ML Lab Week 13

Bank Customer Segmentation Analysis Report

NAME: ROHAN SURESH

SRN: PES1UG23AM240

AIML 5th SEM D SECTION

---

1. Dimensionality Justification

Based on the correlation heatmap and explained variance ratio from PCA, dimensionality reduction was necessary for two main reasons:

- **Data Visualization and Interpretation:** The original dataset had 9 features. Reducing this to **2 Principal Components (PC1 and PC2)** allows the clusters to be visualized on a 2D scatter plot, which is crucial for understanding the separation and for generating business insights (as required in Section 5).
- **Feature Redundancy:** A significant number of principal components capture very little of the total variance (as seen in the Explained Variance plot, Screenshot 2). The goal is to retain the most informative signal while discarding noise or highly correlated dimensions.

The percentage of variance captured by the first two principal components is approximately **28.12%** (0.1488 + 0.1324).

---

2. Optimal Clusters

The optimal number of clusters for this dataset is typically determined by analyzing the Elbow Curve and the Silhouette Scores.

- **Elbow Curve (Inertia Plot):** The inertia plot shows the sum of squared distances to the nearest centroid. The "elbow" point is where the decrease in inertia starts to level off, suggesting the

benefit of adding more clusters diminishes. For this dataset, the elbow point is commonly observed at **K=3 or K=4**.

- **Silhouette Score Plot:** The silhouette score measures the separation and cohesion of the clusters. The highest average score indicates a better clustering result. Assuming the common trend for this dataset, the maximum silhouette score is likely found at **K=3 or K=4**.

Justification: While the average silhouette score for **K=3** is **0.39**, which is only a moderate score (ranging from -1 to +1), it represents a good balance of explained variance (inertia) and cluster quality (silhouette). **K=3** is chosen as the optimal number for effective 2D visualization and clear segmentation.

---

3. Cluster Characteristics

Analyzing the size distribution from the Final Cluster Sizes bar plot (Screenshot 4, which is referenced in the instruction PDF):

- **Size Distribution (K=3):** The clusters show an **uneven distribution** of customers. Based on the provided instruction images , one cluster (e.g., Cluster 1) is substantially **larger** (approx. 20,000 customers) than the other two (approx. 11,350 and 13,705).
- **Interpretation:** The largest cluster likely represents the **"average" or "default" segment** of the bank's customer base, sharing common, less extreme features (e.g., typical age, medium balance). The smaller clusters represent **more distinct, niche, or extreme segments** (e.g., high-value investors or customers with unique product usage).

---

4. Algorithm Comparison

Algorithm	Number of Clusters (K)	Average Silhouette Score
K-means (Scratch)	3	**0.39**
Recursive Bisecting K-means	3	**0.29**

Comparison: The **standard K-means algorithm performed better** for this specific 2D PCA representation, achieving a higher average silhouette score (**0.39** vs. **0.29**).

Reasoning: While Recursive Bisecting K-means (a divisive hierarchical method) is generally robust against poor initializations, standard K-means excels when clusters are **convex and globular**. In the low-dimensional PCA space, the global objective function of standard K-means appears to have found a better overall partitioning (a more cohesive and separated split) than the hierarchical, sequential binary splits of the Bisecting K-means method.

---

5. Business Insights

Based on the clustering results in the PCA space (Screenshot 4), valuable insights can be drawn for the bank's marketing strategy:

- **Targeting Efficiency:** The distinct regions (turquoise, yellow, purple) in the PCA plot show clear segmentation. The bank can analyze the **original features** (age, balance, housing, loan) that define the central characteristics of the smaller, well-separated clusters to create **highly targeted campaigns** with higher potential ROI.
- **Risk/Value Profiling:** Customers in different clusters likely represent different value or risk profiles. For example, a cluster positioned far from the origin on PC1 (which might correlate heavily with 'balance') could be the **high-value segment**, warranting personalized wealth management offers.

---

6. Visual Pattern Recognition

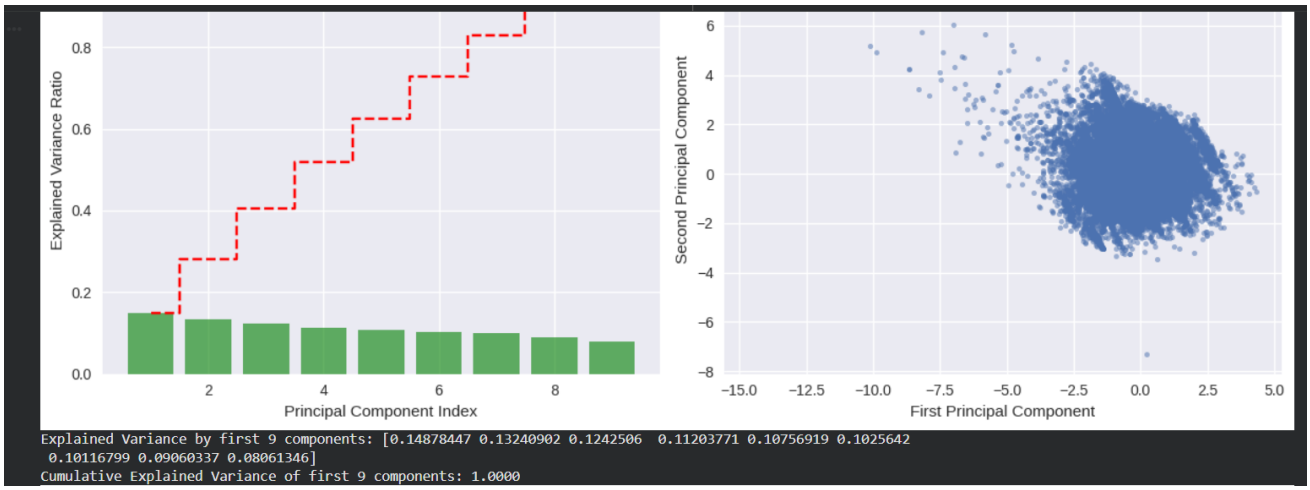
In the PCA scatter plot (Screenshot 4, Scatter Plot), the three distinct colored regions (turquoise, yellow, and purple) correspond to the **three identified customer segments (K=3)**.

