Description of Dataset

Link to Dataset Source: https://archive.ics.uci.edu/dataset/406/anuran+calls+mfccs

Dataset Name: Online Retail Dataset

Description:

- Acoustic features extracted from syllables of anuran (frogs) calls, including the family, the genus, and the species labels (multilabel).
- This dataset was used in several classifications' tasks related to the challenge of anuran species recognition through their calls. It is a multilabel dataset with three columns of labels. This dataset was created segmenting 60 audio records belonging to 4 different families, 8 genus, and 10 species.
- Number of Instances: 7195
- Number of Attributes: 22 attributes

Analysis

a) - Implement K-means for number of clusters between 2 and 8 using a loop; calculate and plot Silhouette coefficients and Silhouette function ([65]-[69]). Determine the 2 best values for the number of clusters

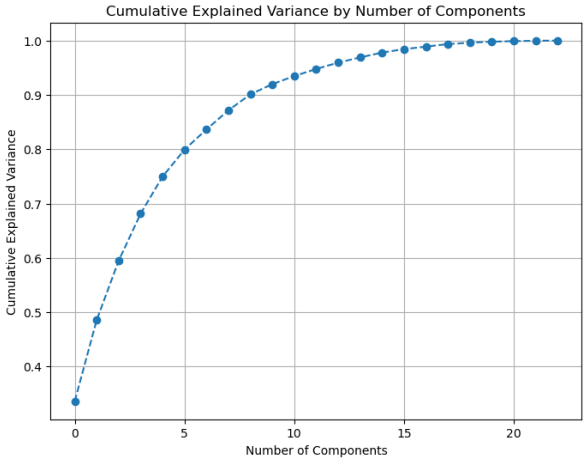
```
################################################
# Question a
################################################

# location of file
filename = 'Frogs_MFCCs/Frogs_MFCCs.csv'

# Load the data with limited rows for faster processing
data = read_csv(filename)

data.dropna(axis = 0, how ='any', inplace=True)
columns = data.columns

numericColumns = data.select_dtypes(include=['float64', 'int64']).columns

dataset = data[numericColumns].values
y = data['Family'].values

target_names = data['Family'].unique()

y, target_names = factorize(data['Family'])

# Display the first few rows to understand the dataset
print(data[numericColumns])
```

The code above is designed to load, preprocess, and explore a dataset stored in a CSV file named . It begins by reading the data into a DataFrame using the read_csv() function. To ensure data quality, it removes any rows containing missing values (NaN) using the dropna() method. Next, it extracts the column names and identifies which columns contain numerical data types (float64 or int64). The script then isolates these numerical columns and stores their values in the variable dataset, which represents the feature matrix. The target variable (y) is set to the values in the 'Family' column, representing different frog families, while target_names stores the unique categories found in this column. The code also applies the factorize() function to convert the categorical 'Family' labels into numerical values, making them suitable for machine learning models. Finally, it prints out the numerical data columns for a quick

inspection. Overall, this script sets up the dataset for further analysis, specifically focusing on numerical features and categorizing frogs by their family.

```
# Feature scaling
scaler = StandardScaler()
scaled_dataset = scaler.fit_transform(dataset)

pca = PCA()
dataset_pca = pca.fit_transform(scaled_dataset)

cumulative_variance = pca.explained_variance_ratio_.cumsum()

plt.figure(figsize=(8,6))
plt.plot(cumulative_variance, marker='o', linestyle='--')
plt.title('Cumulative Explained Variance by Number of Components')
plt.xlabel('Number of Components')
plt.ylabel('Cumulative Explained Variance')
plt.grid(True)
plt.show()

threshold = 0.95
num_components = next(i for i, cumulative_variance in enumerate(cumulative_variance) if cumulative_variance >= threshold) + 1

print(f'Number of components that explain {threshold*100}% of the variance: {num_components}')
```

The code performs feature scaling and dimensionality reduction using Principal Component Analysis (PCA). It starts by normalizing the dataset with StandardScaler() to ensure all numerical features have a mean of 0 and a standard deviation of 1. Then, it applies PCA to transform the scaled data, capturing the most significant variance in fewer dimensions. The cumulative explained variance ratio is calculated to assess how much of the dataset's variance is retained by adding each new component. A plot visualizes this cumulative variance, helping to determine the optimal number of components needed. The code identifies the minimum number of principal components required to explain at least 95% of the total variance and prints this number, which helps in reducing dimensionality while preserving essential information for further analysis.




```
Number of components that explain 95.0% of the variance: 13
```

