Segmenting for Success: A Deep Dive into K-Means Clustering

Minor in AI, IIT Ropar

9th May, 2025

Chapter 2: Beyond the Spreadsheet: K-Means in Action

Welcome back, aspiring data wranglers! In the previous chapter, we laid the groundwork for understanding clustering. We explored how to group similar data points together, focusing on intuition and manual calculations. Now, it's time to level up! We'll trade our spreadsheets for code and unleash the power of the K-Means algorithm.

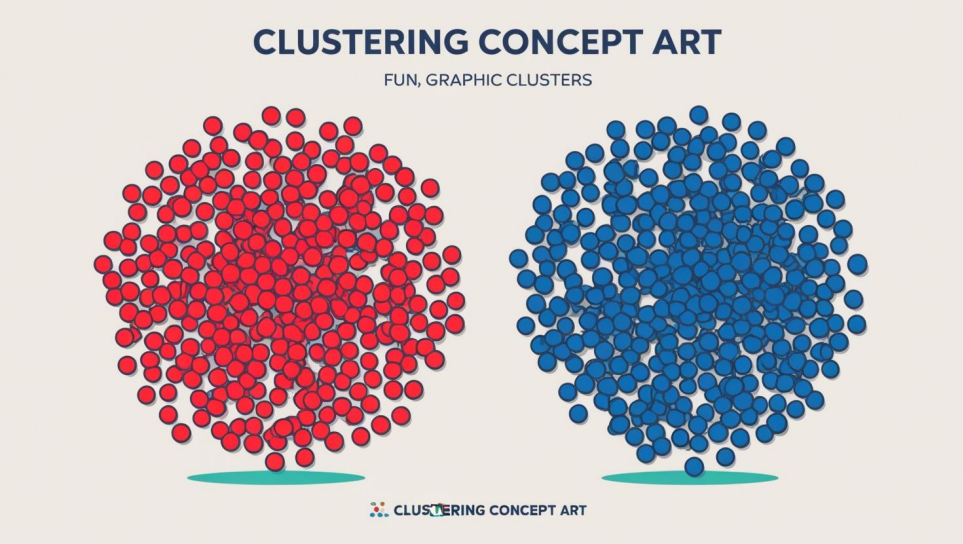

Figure 1: [Placeholder for an illustrative image about data clustering or "leveling up" in data science.]

2.1. The Shopkeeper's Inventory: A Recap

Remember our friendly neighborhood shopkeeper from Chapter 1? He's still trying to make sense of his customer data. He wants to understand his clientele better so he can stock the right items and tailor his marketing efforts.

Let's quickly revisit the data:

Table 1: Shopkeeper's Customer Data Recap

Customer	Frequency (Visits)	Spend (Amount)
C1	2	500
C2	10	800
C3	4	300
C4	11	1200
C5	3	350
C6	9	1000

We manually grouped these customers based on their frequency of visits and spending habits. But, as you can imagine, manually processing large datasets is a tedious and error-prone task. This is where the K-Means algorithm comes to the rescue!

2.2. Translating Intuition into Code

Before diving into libraries, let's peek under the hood and see what a raw K-Means implementation would look like. This helps us appreciate the power and convenience that libraries offer.

```python
import math

# Data
customers = {
    "C1": (2, 500),
    "C2": (10, 800),
    "C3": (4, 300),
    "C4": (11, 1200),
    "C5": (3, 350),
    "C6": (9, 1000)
}

# Initialize centroids manually
centroid_A = customers["C1"]
centroid_B = customers["C4"]

def euclidean(p1, p2):
    return math.sqrt((p1[0] - p2[0]) ** 2 + (p1[1] - p2[1]) ** 2)

def compute_new_centroid(cluster):
    freq = sum(c[0] for c in cluster) / len(cluster)
    spend = sum(c[1] for c in cluster) / len(cluster)
    return (freq, spend)

# Run until convergence
iteration = 1
prev_assignment = {}
while True:
    print(f"\n--- Iteration {iteration} ---")

    cluster_A = []
    cluster_B = []
    assignment = {}

    for cid, (f, s) in customers.items():
        dA = euclidean((f, s), centroid_A)
        dB = euclidean((f, s), centroid_B)

        if dA < dB:
            cluster_A.append((f, s))
            assignment[cid] = 'A'
        else:
            cluster_B.append((f, s))
            assignment[cid] = 'B'

        print(f"{cid}: Dist to A={dA:.2f}, B={dB:.2f} => Cluster {assignment[cid]}")

    if assignment == prev_assignment:
        print("\nConverged.")
        break

    prev_assignment = assignment
    centroid_A = compute_new_centroid(cluster_A)
    centroid_B = compute_new_centroid(cluster_B)

    print(f"\nNew Centroid A: {centroid_A}")
    print(f"New Centroid B: {centroid_B}")

    iteration += 1

# Final Output
```

```
print("\nFinal Clusters:")
print("Cluster A:", [cid for cid, grp in assignment.items() if grp == 'A'])
print("Cluster B:", [cid for cid, grp in assignment.items() if grp == 'B'])
```