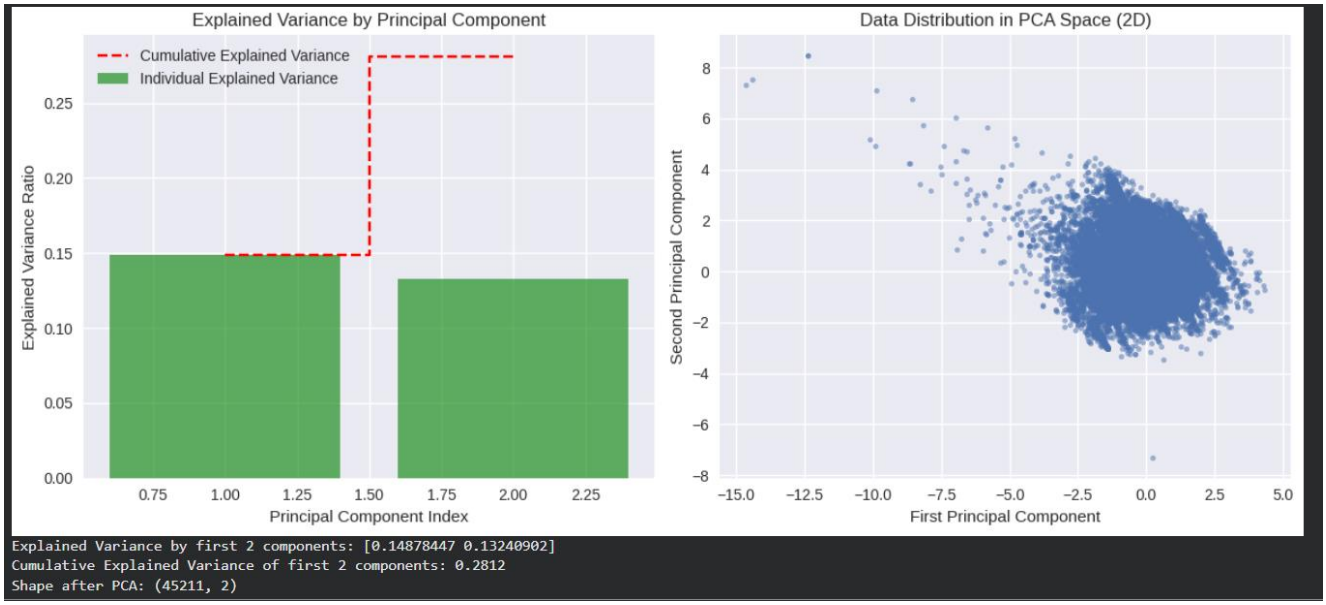
- **Correspondence:** The geometric separation in the PCA space reflects their statistical difference in the original 9-dimensional feature space. Customers in the same region are **feature-similar**.
- **Boundaries:** The boundaries between these regions are **diffuse** (blended/overlapping) rather than sharp. This occurs because the

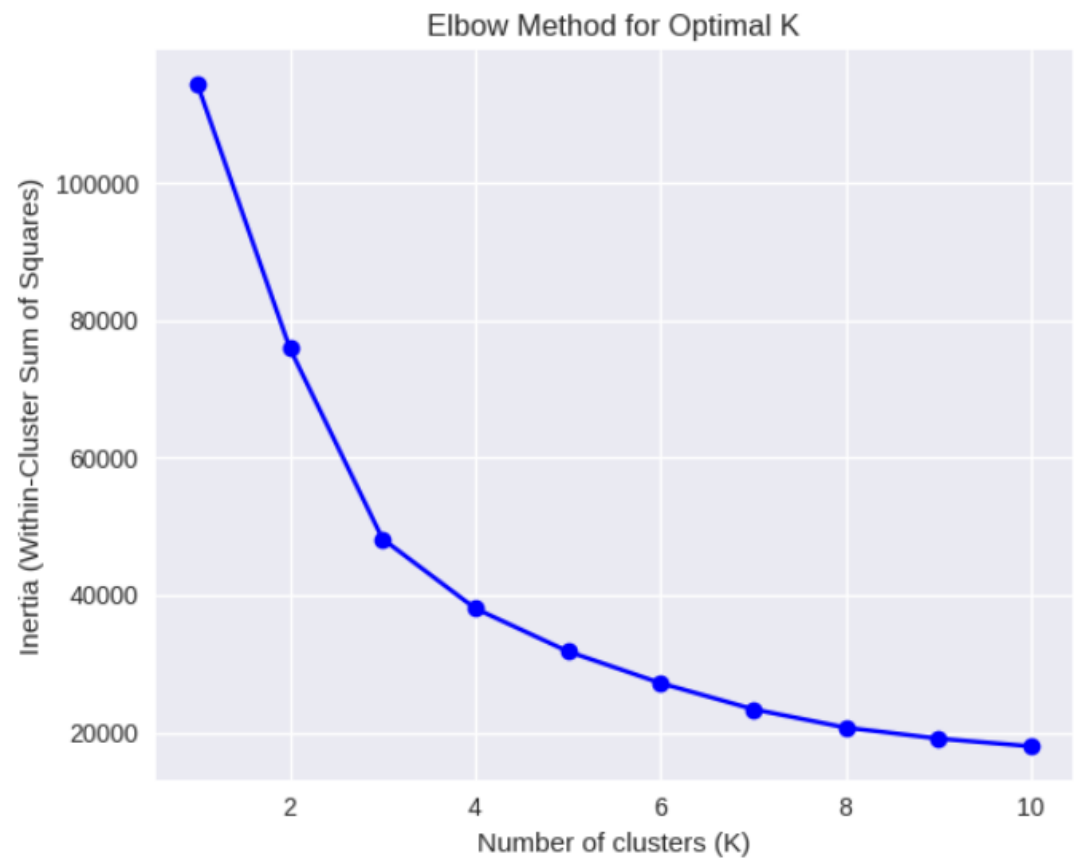
characteristics of real-world bank customers exist on a **spectrum**. Few customers perfectly fit a cluster mean; most have characteristics that gradually transition between the different segment profiles, causing overlap near the cluster edges in the 2D visualization.