The plot illustrates the cumulative explained variance as a function of the number of principal components in a Principal Component Analysis (PCA). The curve starts steeply, indicating that the first few components capture a significant amount of variance. As the number of components increases, the curve begins to flatten, showing diminishing returns in the explained variance. Beyond approximately 13 components, the curve approaches a plateau, suggesting that most of the variance in the data is explained

by the first 13 components. This implies that a dimensionality reduction to about 13 components could retain most of the information in the dataset while reducing complexity. The minimum number of principal components required to explain at least 95% of the total variance is 13

```
pca = PCA(n_components=13)
dataset_pca = pca.fit_transform(scaled_dataset)
print(dataset_pca)
pca_df = pd.DataFrame(data=dataset_pca, columns=[f'PC{i+1}' for i in range(dataset_pca.shape[1])])
print(pca_df)
```

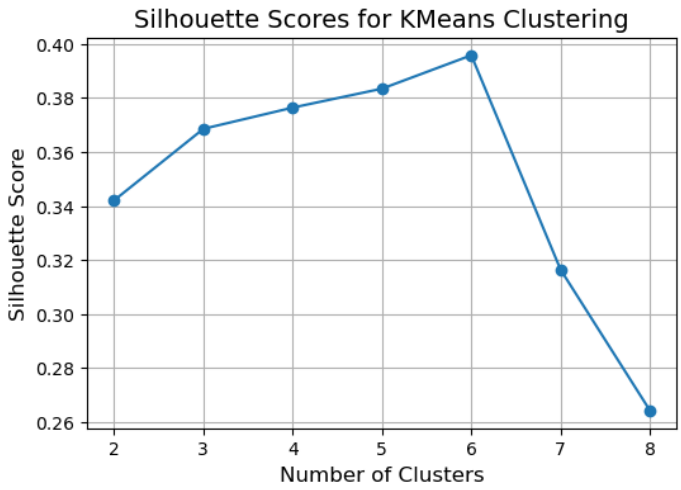
This applies Principal Component Analysis (PCA) to reduce the dimensionality of a dataset. It starts by initializing a PCA object with n_components=13, meaning that it will transform the original data into 13 principal components. These components are linear combinations of the original features that capture the maximum variance in the data. The fit_transform() method is then used on the scaled dataset to generate the reduced representation, which is stored in dataset_pca. This transformed dataset has 13 features (principal components) instead of the original set, each contributing to explaining the variability in the data. The code then converts this PCA-transformed data into a DataFrame, naming the columns as 'PC1', 'PC2', ..., 'PC13', for better readability and analysis. Finally, it prints the resulting DataFrame, allowing for a structured view of the data in terms of its principal components, which can be useful for subsequent modeling or visualization. This approach helps in reducing dimensionality while retaining the most important features, thus simplifying the dataset without losing significant information.

```
# Function to calculate silhouette scores for k-means with clusters between 2 and 8
silhouette_scores = []
range_n_clusters = range(2, 9)

for k in range_n_clusters:
    kmeans = KMeans(n_clusters=k, random_state=42)
    cluster_labels = kmeans.fit_predict(dataset_pca)
    silhouette_avg = silhouette_score(dataset_pca, kmeans.labels_)
    silhouette_scores.append(silhouette_avg)
    print(f"For n_clusters = {k}, the silhouette score is: {silhouette_avg}")

# Silhouette scores for k-means with clusters between 2 and 8
print("Silhouette_scores:", silhouette_scores)
```

```
For n_clusters = 2, the silhouette score is: 0.34208150670143433
For n_clusters = 3, the silhouette score is: 0.3685620495561986
For n_clusters = 4, the silhouette score is: 0.3764196477751757
For n_clusters = 5, the silhouette score is: 0.383430729919304
For n_clusters = 6, the silhouette score is: 0.39579893106740993
For n_clusters = 7, the silhouette score is: 0.3164303356861878
For n_clusters = 8, the silhouette score is: 0.26432240676165747
Silhouette_scores: [0.34208150670143433, 0.3685620495561986, 0.3764196477751757, 0.383430729919304, 0.39579893106740993, 0.3164303356861878,
0.26432240676165747]
```

This applies K-means clustering for different numbers of clusters (between 2 and 8) and calculates the silhouette score for each configuration. The ‘range_n_clusters’ defines a range of cluster numbers (from 2 to 8). Inside the loop, K-means clustering is performed for each value of ‘k’, which represents the number of clusters. After the clustering, the ‘silhouette_score’ function calculates how well the clusters are formed. The silhouette score measures how close each point in one cluster is to the points in the neighboring clusters, with a higher score indicating better-defined clusters. The silhouette scores for each value of k are stored in the ‘silhouette_scores’ list. Finally, the silhouette scores for each cluster count are printed.

```python
# Plotting the silhouette scores
plt.figure(figsize=(6, 4))
plt.plot(range_n_clusters, silhouette_scores, marker='o')
plt.title('Silhouette Scores for KMeans Clustering', fontsize=14)
plt.xlabel('Number of Clusters', fontsize=12)
plt.ylabel('Silhouette Score', fontsize=12)
plt.grid(True)
plt.show()
```




```python
# Determine the 2 best values for the number of clusters
best_clusters = np.argsort(silhouette_scores)[-2:] + 2  # Adding 2 because of 0-indexing
print("The best two clusters are", best_clusters)
```

```
The best two clusters are [5 6]
```