Listing 1: Raw K-Means Implementation from Scratch

This code does the same thing we did manually in the spreadsheet:

1. **Data:** Defines the customer data as a dictionary.
2. **`euclidean` function:** Calculates the Euclidean distance between two points.
3. **`compute_new_centroid` function:** Calculates the average frequency and spending for a given cluster.
4. **The `while` loop:** This is the heart of the K-Means algorithm. It iterates until the clusters stop changing (convergence).
   - It assigns each customer to the nearest centroid.
   - It recalculates the centroids based on the new cluster assignments.

This code, while functional, is a bit lengthy. Imagine writing this for hundreds or thousands of data points! Thankfully, libraries like `scikit-learn` offer a much cleaner and efficient way to perform K-Means clustering.

2.3. K-Means with scikit-learn: A Breath of Fresh Air

Let's see how we can achieve the same results using the `scikit-learn` library:

```
import numpy as np
from sklearn.cluster import KMeans

# Data
customer_ids = ["C1", "C2", "C3", "C4", "C5", "C6"]
data = np.array([
    [2, 500],
    [10, 800],
    [4, 300],
    [11, 1200],
    [3, 350],
    [9, 1000]
])

# Initial centroids from C1 and C4
initial_centroids = np.array([
    [2, 500],    # C1
    [11, 1200]   # C4
])

# KMeans model with manually specified initial centroids
kmeans = KMeans(n_clusters = 2, init = initial_centroids, n_init = 1, max_iter = 25)
kmeans.fit(data)

# Results
labels = kmeans.labels_

# Display results
for cid, label in zip(customer_ids, labels):
    print(f"{cid} - Cluster {label}")

print("\nCluster Centers:")
print(kmeans.cluster_centers_)
```

Listing 2: K-Means with scikit-learn (Manual Centroids)

Let's break down the key parts:

- **`import KMeans from sklearn.cluster`:** Imports the `KMeans` class.
- **`data = np.array(...)`**:Converts our data into a NumPy array, which is the preferred format for `scikit-learn`.

- **`initial_centroids = np.array(...)`**:Sets the initial centroid values.
- **`kmeans = KMeans(n_clusters=2, init = initial_centroids, n_init = 1, max_iter = 25)`**:Creates a `KMeans` object.
  - `n_clusters`: Specifies the number of clusters we want (in this case, 2). This is the "K" in K-Means.
  - `init`: Defines the method for initialization of the centroids. Here we are initializing the centroids manually by passing the `initial_centroids` array.
  - `n_init`: Sets the number of times the K-means algorithm will be run with different centroid seeds. If init is 'k-means++' or 'random', the argument is used to declare the number of runs.
  - `max_iter`: Sets the maximum number of iterations for the algorithm to converge.
- **`kmeans.fit(data)`:** Trains the K-Means model on our data. This is where the magic happens! The algorithm finds the best cluster assignments based on the data and the specified parameters.
- **`labels = kmeans.labels_`:** Retrieves the cluster labels for each data point. `labels_` is an attribute of the trained `kmeans` object. It's a NumPy array where each element corresponds to a data point and indicates which cluster that data point belongs to.
- **`kmeans.cluster_centers_`:** Retrieves the coordinates of the final cluster centroids.

See how much simpler this is compared to the manual implementation? `scikit-learn` handles all the complex calculations and iterations for us.

