Description of Dataset

Link to Dataset Source: https://archive.ics.uci.edu/dataset/352/online+retail

Dataset Name: Online Retail Dataset

Description:

- The Online Retail dataset is a transactional data set which contains all the transactions occurring between 01/12/2010 and 09/12/2011 for a UK-based and registered non-store online retail.
- The company mainly sells unique all-occasion gifts. Many customers of the company are wholesalers.
- Number of Instances: 541909
- Number of Attributes: 6 numerical 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 = '../Online_Retail/Online_Retail.csv'

# Load the data with limited rows for faster processing
dataset = read_csv(filename, encoding='windows-1254')

# Display the first few rows to understand the dataset
print(dataset.head(), "\n")
```

Python

```
  InvoiceNo StockCode                          Description  Quantity  \
0    536365    85123A   WHITE HANGING HEART T-LIGHT HOLDER         6
1    536365     71053                  WHITE METAL LANTERN         6
2    536365    84406B       CREAM CUPID HEARTS COAT HANGER         8
3    536365    84029G  KNITTED UNION FLAG HOT WATER BOTTLE         6
4    536365    84029E       RED WOOLLY HOTTIE WHITE HEART.         6

     InvoiceDate  UnitPrice  CustomerID         Country
0  12/1/10 8:26       2.55     17850.0  United Kingdom
1  12/1/10 8:26       3.39     17850.0  United Kingdom
2  12/1/10 8:26       2.75     17850.0  United Kingdom
3  12/1/10 8:26       3.39     17850.0  United Kingdom
4  12/1/10 8:26       3.39     17850.0  United Kingdom
```

The ‘filename’ variable is set to the relative path of the dataset file. The file format is CSV (Comma-Separated Values), which is commonly used for structured data storage. The ‘read_csv’ function from the pandas library is used to load the data from the specified CSV file into a DataFrame, a two-dimensional tabular data structure. The encoding='windows-1254' argument specifies the character encoding format of the file, which is necessary when dealing with files that may contain special characters or non-ASCII text. Lastly, the ‘print(dataset.head(), "\n")’, which outputs the first few rows (default is 5) of the dataset to give the user a quick preview. This is useful for understanding the structure of the data, identifying columns, and inspecting the data types.

```
# Data preprocessing: adding a TotalPrice column
dataset['TotalPrice'] = dataset['Quantity'] * dataset['UnitPrice']
```

This adds a new column to the dataset called ‘TotalPrice’, which is calculated by multiplying the values in the Quantity column by the values in the ‘UnitPrice’ column. This step creates a new feature that represents the total cost of each transaction. This is essential for understanding the purchasing behavior of customers, as the total price can give more insight into the value of each transaction beyond just the quantity purchased and better for creating clusters.

```
# Aggregating the data by CustomerID to get the total quantity and total price for each customer
customer_data = dataset.groupby('CustomerID').agg({
    'Quantity': 'sum',
    'TotalPrice': 'sum'
}).reset_index()

# Feature scaling
scaler = StandardScaler()
scaled_dataset = scaler.fit_transform(customer_data[['Quantity', 'TotalPrice']])
```

Before applying clustering algorithm K-means, the transactions are grouped by each unique customer and the sum of Quantity and TotalPrice for each customer are calculated. As a result, a new DataFrame (customer_data) is obtained where each row represents a customer, showing how much they have bought in total and the total price they have paid across all their transactions. The features (‘Quantity’ and ‘TotalPrice’) are scaled where feature scaling is a crucial step in clustering because K-means is sensitive to the magnitude of the features. The ‘StandardScaler’ standardizes the data by subtracting the mean and scaling it to unit variance. This ensures that both the ‘Quantity’ and ‘TotalPrice’ columns are on the same scale, preventing one feature from disproportionately influencing the clustering results.

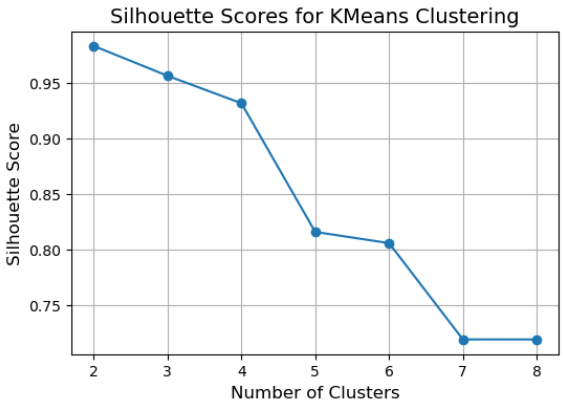
```
# Function to calculate silhouette scores for k-means with clusters between 2 and 8
silhouette_scores = []
range_n_clusters = range(2, 9)

for k in range_n_clusters:
    kmeans = KMeans(n_clusters=k, random_state=42)
    cluster_labels = kmeans.fit_predict(scaled_dataset)
    silhouette_avg = silhouette_score(scaled_dataset, kmeans.labels_)
    silhouette_scores.append(silhouette_avg)