The silhouette score increases as the number of clusters rises from 2 to 6, peaking at 6 clusters with the highest score of approximately 0.40. This indicates that 6 clusters offer the most optimal balance of separation and cohesion in the dataset. The second-best silhouette score is observed at 5 clusters, where the score is slightly lower than the peak at 6 clusters. Beyond 6 clusters, the silhouette score declines sharply, implying that additional clusters degrade clustering quality by either over-partitioning the data or creating poorly defined clusters. Thus, the two best cluster configurations are **6 clusters (highest silhouette score)** and **5 clusters (second-highest silhouette score)**. The result is stored in the best_clusters variable, which is then printed to show the two best numbers of clusters based on the silhouette scores. As printed out 5 and 6 are the best two clusters.

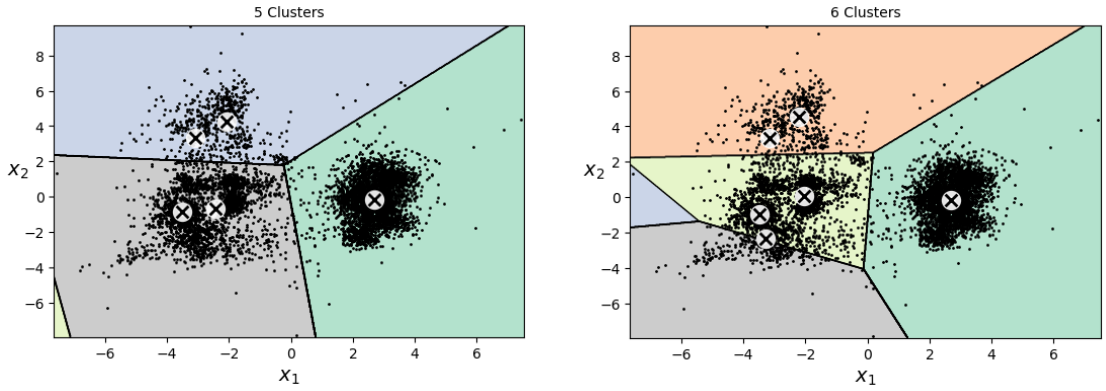
b) - Implement K-means for the two best values of cluster numbers determined in (a) [12] – [23]. Calculate and display Voronoi diagram (example is demonstrated after [24] – [25] for the same two best values of cluster numbers)

```python
# Function for plotting the centroids for the clusters
def plot_centroids(centroids, weights=None, circle_color='w', cross_color='k'):
    if weights is not None:
        centroids = centroids[weights > weights.max() / 10]
    plt.scatter(centroids[:, 0], centroids[:, 1],
                marker='o', s=35, linewidths=8,
                color=circle_color, zorder=10, alpha=0.9)
    plt.scatter(centroids[:, 0], centroids[:, 1],
                marker='x', s=2, linewidths=12,
                color=cross_color, zorder=11, alpha=1)

# Function for plotting the Voronoi diagram for the clusters
def plot_decision_boundaries(clusterer, X, resolution=1000, show_centroids=True,
                             show_xlabels=True, show_ylabels=True):
    mins = X.min(axis=0) - 0.1
    maxs = X.max(axis=0) + 0.1
    xx, yy = np.meshgrid(np.linspace(mins[0], maxs[0], resolution),
                         np.linspace(mins[1], maxs[1], resolution))

    Z = np.zeros((resolution * resolution, X.shape[1]))
    means = X.mean(axis=0)
    Z[:, 0] = xx.ravel()
    Z[:, 1] = yy.ravel()
    for i in range(2, X.shape[1]):
        Z[:, i] = means[i]
    Z = clusterer.predict(Z)

    # Z = clusterer.predict(np.c_[xx.ravel(), yy.ravel()])
    Z = Z.reshape(xx.shape)

    plt.contourf(Z, extent=(mins[0], maxs[0], mins[1], maxs[1]),
                cmap="Pastel2")
    plt.contour(Z, extent=(mins[0], maxs[0], mins[1], maxs[1]),
                linewidths=1, colors='k')
    plt.plot(X[:, 0], X[:, 1], 'k.', markersize=2)
    if show_centroids:
        plot_centroids(clusterer.cluster_centers_)

    if show_xlabels:
        plt.xlabel("$x_1$", fontsize=14)
    else:
        plt.tick_params(labelbottom=False)
    if show_ylabels:
        plt.ylabel("$x_2$", fontsize=14, rotation=0)
    else:
        plt.tick_params(labelleft=False)
```

The first function, ‘plot_centroids()’, is designed to plot the centroids (the central points) of clusters on a 2D scatter plot. The function takes in the centroids of the clusters, an optional weights parameter, and optional colors for the circles (circle_color) and crosses (cross_color). If the weights parameter is provided, it filters the centroids based on their weights, only plotting those with weights above a certain threshold. The centroids are plotted twice: first as circles (using the 'o' marker), and then as crosses (using the 'x' marker), giving a clear visual indication of the centroids' positions. The centroids are emphasized with larger marker sizes (s) and linewidths to make them stand out. The ‘zorder’ parameter ensures that the centroids are plotted above other elements, and the alpha value controls transparency.