2.4. Random Centroid Initialization: Letting the Algorithm Choose

In the previous example, we manually chose the initial centroids (C1 and C4). But what if we want the algorithm to pick them randomly?

```python
import numpy as np
from sklearn.cluster import KMeans

# Data
customer_ids = ["C1", "C2", "C3", "C4", "C5", "C6"]
data = np.array([
    [2, 500],
    [10, 800],
    [4, 300],
    [11, 1200],
    [3, 350],
    [9, 1000]
])

# KMeans model without manually specified initial centroids
kmeans = KMeans(n_clusters = 2, n_init = 10, max_iter = 25) # n_init='auto' is default in newer versions
kmeans.fit(data)

# Results
labels = kmeans.labels_

# Display results
for cid, label in zip(customer_ids, labels):
    print(f"{cid} - Cluster {label}")

print("\nCluster Centers:")
print(kmeans.cluster_centers_)
```

Listing 3: K-Means with scikit-learn (Random Centroids)

The key difference here is that we **removed the `init` parameter**. By default, `scikit-learn` uses the "k-means++" initialization method, which intelligently chooses initial centroids to speed up convergence. Also, we can tweak the `n_init` parameter, which decides how many

times the K-means algorithm is run with different centroid seeds. (Note: newer versions of scikit-learn might default `n_init` to `'auto'` when `init='k-means++'` is used).

2.5. Visualizing the Clusters

To gain a better understanding of the results, let's visualize the clusters using a scatter plot:

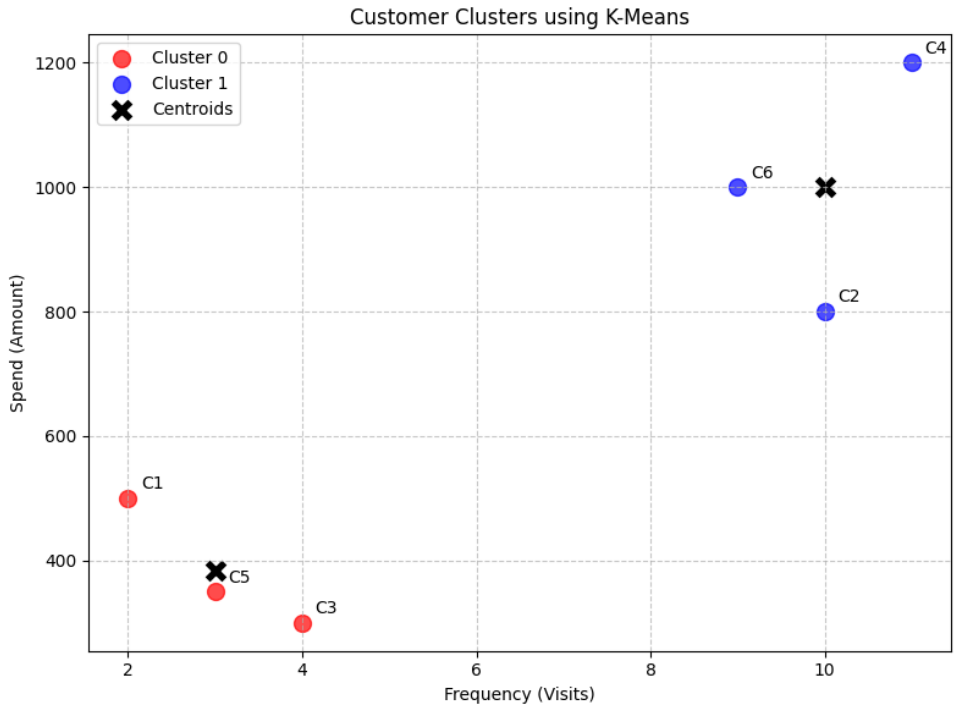

Figure 2: [Placeholder for a Scatter Plot Image showing the Customer Clusters. This plot would visually separate customers based on their frequency and spend, with centroids marked.]

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

# Data
customer_ids = ["C1", "C2", "C3", "C4", "C5", "C6"]
data = np.array([
    [2, 500], [10, 800], [4, 300], [11, 1200], [3, 350], [9, 1000]
])

# Initial centroids from C1 and C4 (for reproducibility with example)
initial_centroids = np.array([ [2, 500], [11, 1200] ])

# Fit KMeans
kmeans = KMeans(n_clusters=2, init=initial_centroids, n_init=1, max_iter=100, random_state=42)
kmeans.fit(data)
labels = kmeans.labels_
centroids = kmeans.cluster_centers_

# Plot
colors = ['red', 'blue']
plt.figure(figsize=(8,6)) # Added figure size for better layout
for i in range(2):
    cluster_points = data[labels == i]
    plt.scatter(cluster_points[:, 0], cluster_points[:, 1], c=colors[i], label=f'Cluster {i}',
     s=100, alpha=0.7)