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

```
Silhouette_scores: [0.9830044706552725, 0.9561031477840037, 0.9314275129237506, 0.8155049239902052, 0.8054488993304991, 0.7187161900187632,
0.7186826301815908]
```

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



The plot shows the silhouette scores for K-means clustering with different numbers of clusters, ranging from 2 to 8. The silhouette score, which measures how well-separated the clusters are, decreases steadily as the number of clusters increases. The highest silhouette score, around 0.97, is achieved with 2 clusters, indicating that the clustering is most distinct and well-separated at this level. As the number of clusters increases beyond 2, the silhouette score drops progressively, suggesting that adding more clusters reduces the overall quality of the clustering. This steep decline is especially noticeable after 4 clusters, with scores approaching 0.75 for 7 and 8 clusters, indicating that the clusters are less well-defined. From this, 2 or 3 clusters seem to provide the best separation, while higher numbers of clusters result in poorer performance.

```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)
```

Python

```
The best two clusters are [3 2]
```

This determines the two best values for the number of clusters based on the silhouette scores. The np.argsort(silhouette_scores) function returns the indices that would sort the silhouette_scores array in ascending order. By selecting [-2:], the code extracts the indices of the two highest silhouette scores, corresponding to the best-performing numbers of clusters. Since the indices in np.argsort() start at 0, the code adds 2 to each index to adjust for the fact that the original cluster range started at 2 (i.e., clusters range from 2 to 8). 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 2 and 3 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 = 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)
```

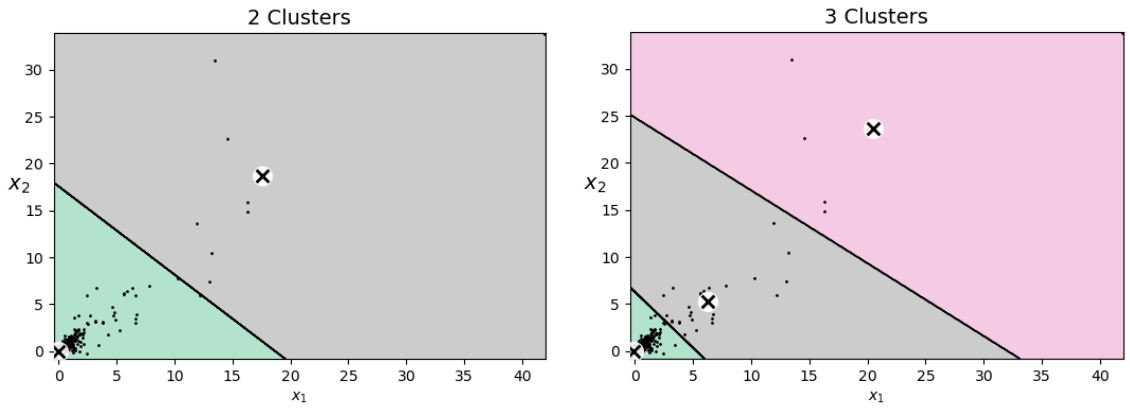
Python

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(scaled_dataset)
    kmeans.cluster_centers_
    plt.figure(figsize=(6, 4))
    plot_decision_boundaries(kmeans, scaled_dataset)
    title = str(i) + ' Clusters'
    plt.title(title, fontsize=10)
    plt.show()
```