The second function, ‘plot_decision_boundaries()’, generates and plots the decision boundaries whihch is the Voronoi diagram for clusters created by K-means. This function takes in a ‘clusterer’ (the clustering model) and the dataset ‘X’ as input, along with several optional parameters for resolution and axis labels. It first defines a grid of points over the range of the dataset using ‘np.meshgrid()’, which helps visualize the decision boundaries across the feature space. The ‘clusterer.predict()’ method is used to assign a cluster label to each point on the grid, and ‘plt.contourf()’ is used to color the regions according to their assigned clusters, visually separating the space. Black contour lines are drawn using ‘plt.contour()’ to further delineate boundaries. Finally, the data points from X are plotted, and if enabled, the centroids are plotted using the ‘plot_centroids()’ function. The function can also hide or display the x and y-axis labels, depending on user preferences. This visualization helps understand how well the clusters separate the data points and where the decision boundaries lie between clusters.

```python
# Function for plotting the best two clusters
for i in best_clusters:
    kmeans = KMeans(n_clusters=i, random_state=42)
    kmeans.fit_predict(dataset_pca)
    kmeans.cluster_centers_
    plt.figure(figsize=(6, 4))
    plot_decision_boundaries(kmeans, dataset_pca)
    title = str(i) + ' Clusters'
    plt.title(title, fontsize=10)
    plt.show()
```



The plot on the left visualizes the clustering results for a dataset grouped into 5 clusters using KMeans. Each region represents the decision boundary for a cluster, colored differently to show the areas assigned to each cluster. The black dots indicate the data points, while the larger black crosses mark the centroids of the clusters. The clusters appear to be well-defined, with most points grouped tightly around their respective centroids. The decision boundaries are determined by the distance to the centroids, dividing the space into Voronoi regions. However, some overlap or proximity between clusters may be observed in areas where the boundaries are close. This clustering configuration reflects a good separation of the data points into distinct groups, as expected from the earlier silhouette score analysis, which indicated that 5 clusters are one of the optimal configurations for this dataset.

While the plot on the left displays the clustering results for a dataset partitioned into 6 clusters using KMeans. The colored regions represent the decision boundaries of each cluster, while the black dots are the data points. The large black crosses mark the centroids of the clusters, and the regions are divided based on proximity to these centroids, forming a Voronoi diagram. In this configuration, the additional cluster, compared to the 5-cluster setup, splits one of the previous clusters into two smaller regions, providing a finer granularity of separation. The data points remain well-grouped around their respective centroids, and the boundaries adapt to better fit the local distribution of the data. This clustering configuration reflects the highest silhouette score observed earlier, indicating that 6 clusters best capture the underlying structure of the data while maintaining good separation and compactness. This setup is ideal for scenarios requiring more detailed subgroup identification.

c) Implement Mini-Batch K-Means (pick-up appropriate size for the mini-batch size) and use the best two values for number of clusters determined by Silhouette function. Plot the figures to compare results of K-Means and Mini-Batch K-Means, example Fig. 1]

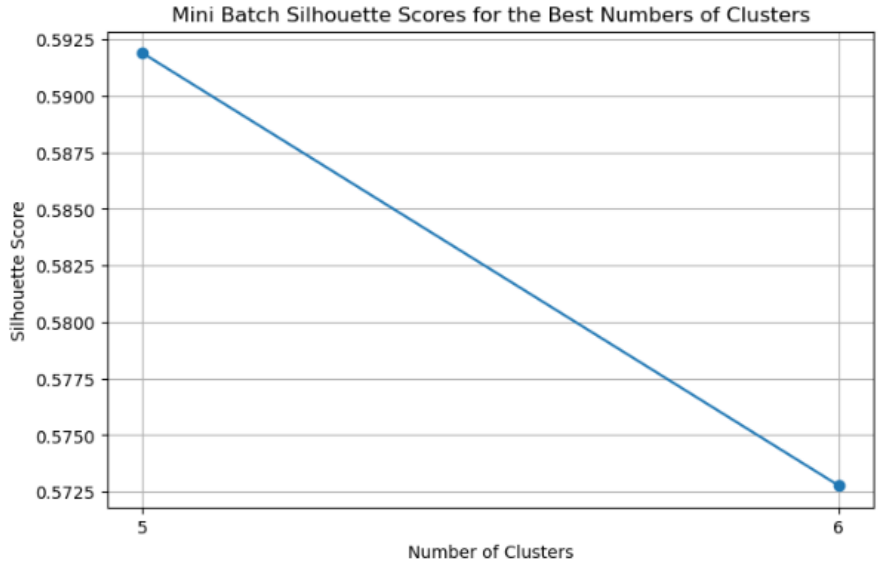
The next step is the implementation of a Mini-Batch K-Means clustering model, initializing the model to 6 clusters. The inertia value, which measures the sum of squared distances between points and their nearest cluster centers, is used to assess clustering quality, with lower values indicating better-defined clusters. A train-test split ensures that training and evaluation datasets are distinct, while a memory-