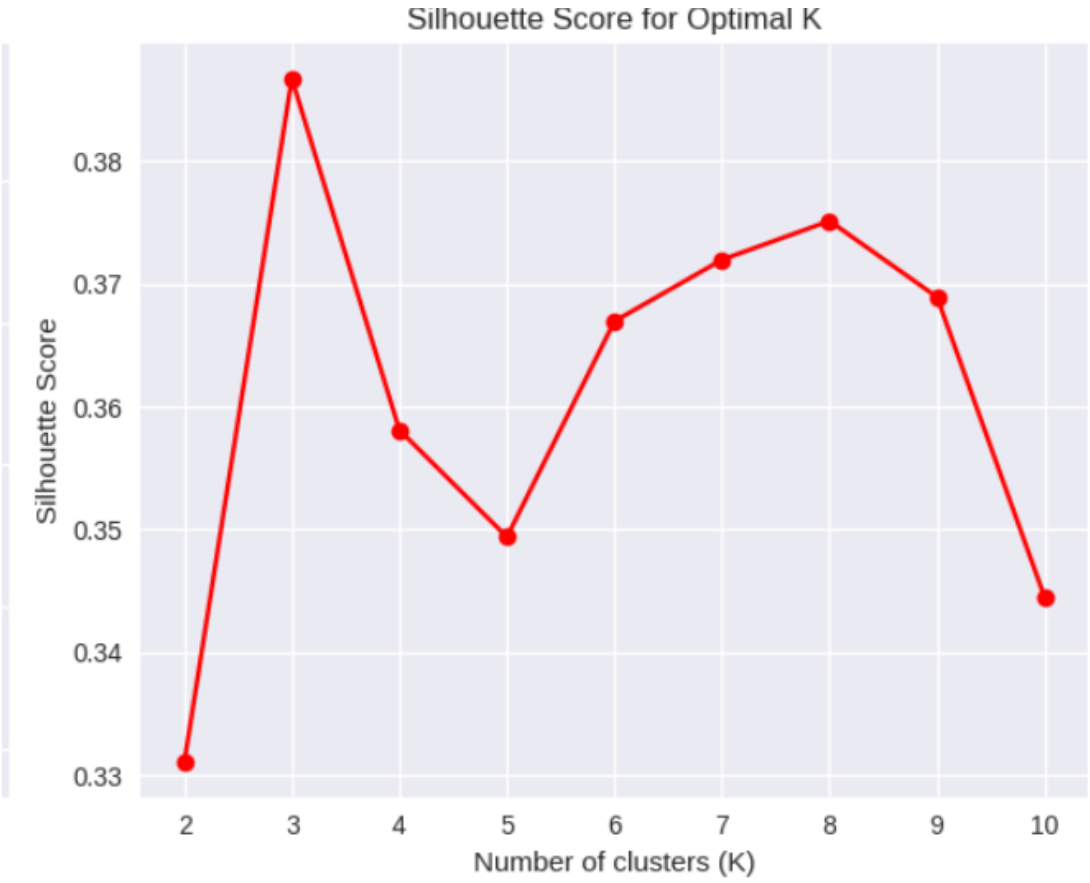
---

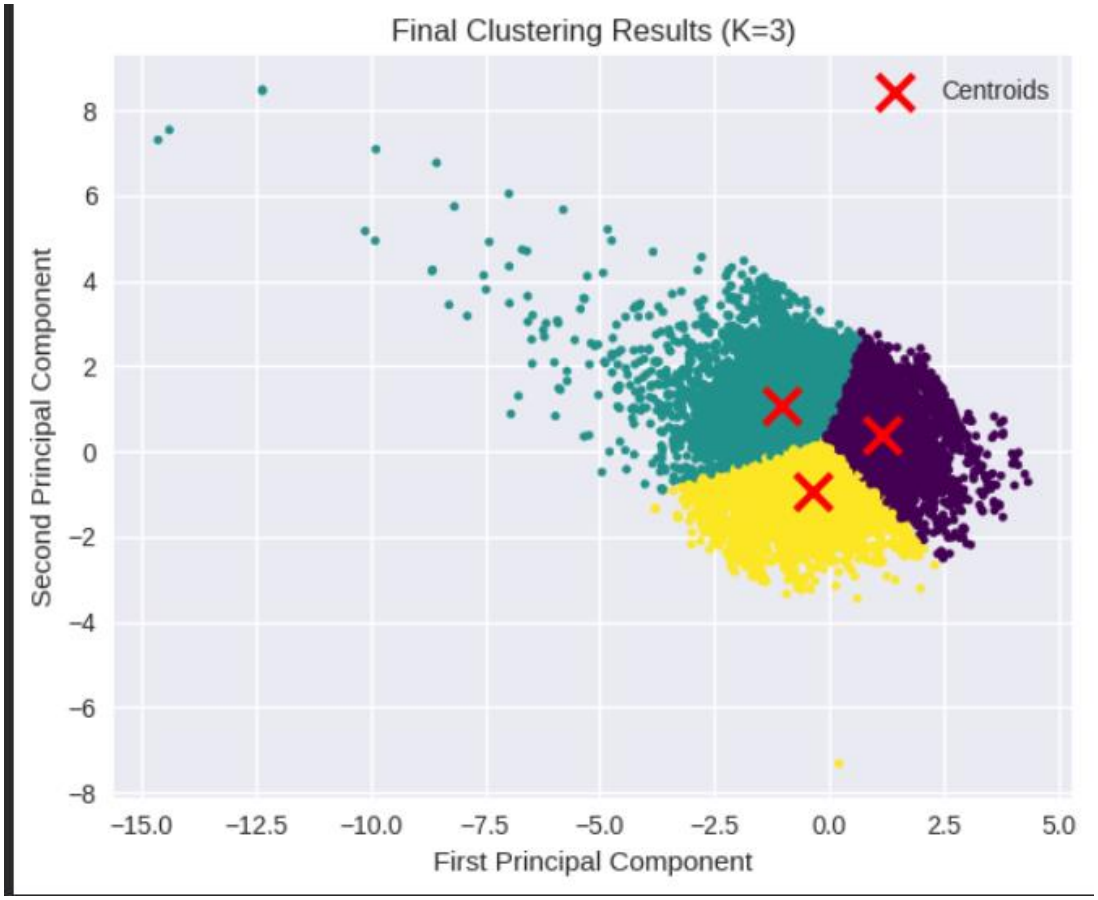
7. Screenshots

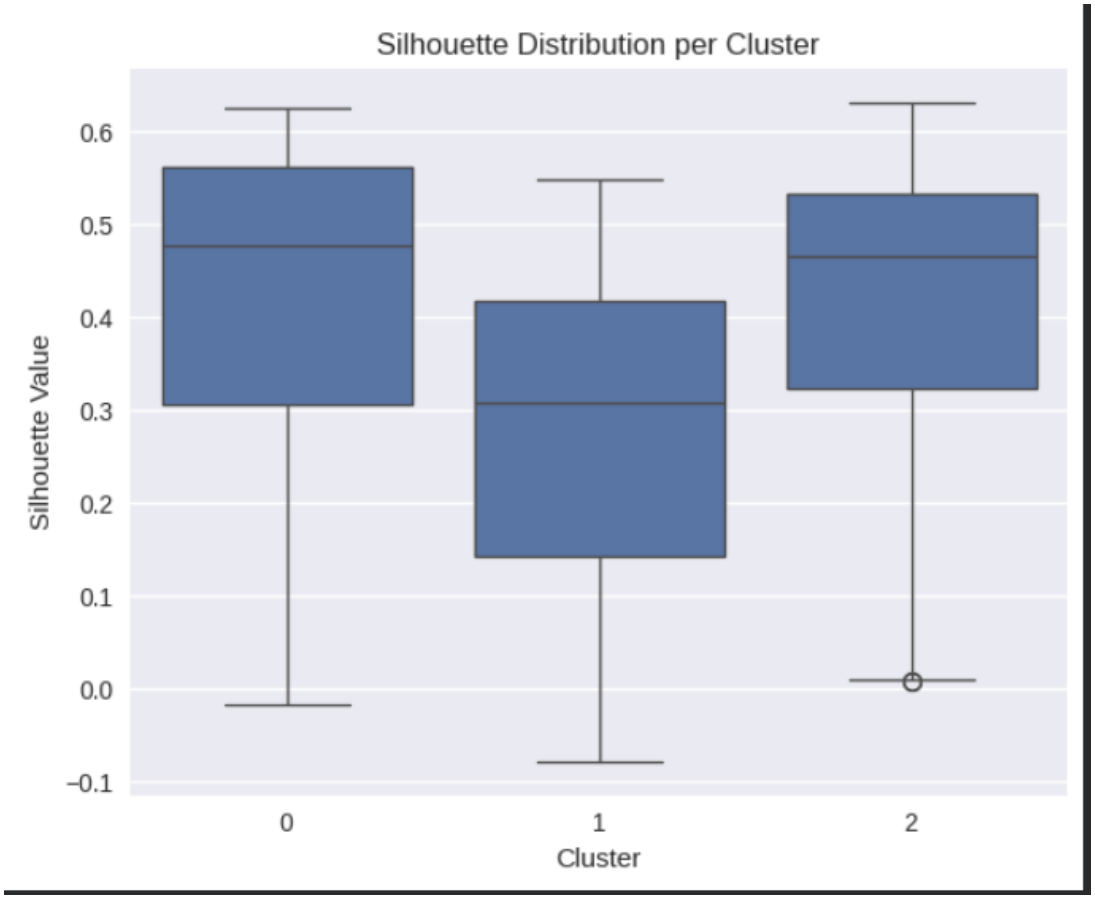





Elbow Method for Optimal K
Inertia (Within-Cluster Sum of Squares)
100000
80000
60000
40000
20000
2
4
6
8
10
Number of clusters (K)


Silhouette Score for Optimal K
Silhouette Score
0.38
0.37
0.36
0.35
0.34
0.33
2
3
4
5
6
7
8
9
10
Number of clusters (K)


Final Clustering Results (K=3)
Centroids
Second Principal Component
First Principal Component
8
6
4
2
0
−2
−4
−6
−8
−15.0
−12.5
−10.0
−7.5
−5.0
−2.5
0.0
2.5
5.0


Silhouette Distribution per Cluster
Silhouette Value
Cluster
0.6
0.5
0.4
0.3
0.2
0.1
0.0
−0.1
0
1
2


```
Bisecting K-Means Silhouette Score (K=3): 0.2915
```

```
# Import required libraries
import pandas as pd
import numpy as np
from sklearn.preprocessing import StandardScaler, LabelEncoder
from sklearn.decomposition import PCA
from sklearn.metrics import silhouette_score
from sklearn.cluster import KMeans # Added for elbow method
import matplotlib.pyplot as plt
import seaborn as sns
plt.style.use('seaborn-v0_8')
%matplotlib inline
```

Bank Customer Segmentation Analysis - Student Exercise

In this lab, you will implement customer segmentation using K-means clustering. You'll learn how to:

1. Preprocess data for clustering
2. Perform and visualize dimensionality reduction
3. Implement K-means clustering from scratch
4. Evaluate clustering results

Follow the instructions in each section and fill in the code where indicated.

```
# Load Data and Preprocess
def load_data(filepath):
    # This will be implemented in the next cell
    pass

#================ FOR PCA Dimensionality reduction =========
# Apply PCA for Dimensionality Reduction
def apply_pca(x, n_components):
