatter(
```

```python
    x=X[y_kmeans == 2, 0], y=X[y_kmeans == 2, 1],
    mode='markers',
    marker=dict(size=10, color='green'),
    name='Cluster 3'
))

# Add Cluster 4
fig.add_trace(go.Scatter(
    x=X[y_kmeans == 3, 0], y=X[y_kmeans == 3, 1],
    mode='markers',
    marker=dict(size=10, color='cyan'),
    name='Cluster 4'
))

# Add Cluster 5
fig.add_trace(go.Scatter(
    x=X[y_kmeans == 4, 0], y=X[y_kmeans == 4, 1],
    mode='markers',
    marker=dict(size=10, color='magenta'),
    name='Cluster 5'
))

# Add Centroids
fig.add_trace(go.Scatter(
    x=kmeans_model.cluster_centers_[:, 0], y=kmeans_model.cluster_centers_[:, 1],
    mode='markers',
    marker=dict(size=15, color='yellow', symbol='star'),
    name='Centroids'
))

# Customize layout
fig.update_layout(
    title="Clusters of Customers",
    xaxis_title="Annual Income (k$)",
    yaxis_title="Spending Score (1-100)",
    legend=dict(x=0.8, y=1.2),
    template='plotly_dark'
)

# Show the plot
fig.show()
```



#####

Made with by Zahid Salim Shaikh

[ ]:

```

```

This notebook was converted with convert.ploomber.io