# Plot centroids
plt.scatter(centroids[:, 0], centroids[:, 1], c='black', marker='X', s=200, label='Centroids',
     edgecolors='w')

```

```
# Add customer IDs as labels
for i, txt in enumerate(customer_ids):
    plt.annotate(txt, (data[i, 0] + 0.15, data[i, 1] + 15)) # Adjusted offset

plt.xlabel('Frequency (Visits)')
plt.ylabel('Spend (Amount)')
plt.title('Customer Clusters using K-Means')
plt.legend()
plt.grid(True, linestyle='--', alpha=0.7)
plt.tight_layout() # Adjust plot to ensure everything fits
plt.show()
```

Listing 4: Plotting Customer Clusters with Matplotlib

This code generates a scatter plot where:

- Each customer is represented by a point.
- The color of the point indicates the cluster it belongs to.
- The "X" marks represent the centroids of each cluster.

Visualizing the clusters helps us assess the quality of the clustering and identify potential patterns.

2.6. The Elbow Method: Finding the Optimal "K"

Choosing the right number of clusters ("K") is crucial for K-Means. How do we know if we should use 2 clusters, 3 clusters, or more? The elbow method provides a visual approach to determine the optimal "K" value.

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

# Example data (expanded for a more illustrative elbow curve)
data_dict = { # Renamed from 'data' to avoid conflict
    'Frequency': [2, 10, 4, 11, 3, 9, 3, 7, 9, 4, 12, 5, 8, 5, 1, 13, 6, 8, 2, 10],
    'Spend': [500, 800, 300, 1200, 350, 1000, 750, 800, 1000, 1200, 450, 300, 200, 600, 400,
    1300, 320, 900, 550, 1100]
}
df = pd.DataFrame(data_dict)

# Scale the features
scaler = StandardScaler()
scaled_data = scaler.fit_transform(df)

# Try KMeans for different values of k
inertia = []
K_range = range(1, 11) # Renamed from K
for k_val in K_range: # Renamed from k
    kmeans_model = KMeans(n_clusters=k_val, random_state=42, n_init=10) # Added n_init
    kmeans_model.fit(scaled_data)
    inertia.append(kmeans_model.inertia_)

# Plot the Elbow
plt.figure(figsize=(8, 4))
plt.plot(K_range, inertia, 'bo-')
plt.xlabel('Number of clusters (k)')
plt.ylabel('Inertia (Sum of squared distances)')
plt.title('Elbow Method For Optimal k')
plt.xticks(K_range) # Ensure all k values are shown as ticks
plt.grid(True, linestyle='--', alpha=0.7)
plt.show()
```

Listing 5: Elbow Method for Optimal K

This code does the following:

1. **Calculates Inertia:** For each value of "K" (from 1 to 10), it runs K-Means and calculates the inertia. Inertia is the sum of squared distances of samples to their closest cluster center.

It's a measure of how internally coherent the clusters are. Lower inertia is generally better.

2. **Plots the Elbow Curve:** It plots the inertia values against the corresponding "K" values.

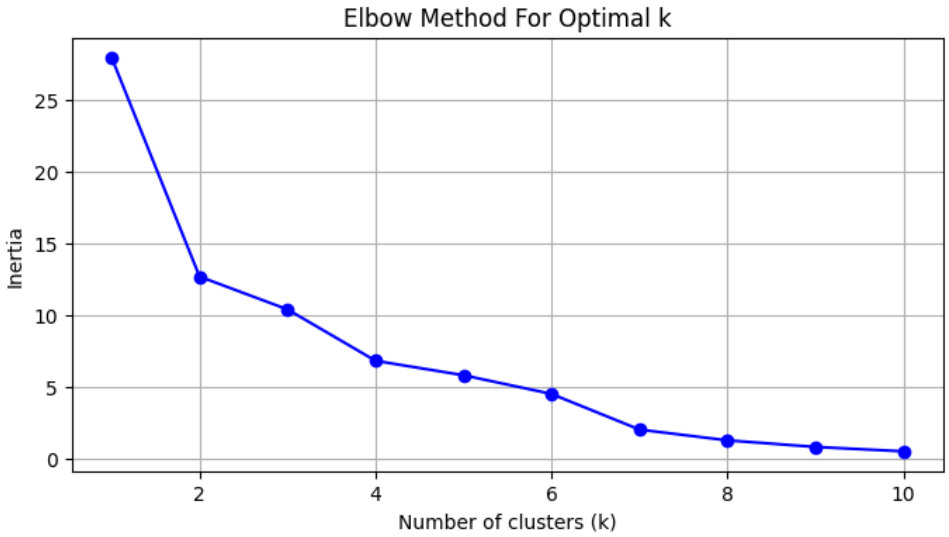

Figure 3: [Placeholder for the Elbow Method Curve Image. This graph would show inertia decreasing as K increases, with an "elbow" indicating the optimal K.]