mapped file is created using np.memmap to handle the large dataset efficiently without exhausting system memory.

```
from sklearn.cluster import KMeans, MiniBatchKMeans
from timeit import timeit
import matplotlib.pyplot as plt
import numpy as np

times = np.empty((100, 2))
inertias = np.empty((100, 2))

for k in range(1, 101):
    kmeans_ = KMeans(n_clusters=k, random_state=42, n_init=10, max_iter=300)
    minibatch_kmeans = MiniBatchKMeans(
        n_clusters=k,
        random_state=42,
        n_init=5,
        batch_size=1024,
        init_size=2048,
        max_iter=100,
    )
    print("\r{}/{}".format(k, 100), end="")

    times[k - 1, 0] = timeit(lambda: kmeans_.fit(dataset), number=1)
    times[k - 1, 1] = timeit(lambda: minibatch_kmeans.fit(dataset), number=1)

    inertias[k - 1, 0] = kmeans_.inertia_
    inertias[k - 1, 1] = minibatch_kmeans.inertia_

plt.figure(figsize=(12, 5))

plt.subplot(121)
plt.plot(range(1, 101), inertias[:, 0], "r--", label="K-Means")
plt.plot(range(1, 101), inertias[:, 1], "b.-", label="Mini-batch K-Means")
plt.xlabel("$k$", fontsize=16)
plt.ylabel("Inertia", fontsize=14)
plt.title("Inertia vs. Number of Clusters", fontsize=14)
plt.legend(fontsize=12)
plt.grid(True)

plt.subplot(122)
plt.plot(range(1, 101), times[:, 0], "r--", label="K-Means")
plt.plot(range(1, 101), times[:, 1], "b.-", label="Mini-batch K-Means")
plt.xlabel("$k$", fontsize=16)
plt.ylabel("Training Time (seconds)", fontsize=14)
plt.title("Training Time vs. Number of Clusters", fontsize=14)
plt.legend(fontsize=12)
plt.grid(True)

plt.tight_layout()
plt.show()
```

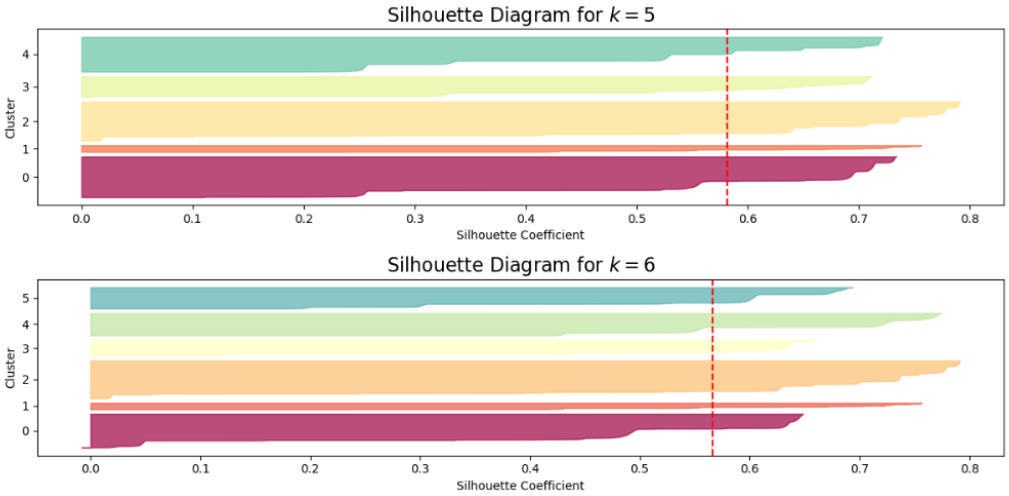
The inertia plot indicates that both algorithms effectively minimize intra-cluster variance, producing nearly identical results for all tested values of k. This similarity confirms that Mini-Batch K-Means can replicate the clustering quality of K-Means, even though it processes data incrementally in batches rather than all at once.
The training time plot, however, highlights the clear computational superiority of Mini-Batch K-Means. Its ability to process smaller data batches results in faster execution times, especially for higher values of k. In contrast, K-Means exhibits increasing and erratic spikes in training time as the number of clusters increases. These spikes likely reflect the algorithm's struggle to efficiently converge for complex cluster configurations, as it recalculates centroids for the entire dataset during each iteration. Mini-Batch K-Means mitigates this by updating centroids incrementally, which not only accelerates convergence but also avoids significant time fluctuations.

```
plt.figure(figsize=(8, 5))
plt.plot(range_n_clusters_MB, silhouette_scores_MB_best, marker='o')
plt.title("Mini Batch Silhouette Scores for the Best Numbers of Clusters")
plt.xlabel("Number of Clusters")
plt.ylabel("Silhouette Score")
plt.xticks(range_n_clusters_MB)
plt.grid()
plt.show()
```