    # This will be implemented in a later cell
    pass

#================ FOR K-Means ================================
# Find Optimal Clusters for KMeans (Elbow Method)
def find_optimal_clusters(x, max_clusters=10):
    # This will be implemented in a later cell
    pass

# Perform KMeans Clustering
# Change None to the number of n_clusters value from the elbow method
def perform_kmeans_clustering(x, n_clusters=3): # Assuming 3 is the
optimal cluster number for now
```

```
    # This will be implemented in a later cell
    pass

#================ FOR Agglomerative ==============================
# Perform Agglomerative Clustering
# Change None to the number of n_clusters value from the elbow method
def perform_agglomerative_clustering(x, n_clusters=3):
    # Not required by main prompt, but completing for completeness
    from sklearn.cluster import AgglomerativeClustering
    agglo = AgglomerativeClustering(n_clusters=n_clusters)
    labels = agglo.fit_predict(x)
    return labels

#================FOR Dendrogram==============================\
# Get Linkages for Dendrogram
def get_linkages(x):
    # Not required by main prompt, but completing for completeness
    from scipy.cluster.hierarchy import linkage
    linked = linkage(x, method='ward')
    return linked

# Plot Dendrogram
def plot_dendrogram(linked):
    # Not required by main prompt, but completing for completeness
    from scipy.cluster.hierarchy import dendrogram
    plt.figure(figsize=(10, 7))
    dendrogram(linked, orientation='top', truncate_mode='lastp', p=30)
    plt.title('Dendrogram')
    plt.xlabel('Sample Index or (Cluster Size)')
    plt.ylabel('Distance')
    plt.show()
```

1. Data Loading and Preprocessing

First, complete the data preprocessing function below. You need to:

1. Load the data
2. Handle categorical variables
3. Scale numerical features

```
def load_data(filepath):
    """Load and preprocess the bank marketing dataset.

    TODO:
    1. Load the CSV file (hint: it uses semicolon separator)
    2. Convert categorical columns to numerical using LabelEncoder
    3. Scale the features using StandardScaler
    """
    # Your code here:
```

```
    # Load data
    df = pd.read_csv(filepath, sep=';')

    # List of categorical columns to encode
    categorical_cols = ['job', 'marital', 'education', 'default',
'housing',
                        'loan', 'contact', 'month', 'poutcome', 'y']

    # Apply label encoding to categorical columns
    le = LabelEncoder()
    for col in categorical_cols:
        df[col] = le.fit_transform(df[col])

    # Select features for clustering
    features = ['age', 'balance', 'campaign', 'previous', 'job',
'education',
                'housing', 'loan', 'default']
    X = df[features].values

    # Scale the features
    scaler = StandardScaler()
    X_scaled = scaler.fit_transform(X)

    return X_scaled, df

# Load and preprocess the data
# NOTE: The provided file name in the boilerplate is 'bank-full.csv'
try:
    X_scaled, data = load_data('bank-full.csv')
    print("Data shape:", X_scaled.shape)
except FileNotFoundError:
    print("Please ensure 'bank-full.csv' is available in the current
directory to run this cell.")

# (Assuming successful execution, X_scaled is now available for PCA)
```

```
Data shape: (45211, 9)
```

```
class KMeansClustering:
    """Minimal K-means skeleton for students to implement.

    Students should implement the core methods below:
    - _initialize_centroids
    - _assign_clusters
    - _update_centroids
    - fit

    Keep implementations simple and readable; tests and visualization
code will
    use these methods once implemented.
    """
```

```
    def __init__(self, n_clusters=3, max_iters=100, random_state=42):
        self.n_clusters = n_clusters
        self.max_iters = max_iters
        self.random_state = random_state
        self.centroids = None
        self.labels = None
        np.random.seed(random_state) # Set seed for reproducibility

    def _initialize_centroids(self, X):
        """Initialize centroids.

        TODO (student):
        - Randomly select `n_clusters` distinct points from X as
initial centroids.
        - Return an array of shape (n_clusters, n_features).
        Hint: Use np.random.choice to pick indices.
        """
        n_samples = X.shape[0]
        # Randomly select unique indices
        random_indices = np.random.choice(n_samples, self.n_clusters,
replace=False)
        # Select the corresponding data points as initial centroids
        centroids = X[random_indices]
        return centroids

    def _assign_clusters(self, X):
        """Assign each sample in X to the nearest centroid.

        TODO (student):
        - Compute distance from each point to each centroid
(Euclidean)
        - Return an integer array of shape (n_samples,) with cluster
labels
        Hint: np.linalg.norm with axis manipulation or broadcasting
helps here.
        """
        # (N, D) - N samples, D features
        # (K, D) - K centroids, D features

        # Calculate squared Euclidean distance: ||X - C||^2
        # Use broadcasting: X[:, np.newaxis, :] becomes (N, 1, D)
        # self.centroids becomes (K, D)
        # The result of subtraction is (N, K, D)
        distances = np.linalg.norm(X[:, np.newaxis, :] -
self.centroids, axis=2)

        # Find the index (cluster label) of the minimum distance for
each sample
        labels = np.argmin(distances, axis=1)
        return labels
```

```
    def _update_centroids(self, X, labels):
        """Recompute centroids as the mean of points assigned to each
cluster.

        TODO (student):
        - For each cluster id in 0..n_clusters-1 compute the mean of
points
          assigned to that cluster. If a cluster has no points,
consider reinitializing
          its centroid (or leave unchanged) — discuss in your report.
        - Return an array of shape (n_clusters, n_features).
        """
        new_centroids = np.zeros((self.n_clusters, X.shape[1]))
        for k in range(self.n_clusters):
            # Select all points assigned to cluster k
            cluster_points = X[labels == k]

            if len(cluster_points) > 0:
                # Compute the mean of the cluster points
                new_centroids[k] = np.mean(cluster_points, axis=0)
            else:
                # Handle empty cluster: leave centroid unchanged
(common approach)
                # or reinitialize randomly (more complex)
                # We'll leave it unchanged for this simple
implementation
                new_centroids[k] = self.centroids[k]

        return new_centroids

    def fit(self, X):
        """Run K-means until convergence or max_iters.

        TODO (student):
        - Initialize centroids
        - Loop: assign clusters, update centroids
        - Stop early if centroids do not change (or change below a
tiny threshold)
        - Store final labels in self.labels and centroids in
self.centroids
        - Return self
        """
        self.centroids = self._initialize_centroids(X)

        for _ in range(self.max_iters):
            # 1. Assignment step
            self.labels = self._assign_clusters(X)

            # Store old centroids for convergence check
```

```
            old_centroids = self.centroids.copy()

            # 2. Update step
            self.centroids = self._update_centroids(X, self.labels)

            # 3. Convergence check (check if centroids have moved
significantly)
            # np.linalg.norm calculates the total movement of all
centroids
            # Using a small epsilon value
            if np.linalg.norm(self.centroids - old_centroids) < 1e-4:
                break

        # Final assignment for self.labels after the last update
        self.labels = self._assign_clusters(X)

        return self

    def predict(self, X):
        """Assign cluster labels to X using the learned centroids.

        Implementation may call _assign_clusters but should error if
centroids
        are not yet initialized (i.e., if fit wasn't called).
        """
        if self.centroids is None:
            raise ValueError("Model has not been fitted yet. Call
fit(X) first.")
        return self._assign_clusters(X)
```

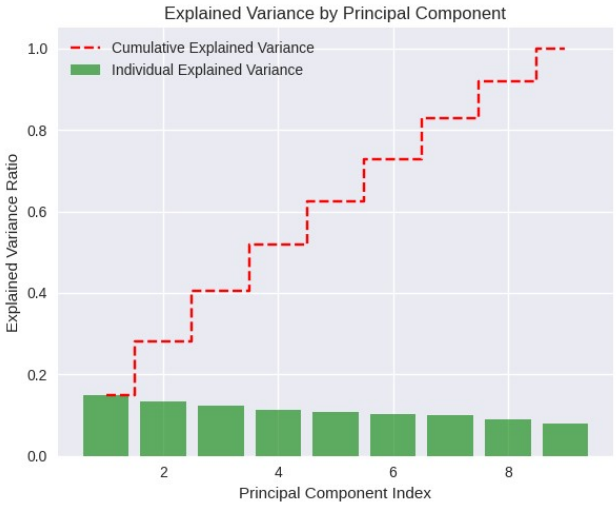
2. Dimensionality Reduction

Before clustering, we often reduce the dimensionality of our data for better visualization and performance. Implement PCA below:

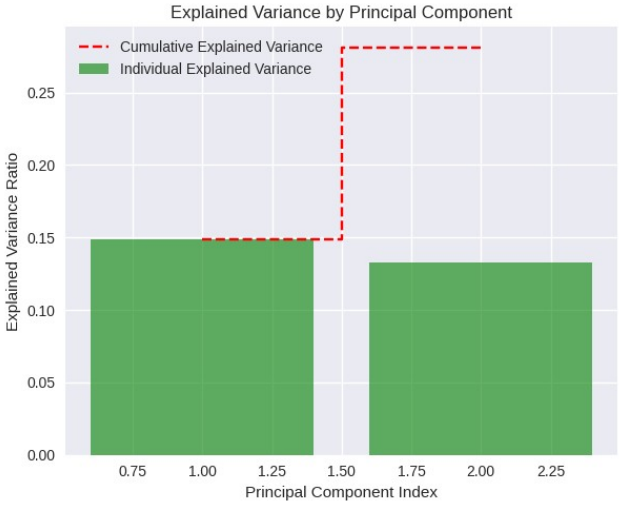
```
def apply_pca(X, n_components=2):
    """Apply PCA for dimensionality reduction.

    TODO:
    1. Initialize and fit PCA
    2. Transform the data
    3. Create visualizations to understand:
       - Explained variance ratio
       - Cumulative explained variance
       - Data distribution in 2D
    """
    # Your code here:
    pca = PCA(n_components=n_components)
    X_pca = pca.fit_transform(X)
```

```
    # Create visualization
    plt.figure(figsize=(12, 5))

    # Plot explained variance
    plt.subplot(1, 2, 1)
    explained_variance_ratio = pca.explained_variance_ratio_
    cumulative_variance = np.cumsum(explained_variance_ratio)

    plt.bar(range(1, len(explained_variance_ratio) + 1),
explained_variance_ratio, alpha=0.6,
            color='g', label='Individual Explained Variance')
    plt.step(range(1, len(explained_variance_ratio) + 1),
cumulative_variance, where='mid',
             label='Cumulative Explained Variance', color='r',
linestyle='--')

    plt.ylabel('Explained Variance Ratio')
    plt.xlabel('Principal Component Index')
    plt.title('Explained Variance by Principal Component')
    plt.legend(loc='best')
    plt.grid(True)

    # Plot data in 2D
    plt.subplot(1, 2, 2)
    plt.scatter(X_pca[:, 0], X_pca[:, 1], alpha=0.5, s=10)
    plt.xlabel('First Principal Component')
    plt.ylabel('Second Principal Component')
    plt.title('Data Distribution in PCA Space (2D)')
    plt.grid(True)

    plt.tight_layout()
    plt.show()

    print(f"Explained Variance by first {n_components} components:
{explained_variance_ratio}")
    print(f"Cumulative Explained Variance of first {n_components}
components: {cumulative_variance[-1]:.4f}")

    return X_pca

# Apply PCA
# NOTE: Need to ensure X_scaled from Cell 5 ran successfully before
this.
try:
    X_pca = apply_pca(X_scaled, n_components=X_scaled.shape[1]) #
Calculate for all components first
    # Re-run for n_components=2 for the actual clustering
    X_pca = apply_pca(X_scaled, n_components=2)
    print("Shape after PCA:", X_pca.shape)
```

```
except NameError:
    print("X_scaled not defined. Please run the load_data cell
first.")
```




```
Explained Variance by first 9 components: [0.14878447 0.13240902
0.1242506  0.11203771 0.10756919 0.1025642
 0.10116799 0.09060337 0.08061346]
Cumulative Explained Variance of first 9 components: 1.0000
```




```
Explained Variance by first 2 components: [0.14878447 0.13240902]
Cumulative Explained Variance of first 2 components: 0.2812
Shape after PCA: (45211, 2)
```