Python



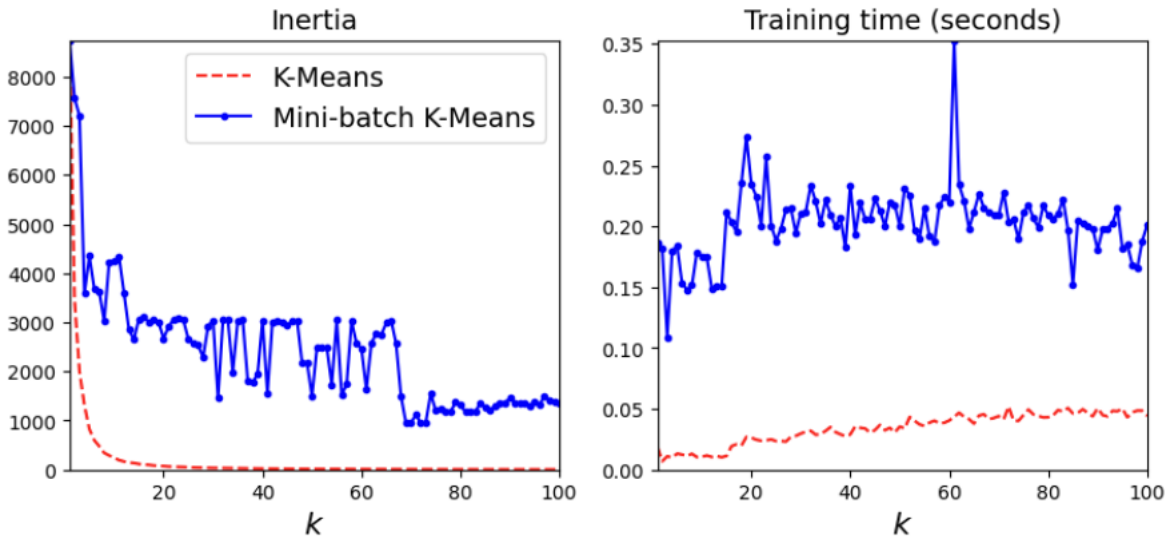
The plot on the left shows the decision boundaries for a K-means clustering model with two clusters while the plot on the right shows with three clusters. Both plots effectively illustrate how the data is divided into their distinct number of clusters.

The plot on the left has two distinct regions shaded gray and green to represent the areas dominated by each cluster. The black contour line marks the boundary that separates the clusters. The data points, plotted as small black dots, are mostly concentrated in the lower-left region of the plot, while a few points extend into the upper-right region. The centroids of the clusters are marked with large black crosses ('x') where one centroid is located near the dense cluster of data points in the green region, while the other is positioned further away in the gray region, indicating that the data points are more spread out in that area.

The plot on the right is represented by different colored regions—green, gray, and pink—separated by black contour lines. The data points are mostly concentrated near the origin, with clusters forming in the green and gray regions. Three centroids, marked with large black crosses, indicate the center of each cluster. One centroid is in the densely populated green region near the origin, another is in the gray region capturing spread along the horizontal axis, and the third is farther away in the pink region, representing more dispersed data points.

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.

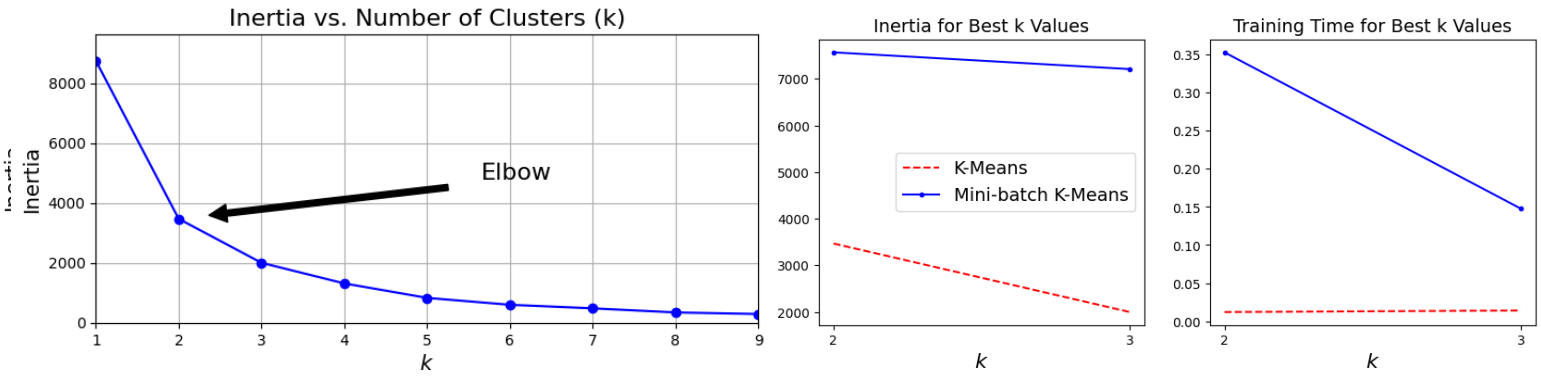
We now have to compare the results of K-Means with the results of Mini-Batch K-Means. Mini-Batch K-Means is a variation of K-Means that reduces computation time by working with small random samples (mini-batches) from the dataset instead of the entire dataset.
The batch size is an important parameter in Mini-Batch K-Means, as it determines how many data points are used in each iteration. A reasonable mini-batch size is chosen (typically much smaller than the dataset size) to balance computational efficiency and clustering performance. In this case, batch_size is set to 100, which is a typical starting point for datasets of moderate size.