The elbow method suggests choosing the "K" value at the "elbow" of the curve – the point where the rate of decrease in inertia starts to slow down significantly. This point is considered a good balance between minimizing inertia and avoiding overfitting (having too many clusters).

2.7. Scaling Up: Clustering a Real-World Dataset

Now, let's tackle a more challenging scenario with a real-world dataset: online retail data. This dataset contains transaction information for an online store, with over 500,000 rows!

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
# import seaborn as sns # Not used in this snippet, can be removed if not needed later

from sklearn.preprocessing import StandardScaler
from sklearn.cluster import KMeans
# from sklearn.metrics import silhouette_score # Not used in this snippet

# Step 1: Load the dataset
# Assuming 'Online Retail.xlsx' is in the same directory or path is specified
# df = pd.read_excel('Online Retail.xlsx') # This line would be active

# Create a dummy dataframe for demonstration as the file is not available
data_sample = {
    'InvoiceNo': ['536365', '536365', 'C536379', '536381', '536381', '536382', '536383', '536384', '536385', '536385'],
    'StockCode': ['85123A', '71053', '22485', '22728', '22727', '21755', '84879', '22423', '85099B', '22633'],
    'Description': ['CREAM CUPID HEARTS COAT HANGER', 'WHITE METAL LANTERN', 'MANUAL', 'ALARM CLOCK BAKELIKE PINK', 'ALARM CLOCK BAKELIKE RED', 'STRAWBERRY CERAMIC TRINKET BOX', 'ASSORTED COLOUR BIRD ORNAMENT', 'REGENCY CAKESTAND 3 TIER', 'JUMBO BAG RED RETROSPOT', 'HAND WARMER UNION JACK'],
    'Quantity': [6, 6, -1, 2, 2, 12, 32, 1, 10, 6],
    'InvoiceDate': ['12/1/2010 8:26', '12/1/2010 8:26', '12/1/2010 9:41', '12/1/2010 9:41', '12/1/2010 9:41', '12/1/2010 9:45', '12/1/2010 9:49', '12/1/2010 9:53', '12/1/2010 9:56', '12/1/2010 9:56'],
    'UnitPrice': [2.55, 3.39, 2.1, 3.75, 3.75, 1.65, 1.69, 12.75, 1.95, 1.85],
    'CustomerID': [17850.0, 17850.0, 14527.0, 15311.0, 15311.0, 16133.0, 13047.0, 18074.0, 17850.0, 17850.0]
```

```
}
df = pd.DataFrame(data_sample)
df['InvoiceDate'] = pd.to_datetime(df['InvoiceDate'])


# Step 2: Data Preprocessing
# Remove the rows with missing customer ids
df = df[pd.notnull(df['CustomerID'])]

# Remove canceled orders (InvoiceNo starting with 'C')
df = df[~df['InvoiceNo'].astype(str).str.startswith('C')]

# Remove negative quantities (and zero, if applicable)
df = df[df['Quantity'] > 0]

# Step 3: Feature Engineering
# Calculate TotalPrice
df['TotalPrice'] = df['Quantity'] * df['UnitPrice']

# Aggregate data by CustomerID
customer_df = df.groupby('CustomerID').agg(
    Frequency=('InvoiceNo', 'nunique'),  # Frequency
    TotalQuantity=('Quantity', 'sum'),
    Monetary=('TotalPrice', 'sum')
).reset_index()

# Step 4: Feature Scaling
features = ['Frequency', 'TotalQuantity', 'Monetary']
scaler = StandardScaler()
scaled_features = scaler.fit_transform(customer_df[features])