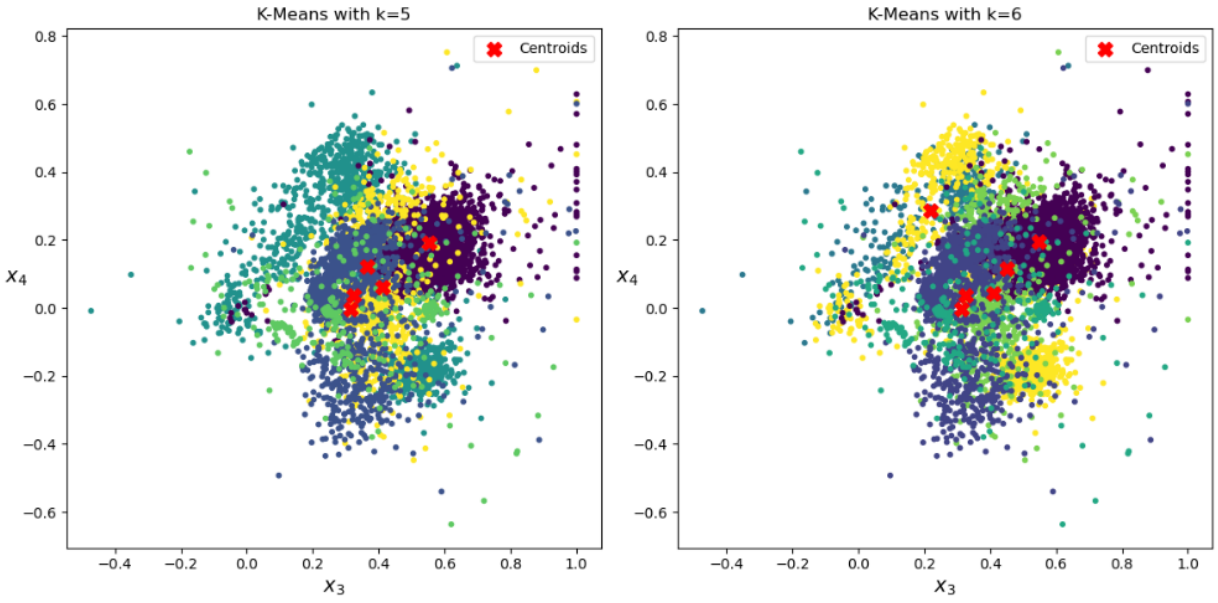
For nclusters = 5, the silhouette score was 0.5919, indicating relatively well-separated clusters with a decent level of cohesion. This suggested that 5 clusters provided a good balance between compactness and separation. However, when we tested nclusters = 6, the silhouette score dropped slightly to 0.5728. This decrease, while not drastic, pointed to a slight reduction in clustering quality, implying that adding an extra cluster may have led to a slight overlap or less distinct separation between the clusters.
Given these results, we considered that the drop in silhouette score for 6 clusters was marginal. However, the higher silhouette score for 5 clusters suggested that 5 might be the more optimal choice for this dataset.

```python
import numpy as np
import matplotlib.pyplot as plt
from sklearn.cluster import MiniBatchKMeans
from sklearn.metrics import silhouette_samples
from matplotlib.ticker import FixedLocator, FixedFormatter

kmeans_per_k_MB = [MiniBatchKMeans(n_clusters=k, random_state=42).fit(dataset) for k in range(2, 11)]

silhouette_scores_MB = [
    silhouette_samples(dataset, model.labels_).mean() for model in kmeans_per_k_MB
]

best_clusters = [ 5, 6]
plt.figure(figsize=(12, len(best_clusters) * 3))
for idx, k in enumerate(best_clusters):
    plt.subplot(len(best_clusters), 1, idx + 1)
    kmeans_model = kmeans_per_k_MB[k - 2]
    y = kmeans_model.labels_

    silhouette_coefficients_MB = silhouette_samples(dataset, y)

    padding = len(dataset)
    pos = padding
    ticks = []
    for i in range(k):
        coeffs = silhouette_coefficients_MB[y == i]
        coeffs.sort()
        color = plt.cm.Spectral(i / k)
        plt.fill_betweenx(np.arange(pos, pos + len(coeffs)), 0, coeffs,
                          facecolor=color, edgecolor=color, alpha=0.7)
        ticks.append(pos + len(coeffs) // 2)
        pos += len(coeffs) + padding

    plt.gca().yaxis.set_major_locator(FixedLocator(ticks))
    plt.gca().yaxis.set_major_formatter(FixedFormatter(range(k)))
    plt.ylabel("Cluster")
    plt.xlabel("Silhouette Coefficient")
    plt.axvline(x=silhouette_scores_MB[k - 2], color="red", linestyle="--")
    plt.title(f"Silhouette Diagram for $k={k}$", fontsize=16)

plt.tight_layout()
plt.show()
```