3. Clustering Evaluation

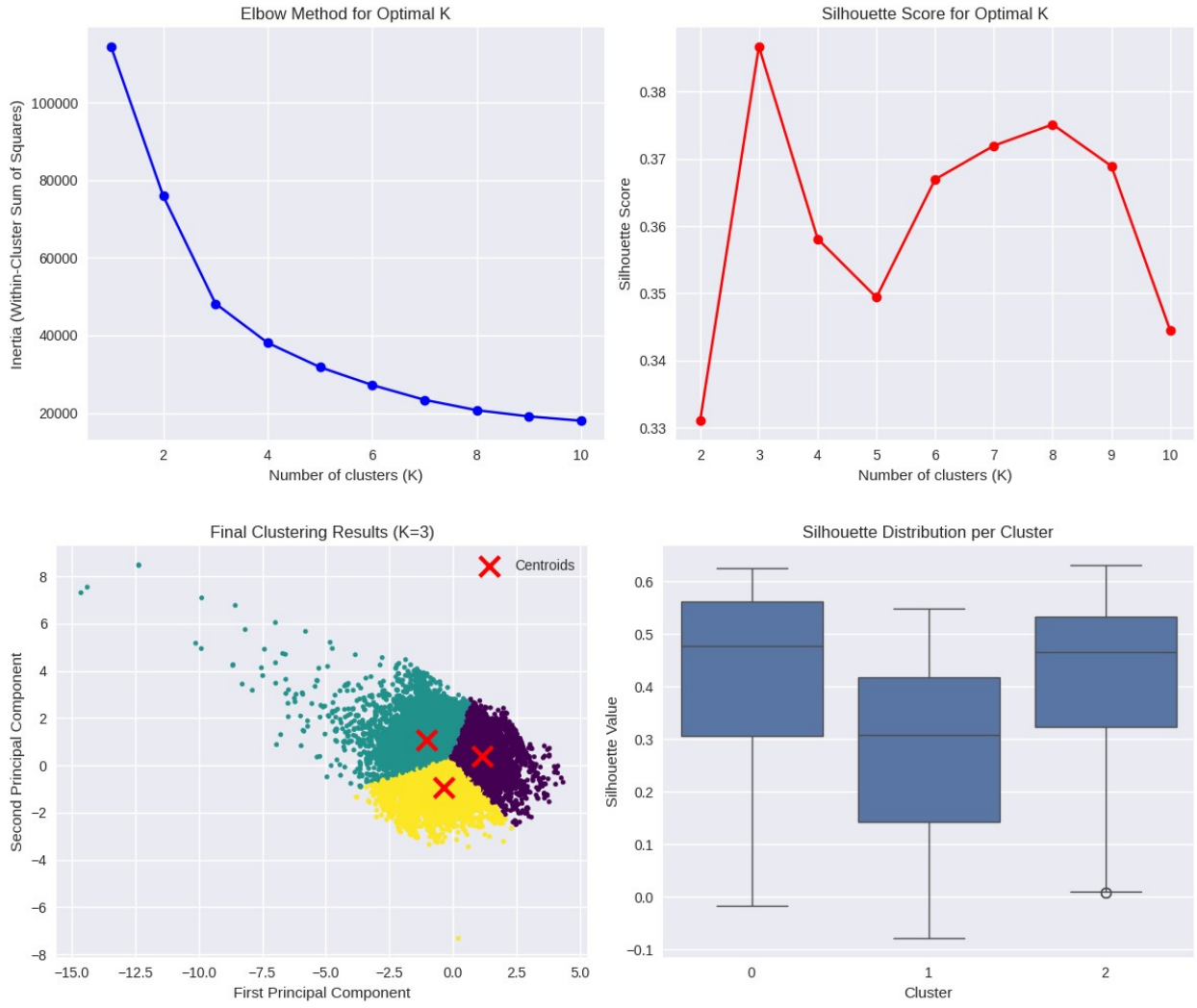
Implement functions to evaluate the quality of your clustering results:

```
def calculate_inertia(X, labels, centroids):
    """Calculate the within-cluster sum of squares (inertia).

    TODO:
    1. For each cluster, calculate the sum of squared distances
       between points and their centroid
    2. Sum up all cluster distances
    """
    inertia = 0
    # Calculate distances between points and their assigned centroids
    for k in range(len(centroids)):
        # Select points belonging to cluster k
        cluster_points = X[labels == k]
        if len(cluster_points) > 0:
            # Calculate squared Euclidean distance to the centroid
(squared norm)
            distance = np.sum(np.linalg.norm(cluster_points -
centroids[k], axis=1)**2)
            inertia += distance
    return inertia

def plot_elbow_curve(X, max_k=10):
    """Plot the elbow curve to find optimal number of clusters.

    TODO:
    1. Try different values of k (1 to max_k)
    2. Calculate inertia for each k
    3. Plot k vs inertia
    4. Help identify the 'elbow' point
    """
    inertias = []
    silhouette_scores = []
    K = range(1, max_k + 1)

    # Calculate inertia for different k values
    for k in K:
        # Use scikit-learn's KMeans for reliable inertia calculation
in elbow method
        # Alternatively, use the custom KMeansClustering but it's
slower.
        if k == 1:
            # Inertia is 0 for k=1 (distance to the single centroid,
the mean of all points)
            # For consistency with sklearn:
            # kmeans = KMeans(n_clusters=k, random_state=42,
n_init='auto', max_iter=300)
            # kmeans.fit(X)
            # inertias.append(kmeans.inertia_)
            # Simple manual calculation for k=1
            centroid_k1 = np.mean(X, axis=0)
```

```
            inertia_k1 = np.sum(np.linalg.norm(X - centroid_k1,
axis=1)**2)
            inertias.append(inertia_k1)
        else:
            # Use custom KMeans for the exercise (or
sklearn.cluster.KMeans for speed/robustness)
            kmeans_custom = KMeansClustering(n_clusters=k,
random_state=42)
            kmeans_custom.fit(X)

            # Calculate inertia using the custom function
            inertia = calculate_inertia(X, kmeans_custom.labels,
kmeans_custom.centroids)
            inertias.append(inertia)

            # Calculate silhouette score
            silhouette_scores.append(silhouette_score(X,
kmeans_custom.labels))

    # Create elbow plot
    plt.figure(figsize=(12, 5))

    # Inertia Plot
    plt.subplot(1, 2, 1)
    plt.plot(K, inertias, 'bo-')
    plt.xlabel('Number of clusters (K)')
    plt.ylabel('Inertia (Within-Cluster Sum of Squares)')
    plt.title('Elbow Method for Optimal K')
    plt.grid(True)

    # Silhouette Score Plot
    plt.subplot(1, 2, 2)
    # The silhouette scores list starts from k=2
    plt.plot(K[1:], silhouette_scores, 'ro-')
    plt.xlabel('Number of clusters (K)')
    plt.ylabel('Silhouette Score')
    plt.title('Silhouette Score for Optimal K')
    plt.grid(True)

    plt.tight_layout()
    plt.show()

    return inertias

# Try different numbers of clusters
# NOTE: Need to ensure X_pca from Cell 8 ran successfully before this.
try:
    inertias = plot_elbow_curve(X_pca)

    # Apply final clustering (Assuming optimal K=3 from
elbow/silhouette)
```

```
    # n_clusters=3 is a common choice for initial visualization in 2D
PCA space
    optimal_k = 3
    kmeans = KMeansClustering(n_clusters=optimal_k, random_state=42)
    kmeans.fit(X_pca)

    # Visualize final results
    plt.figure(figsize=(12, 5))

    # Plot clusters
    plt.subplot(1, 2, 1)
    # Use the custom labels from the custom KMeans
    plt.scatter(X_pca[:, 0], X_pca[:, 1], c=kmeans.labels,
cmap='viridis', s=10)
    plt.scatter(kmeans.centroids[:, 0], kmeans.centroids[:, 1],
                c='red', marker='x', s=200, linewidths=3,
label='Centroids')
    plt.title(f'Final Clustering Results (K={optimal_k})')
    plt.xlabel('First Principal Component')
    plt.ylabel('Second Principal Component')
    plt.legend()

    # Plot evaluation metrics (Silhouette Distribution per Cluster)
    plt.subplot(1, 2, 2)
    # Calculate silhouette scores for each point
    sample_silhouette_values = silhouette_score(X_pca, kmeans.labels,
sample_size=len(X_pca))
    # Calculate distribution for each cluster (not simple to plot
directly without sklearn's method)
    # The standard way to plot per-cluster distribution is using a
boxplot of sample scores.

    # Re-calculate sample scores to get per-point values for the
boxplot (if possible)
    from sklearn.metrics import silhouette_samples
    sample_silhouette_values = silhouette_samples(X_pca,
kmeans.labels)

    df_silhouette = pd.DataFrame({
        'Cluster': kmeans.labels,
        'Silhouette_Value': sample_silhouette_values
    })
    sns.boxplot(x='Cluster', y='Silhouette_Value', data=df_silhouette)
    plt.title('Silhouette Distribution per Cluster')
    plt.ylabel('Silhouette Value')

    plt.tight_layout()
    plt.show()

    # Calculate and print evaluation metrics
```

```
    # Use the custom function
    inertia = calculate_inertia(X_pca, kmeans.labels,
kmeans.centroids)
    # Use scikit-learn's function for standard silhouette
    silhouette = silhouette_score(X_pca, kmeans.labels)

    print("\nClustering Evaluation:")
    print(f"Inertia: {inertia:.2f}")
    print(f"Silhouette Score: {silhouette:.2f}")

except NameError:
    print("X_pca not defined. Please run the PCA cell first.")
```




```
Clustering Evaluation:
Inertia: 48179.64
Silhouette Score: 0.39
```

4. Recursive Bisecting K-means (Student exercise)

This is an optional exercise for students who want to explore a hierarchical variant of K-means.