# Step 5: Determine the optimal number of clusters using the Elbow Method
sse = [] # Sum of Squared Errors (Inertia)
k_range = range(1, 11)
for k in k_range:
    kmeans = KMeans(n_clusters=k, random_state=42, n_init=10) # Added n_init
    kmeans.fit(scaled_features)
    sse.append(kmeans.inertia_)

print(sse)
print("\n") # For spacing in output if run

# Plot the Elbow Curve
plt.figure(figsize=(8, 5))
plt.plot(k_range, sse, marker='o')
plt.title('Elbow Method for Online Retail Data')
plt.xlabel('Number of clusters (k)')
plt.ylabel('Sum of squared distances (Inertia)')
plt.xticks(k_range)
plt.grid(True, linestyle='--', alpha=0.7)
plt.show()
```

Listing 6: Processing Online Retail Data and Elbow Method

This code performs the following steps:

1. **Data Loading:** Loads the online retail dataset (here, a sample is used for demonstration).
2. **Data Preprocessing:**
   - Removes rows with missing customer IDs.
   - Removes cancelled orders (identified by "C" in the `InvoiceNo`).
   - Removes rows with negative quantities (indicating returns).
3. **Feature Engineering:**
   - Calculates the total price for each transaction (`Quantity` * `UnitPrice`).
   - Aggregates the data by customer ID to create features like frequency (number of unique invoices), total quantity purchased, and total monetary value.
4. **Feature Scaling:** Scales the features using `StandardScaler`. This is important because K-Means is sensitive to the scale of the features.

5. **Elbow Method:** Calculates and plots the inertia for different values of "K" to help determine the optimal number of clusters.

2.8. Silhouette Score: A Complementary Metric

While the elbow method provides a visual guide, we can also use a more quantitative measure to evaluate the clustering performance: the silhouette score.

```
# Continuing from the previous section (scaled_features is available)
from sklearn.metrics import silhouette_score
# Ensure scaled_features is defined from the previous step.
# For standalone running, you'd need to regenerate customer_df and scaled_features.
# Assuming scaled_features from section 2.7 is in memory.
# If not, re-run the data prep steps from 2.7 or load a pre-scaled dataset.
# For this example, let's re-use the `scaled_data` from section 2.6 for a quick run,
# as `scaled_features` from 2.7 depends on the larger (dummy) retail dataset.
# In a real notebook, `scaled_features` would persist.
# Using `scaled_data` (from the smaller example in 2.6) for silhouette plot:
# Example: scaled_features_for_silhouette = scaled_data
# If using the Online Retail data, `scaled_features` should be used.
# Here, we'll assume `scaled_features` is ready from the previous block.
# If it's not (e.g., running this section independently with a new kernel),
# the following will error or use a different `scaled_features`.
# For demonstration, let's assume scaled_features (from retail data) is ready.
# If `scaled_features` is small (e.g. from dummy data), the plot might not be very informative.

# Step 6: Evaluate clustering with Silhouette Score
silhouette_scores = []
k_range_silhouette = range(2, 11) # Silhouette score is not defined for k=1
for k in k_range_silhouette:
    kmeans = KMeans(n_clusters=k, random_state=42, n_init=10) # Added n_init
    labels = kmeans.fit_predict(scaled_features) # Use scaled_features from retail data
    score =  silhouette_score(scaled_features, labels)
    silhouette_scores.append(score)
    print(f'For n_clusters = {k}, the Silhouette Score is {score:.4f}')

# Plot Silhouette Scores
plt.figure(figsize=(8, 5))
plt.plot(k_range_silhouette, silhouette_scores, marker='o')
plt.title('Silhouette Scores for Various Clusters (Online Retail Data)')
plt.xlabel('Number of clusters (k)')
plt.ylabel('Silhouette Score')
plt.xticks(k_range_silhouette)
plt.grid(True, linestyle='--', alpha=0.7)
plt.show()
```

Listing 7: Silhouette Score Evaluation

- **Silhouette Score:** The silhouette score measures how well each data point fits within its cluster. It ranges from -1 to +1, where:
  - +1: Indicates that the data point is well-clustered.
  - 0: Indicates that the data point is close to the decision boundary between two clusters.
  - -1: Indicates that the data point might be assigned to the wrong cluster.

We calculate the silhouette score for different values of "K" and choose the "K" that maximizes the silhouette score.

2.9 Next Steps

Finally, we would choose the optimal number of clusters based on both the elbow method and the silhouette score. Then, we would apply K-Means with that optimal "K" value, assign cluster labels to the customers, and analyze the characteristics of each cluster.

2.10. Conclusion

This concludes our journey into the world of K-Means clustering with code. We've explored how to implement K-Means using `scikit-learn`, how to choose the optimal number of clusters using the elbow method and the silhouette score, and how to apply K-Means to a real-world dataset.

Remember, K-Means is a powerful tool for segmentation and can be applied to a wide range of problems. So, go out there and start clustering!