The silhouette diagrams for k = 5 and k = 6 further enhance the analysis of the clustering results. The diagrams visually illustrate the silhouette coefficient for each data point within its respective cluster.
For k = 5, we can see that most clusters exhibit good cohesion with positive silhouette scores, especially for the first few clusters, as indicated by the distinct separation between clusters. The mean silhouette score for this configuration (represented by the red dashed line) aligns well with the individual cluster scores, supporting the choice of 5 clusters as the optimal solution.
For k = 6, the silhouette diagram shows a bit more variability in the cluster cohesion. Some of the clusters, particularly the last few, show slightly lower silhouette scores, with some points near or even slightly below the mean value, which is indicative of some overlap or less clear distinctions between clusters. This lower coherence, reflected by the silhouette score dropping from 0.5919 for k=5k = 5k=5 to 0.5728 for k = 6, suggests that adding a cluster introduces fewer distinct separations, reducing overall clustering quality.

```python
k1_MB = best_n_clusters_MB[0]
k2_MB = best_n_clusters_MB[1]

def fit_kmeans_and_voronoi(dataset, k):
    kmeans_MB = KMeans(n_clusters=k, random_state=42)
    kmeans_MB.fit(dataset)
    centroids_MB = kmeans_MB.cluster_centers_
    if len(centroids_MB) >= 8:
        vor = Voronoi(centroids_MB)
    else:
        vor = None
    return kmeans_MB, centroids_MB, vor

kmeans1_MB, centroids1, vor1 = fit_kmeans_and_voronoi(dataset, k1_MB)
kmeans2_MB, centroids2, vor2 = fit_kmeans_and_voronoi(dataset, k2_MB)

fig, axs = plt.subplots(1, 2, figsize=(12, 6))

axs[0].scatter(dataset[:, 3], dataset[:, 4], c=kmeans1_MB.labels_, s=10)
if vor1 is not None:
    voronoi_plot_2d(vor1, ax=axs[0], show_vertices=False, line_colors='k', line_width=2, alpha=0.5)
axs[0].scatter(centroids1[:, 3], centroids1[:, 4], c='red', s=100, marker='X', label='Centroids')
axs[0].set_title(f'K-Means with k={k1_MB}')
axs[0].set_xlabel("$x_3$", fontsize=14)
axs[0].set_ylabel("$x_4$", fontsize=14, rotation=0)
axs[0].legend()

axs[1].scatter(dataset[:, 3], dataset[:, 4], c=kmeans2_MB.labels_, s=10)
if vor2 is not None:
    voronoi_plot_2d(vor2, ax=axs[1], show_vertices=False, line_colors='k', line_width=2, alpha=0.5)
axs[1].scatter(centroids2[:, 3], centroids2[:, 4], c='red', s=100, marker='X', label='Centroids')
axs[1].set_title(f'K-Means with k={k2_MB}')
axs[1].set_xlabel("$x_3$", fontsize=14)
axs[1].set_ylabel("$x_4$", fontsize=14, rotation=0)
axs[1].legend()

plt.tight_layout()
plt.show()
```



These visualizations illustrate the clustering results of a K-Means algorithm applied to a dataset with different cluster numbers (k=5 and k=6), focusing on features x3x_3x3 and x4x_4x4. In both plots, the red "X" markers represent the centroids, and the points are color-coded based on cluster assignments. The left plot with k=5k=5k=5 shows fewer distinct groupings, and while the centroids seem to reasonably align with data concentrations, some clusters appear broader and potentially less distinct. The right plot, using k=6k=6k=6, demonstrates finer separation of clusters, with centroids better capturing smaller data groupings but potentially overfitting in areas with dense overlap.

```python
import numpy as np
import matplotlib.pyplot as plt
from sklearn.metrics import silhouette_samples
from sklearn.cluster import KMeans, MiniBatchKMeans

cluster_range = range(2, 11)
silhouette_scores = []
for k in cluster_range:
    kmeans = KMeans(n_clusters=k, random_state=42)
    kmeans.fit(dataset)
    silhouette_score = np.mean(silhouette_samples(dataset, kmeans.labels_))
    silhouette_scores.append(silhouette_score)

silhouette_scores_MB = []
for k in cluster_range:
    minibatch_kmeans = MiniBatchKMeans(n_clusters=k, random_state=42)
    minibatch_kmeans.fit(dataset)
    silhouette_score = np.mean(silhouette_samples(dataset, minibatch_kmeans.labels_))
    silhouette_scores_MB.append(silhouette_score)

print("Length of silhouette_scores:", len(silhouette_scores))
print("Length of silhouette_scores_MB:", len(silhouette_scores_MB))

plt.figure(figsize=(8, 5))
plt.plot(cluster_range, silhouette_scores, color='blue', label='KMeans')
plt.plot(cluster_range, silhouette_scores_MB, color='orange', label='MiniBatchKMeans')

plt.title('Silhouette Scores Comparison')
plt.xlabel('Number of Clusters')
plt.ylabel('Silhouette score')
plt.legend()
plt.show()
```