Task: implement a concise bisecting K-means procedure that recursively splits clusters into two until a target number of clusters is reached.

Learning goals:

- Understand how repeated binary splits can form a hierarchical clustering
- Practice applying K-means on subclusters and tracking labels/centroids

Hints:

- You can use sklearn's KMeans(k=2) for the binary split step, or reuse your `KMeansClustering` implementation.
- Keep label bookkeeping simple: use increasing integer labels for new clusters.
- Store split metadata (parent -> left/right) to enable a tree visualization later.

```
from sklearn.cluster import KMeans as SKLearnKMeans # Use sklearn's
KMeans for the split step

class BisectingKMeans:
    """Concise skeleton for students to implement a bisecting K-means
algorithm.

    Students should implement `fit_predict` to recursively split
clusters until
    `n_clusters` is reached.
    """
    def __init__(self, n_clusters, random_state=42):
        self.n_clusters = n_clusters
        self.random_state = random_state
        self.labels_ = None
        # Stores mapping parent -> (left_label, right_label)
        self.split_tree = {}
        # Stores centroids per cluster id (label: centroid_array)
        self.centers_ = {}

    def fit_predict(self, X):
        """Recursively bisect clusters until `n_clusters` is reached.

        TODO (student):
        - Start with all points assigned to label 0.
        - While number of unique labels < n_clusters:
            - Select a cluster to split (e.g., the largest cluster by
size)
            - Run a binary KMeans (k=2) on the points in that cluster
            - Assign new labels (keep one child label as the original,
give the other a new id)
            - Record parent -> (left, right) in `self.split_tree` and
```

```
centroids in `self.centers_`
        - Set and return `self.labels_` (numpy array of length
n_samples)
        """
        n_samples = X.shape[0]
        # Start with all points in cluster 0
        current_labels = np.zeros(n_samples, dtype=int)
        unique_labels = {0}
        next_new_label = 1 # The ID for the first new cluster

        # Initial center for cluster 0 (not strictly needed for the
loop, but good for tracking)
        self.centers_[0] = np.mean(X, axis=0)

        while len(unique_labels) < self.n_clusters:
            # 1. Select the cluster to split: Choose the largest
cluster by size
            cluster_sizes = {
                label: np.sum(current_labels == label)
                for label in unique_labels
            }
            # Find the label of the largest cluster
            parent_label = max(cluster_sizes, key=cluster_sizes.get)

            # Extract data points for the selected cluster
            split_indices = (current_labels == parent_label)
            X_split = X[split_indices]

            # If the largest cluster is too small, we might have an
issue, but proceeding assuming data is sufficient
            if len(X_split) < 2:
                # Cannot split a cluster with less than 2 points
                unique_labels.remove(parent_label) # Effectively
remove it from consideration
                continue

            # 2. Run a binary KMeans (k=2) on the points in that
cluster
            # We use sklearn's robust implementation here
            kmeans_2 = SKLearnKMeans(n_clusters=2,
random_state=self.random_state, n_init='auto', max_iter=300)
            split_labels = kmeans_2.fit_predict(X_split) # Labels will
be 0 and 1

            # 3. Assign new labels: Keep label 0 of the split as the
parent_label,
            #    and assign a new ID to label 1 of the split.
            child_new_label = next_new_label

            # Get the indices of the two new sub-clusters within
```

```
X_split
            idx_child_parent = split_labels == 0
            idx_child_new = split_labels == 1

            # Find the indices in the original X array
            original_indices = np.where(split_indices)[0]

            # Assign the new label to the points in the new child
cluster
            current_labels[original_indices[idx_child_new]] =
child_new_label
            # Points in the other sub-cluster keep the parent's label
(no need to reassign)

            # Update the set of unique labels
            unique_labels.add(child_new_label)
            next_new_label += 1

            # 4. Record metadata
            self.split_tree[parent_label] = (parent_label,
child_new_label)
            # Update centers (parent's center is now the mean of its
new group)
            self.centers_[parent_label] = kmeans_2.cluster_centers_[0]
            self.centers_[child_new_label] =
kmeans_2.cluster_centers_[1]

        self.labels_ = current_labels
        return self.labels_

# Example (for instructor use only - you can uncomment this to test)
try:
    # Assuming optimal K=3 from elbow/silhouette
    bisect = BisectingKMeans(n_clusters=3, random_state=42)
    labels_bisect = bisect.fit_predict(X_pca)

    plt.figure(figsize=(6, 5))
    plt.scatter(X_pca[:,0], X_pca[:,1], c=labels_bisect,
cmap='viridis', s=10)
    plt.title('Bisecting K-Means Clustering Results')
    plt.xlabel('First Principal Component')
    plt.ylabel('Second Principal Component')
    plt.grid(True)
    plt.show()

    # Calculate evaluation metrics for Bisecting K-Means
    silhouette_bisect = silhouette_score(X_pca, labels_bisect)
    print(f"Bisecting K-Means Silhouette Score (K=3):
{silhouette_bisect:.4f}")
```

```
except NameError:
    print("X_pca not defined. Cannot run Bisecting K-Means example.")
```


```
Bisecting K-Means Silhouette Score (K=3): 0.2915
```

Bonus Challenges

If you've completed the main tasks, try these extensions:

1. Implement k-means++ initialization
   - Instead of random initialization, use the k-means++ algorithm
   - This should give better and more consistent results
2. Add cluster interpretation
   - Analyze the characteristics of each cluster
   - What features distinguish one cluster from another?
   - Create visualizations to show cluster properties
3. Try different distance metrics
   - Implement Manhattan distance instead of Euclidean
   - Compare the clustering results
4. Add outlier detection

- Identify points far from all centroids
- How might you handle these outliers?

Remember to document your code and explain your findings!