However, we can use the result of the Silhouette function used earlier instead of going through all the different numbers of clusters. We found that the best two values for the number of clusters are k = 2 and k = 3. To compare the results of K-Means and Mini-Batch L-Means we can use the following code:

```python
kmeans_per_k = [KMeans(n_clusters=k, random_state=42).fit(X) for k in range(1, 10)]
inertias = [model.inertia_ for model in kmeans_per_k]

plt.figure(figsize=(8, 3.5))
plt.plot(range(1, 10), inertias, "bo-")
plt.xlabel("$k$", fontsize=14)
plt.ylabel("Inertia", fontsize=14)
plt.annotate('Elbow',
             xy=(2, inertias[1]),
             xytext=(0.55, 0.55),
             textcoords='figure fraction',
             fontsize=16,
             arrowprops=dict(facecolor='black', shrink=0.1)
            )
plt.axis([1, 9, 0, max(inertias) * 1.1])
plt.title("Inertia vs. Number of Clusters (k)", fontsize=16)
plt.grid()
plt.show()

best_k_values = [2, 3]

times = np.empty((len(best_k_values), 2))
inertias_best_k = np.empty((len(best_k_values), 2))

for idx, k in enumerate(best_k_values):
    kmeans_ = KMeans(n_clusters=k, random_state=42)
    minibatch_kmeans = MiniBatchKMeans(n_clusters=k, random_state=42)
    times[idx, 0] = timeit(lambda: kmeans_.fit(X), number=1)
    times[idx, 1] = timeit(lambda: minibatch_kmeans.fit(X), number=1)
    inertias_best_k[idx, 0] = kmeans_.inertia_
    inertias_best_k[idx, 1] = minibatch_kmeans.inertia_

plt.figure(figsize=(10, 4))
plt.subplot(121)
plt.plot(best_k_values, inertias_best_k[:, 0], "r--", label="K-Means")
plt.plot(best_k_values, inertias_best_k[:, 1], "b.-", label="Mini-batch K-Means")
plt.xlabel("$k$", fontsize=16)
plt.title("Inertia for Best k Values", fontsize=14)
plt.legend(fontsize=14)
plt.xticks(best_k_values)
plt.subplot(122)
plt.plot(best_k_values, times[:, 0], "r--", label="K-Means")
plt.plot(best_k_values, times[:, 1], "b.-", label="Mini-batch K-Means")
plt.xlabel("$k$", fontsize=16)
plt.title("Training Time for Best k Values", fontsize=14)
plt.xticks(best_k_values)
plt.show()
```

The following is the output of the program:



The first graph shows a plot of inertia against the number of clusters k. As k increases, inertia decreases. The "Elbow" occurs at k=2, where the rate of decrease in inertia slows down. This confirms that 2 clusters

are likely the optimal number of clusters, as adding more clusters beyond this point doesn't significantly reduce inertia.

In the graph Inertia for Best k Values, we notice that for both k=2 and k=3, the inertia values for Mini-Batch K-Means are higher than those for K-Means. This indicates that Mini-Batch K-Means does not cluster the data as tightly as the traditional K-Means algorithm. This is probably because Mini-Batch K-Means is known to produce slightly higher inertia due to its approximation approach.

In the Training Time for Best k values graph, we notice that K-Means takes a relatively constant and very small amount of time, close to zero for k=2 and k=3. On the other hand, Mini-Batch K-Means has a significantly longer training time, especially for k=2, though it improves slightly for k=3. This is somewhat counterintuitive, as Mini-Batch K-Means is typically expected to be faster, especially for large datasets.