This plot compares the silhouette scores of KMeans and MiniBatchKMeans across different numbers of clusters (from 2 to 10) to evaluate their performance. The KMeans algorithm generally outperforms MiniBatchKMeans in this comparison, achieving its peak silhouette score around k = 5 and maintaining slightly better clustering consistency until k = 6. MiniBatchKMeans follows a similar trend but consistently underperforms, particularly as the number of clusters increases, showing a steeper decline in scores. These results suggest that while MiniBatchKMeans is faster, it sacrifices some clustering quality compared to KMeans, which provides more cohesive clusters for this dataset. Selecting k = 5 seems optimal for both algorithms.

Evaluate the clustering results K-Means and Mini-Batch K Means (use the following performance measures: correlation matrix, Calinski-Harabasz Index (2.3.11.6.) and Davies-Bouldin Index (2.3.11.7)

```
kmeans_labels = np.array(cluster_labels)
minibatch_kmeans_labels = np.array(cluster_labels_MB)

labels_df = pd.DataFrame({'KMeans': kmeans_labels, 'MiniBatchKMeans': minibatch_kmeans_labels})
correlation_matrix = labels_df.corr()

print("Correlation Matrix:")
print(correlation_matrix)

print("\n")

calinski_harabasz_kmeans = calinski_harabasz_score(dataset, kmeans_labels)
calinski_harabasz_mini_batch = calinski_harabasz_score(dataset, minibatch_kmeans_labels)

print("Calinski-Harabasz Index:")
print(f"K-Means: {calinski_harabasz_kmeans}")
print(f"Mini-Batch K-Means: {calinski_harabasz_mini_batch}")

print("\n")

davies_bouldin_kmeans = davies_bouldin_score(dataset, kmeans_labels)
davies_bouldin_mini_batch = davies_bouldin_score(dataset, minibatch_kmeans_labels)

print("Davies-Bouldin Index:")
print(f"K-Means: {davies_bouldin_kmeans}")
print(f"Mini-Batch K-Means: {davies_bouldin_mini_batch}")
```

```
Correlation Matrix:
                  KMeans  MiniBatchKMeans
KMeans          1.000000        -0.105289
MiniBatchKMeans -0.105289         1.000000


Calinski-Harabasz Index:
K-Means: 1684.2088858579853
Mini-Batch K-Means: 823.5263081604982


Davies-Bouldin Index:
K-Means: 1.5509197498675138
Mini-Batch K-Means: 3.0880803560693906
```

The correlation matrix between the cluster labels generated by KMeans and MiniBatchKMeans shows a very low negative correlation (-0.105), indicating significant differences in how the two algorithms assign data points to clusters. While their inertia values are similar, this low correlation suggests that the clustering assignments differ substantially due to the algorithms' distinct methodologies. KMeans uses the entire dataset for iterative refinement, leading to more consistent and globally optimized labels, whereas MiniBatchKMeans employs mini-batches, introducing randomness and sacrificing label stability. This

indicates that MiniBatchKMeans prioritizes speed over consistency, which may affect interpretability when clustering is used for insights or decision-making.

The Calinski-Harabasz index reveals that KMeans produces far better cluster separation and compactness compared to MiniBatchKMeans, with scores of 1684.21 and 823.53, respectively. This metric measures the ratio of between-cluster dispersion to within-cluster dispersion, where higher values indicate better-defined clusters. The significant difference in scores highlights KMeans' ability to form tighter and more distinct clusters, thanks to its use of the full dataset. In contrast, MiniBatchKMeans' reliance on subset optimization compromises the clarity and separation of its clusters, making it less effective when high-quality clustering is required.

Similarly, the Davies-Bouldin index underscores KMeans' superiority in cluster quality, with a lower score of 1.55 compared to MiniBatchKMeans' 3.09. This index evaluates the average similarity between clusters, where lower values reflect more compact and well-separated clusters. KMeans' lower Davies-Bouldin score confirms its ability to produce tighter and more distinct clusters, whereas MiniBatchKMeans' higher score indicates overlapping or less cohesive clusters. While MiniBatchKMeans provides computational efficiency, it comes at the cost of reduced clustering quality, making it less suitable for tasks requiring clear and interpretable cluster boundaries.