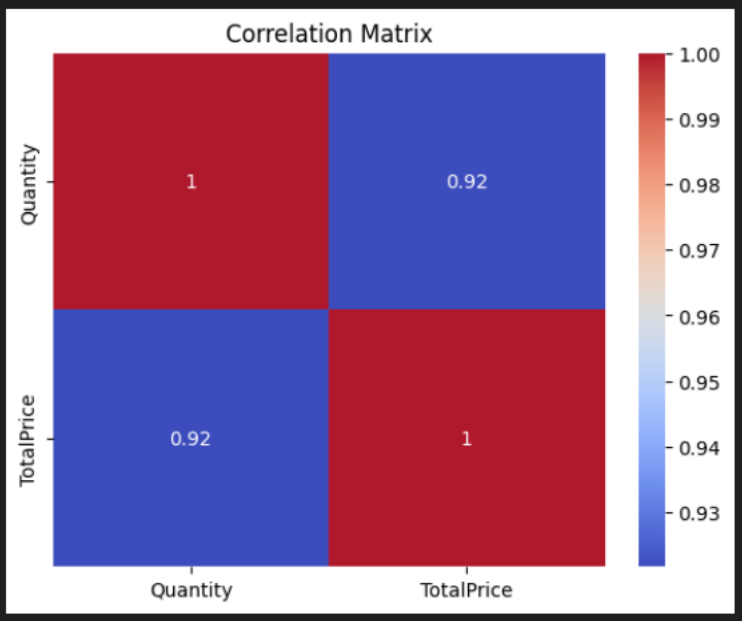
In conclusion, we decided not to change thebatch size of 100 for Mini-Batch K-Means because after experimenting with larger batch sizes, we noticed that they resulted in increased training times. Although larger batches theoretically provide more accurate centroid updates by incorporating more data points at once, they can also lead to diminishing returns in computational efficiency, like in our case.

c) 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)

```python
from sklearn.metrics import calinski_harabasz_score, davies_bouldin_score
import seaborn as sns


# Assuming 'scaled_dataset' contains your scaled features
X = scaled_dataset  # Ensure this contains Quantity and TotalPrice

# Perform clustering
kmeans = KMeans(n_clusters=2, random_state=42).fit(X)
minibatch_kmeans = MiniBatchKMeans(n_clusters=2, random_state=42).fit(X)

# Evaluate clustering
ch_index_kmeans = calinski_harabasz_score(X, kmeans.labels_)
db_index_kmeans = davies_bouldin_score(X, kmeans.labels_)

ch_index_minibatch = calinski_harabasz_score(X, minibatch_kmeans.labels_)
db_index_minibatch = davies_bouldin_score(X, minibatch_kmeans.labels_)

# Print results
print("Calinski-Harabasz Index for K-Means:", ch_index_kmeans)
print("Davies-Bouldin Index for K-Means:", db_index_kmeans)
print("Calinski-Harabasz Index for Mini-Batch K-Means:", ch_index_minibatch)
print("Davies-Bouldin Index for Mini-Batch K-Means:", db_index_minibatch)

# Correlation matrix
correlation_matrix = pd.DataFrame(X, columns=['Quantity', 'TotalPrice']).corr()
print("Correlation Matrix:\n", correlation_matrix)

# Visualize the correlation matrix
sns.heatmap(correlation_matrix, annot=True, cmap='coolwarm')
plt.title('Correlation Matrix')
plt.show()
```

The output of the code is the following:

```
Calinski-Harabasz Index for K-Means: 6649.465459498878
Davies-Bouldin Index for K-Means: 0.41365781175655486
Calinski-Harabasz Index for Mini-Batch K-Means: 927.5931645544644
Davies-Bouldin Index for Mini-Batch K-Means: 0.9781904766072186
Correlation Matrix:
             Quantity  TotalPrice
Quantity    1.000000    0.921649
TotalPrice  0.921649    1.000000
```



The CH index for K-Means is significantly higher than that for Mini-Batch K-Means, indicating that K-Means forms better-defined clusters with greater separation between them. Mini-Batch K-Means has a much lower CH index, meaning that its clusters are not as distinct. This is expected, as Mini-Batch K-Means is an approximation algorithm, which trades some clustering quality generally for faster execution.

The DB Index for K-Means (0.414) is considerably lower than that for Mini-Batch K-Means (0.978). This reinforces the conclusion that K-Means produce better clusters with a more distinct separation between them. The higher DB index for Mini-Batch K-Means indicates that its clusters are not as well-separated, with greater overlap between neighboring clusters.

Focusing on the Correlation Matrix, we used Q**uantity** and **TotalPrice** because we are evaluating how well the clustering model has captured the structure of the data based on these two features. The correlation matrix is showing how these features relate to each other, potentially revealing patterns that could inform us understanding of how clusters are formed. Of course we have a relation of 1 (the maximum) when Total Price and Quantity are checked with themselves, but also we can notice that there is a very strong positive correlation (0.9216) between Quantity and Total Price. This suggests that as the quantity of items increases, the total price tends to increase proportionally.
This strong correlation indicates that these two features are closely related, which could influence the clustering results. Since both features are highly correlated, any clustering algorithm will likely group them in similar ways. Also, we can maybe find patterns based on this big correlation. For example, high correlation may indicate that customers who buy more items tend to spend significantly more, while low correlations could suggest more varied purchasing habits, with some customers buying large quantities of lower-priced items.