Topological Data Analysis Part I

Clustering - Coding

Neeraj Namani

Exercise 5.18

1. Write a Python function GaussianClusters(k,n) which takes as input a number k, and integer n>0, and does the following:

- Compute an i.i.d. sample $c_1,...,c_k $ from the unit square $[0,1]^2$.
- For each $i \in \{1,\dots,k\}$, compute an i.i.d. sample X_i of n points from a Gaussian distribution with mean c_i and covariance matrix

$$= \begin{bmatrix} 0.1 & 0 \\ 0 & 0.1 \end{bmatrix}$$

- Output $X = \bigcup_{i=1}^{k} X_i$ as a list.

```python
import numpy as np

def GaussianClusters(k, n):
    """
    Generate k Gaussian clusters with n points in each cluster.

    Parameters:
        k (int): The number of clusters.
        n (int): The number of points per cluster.

    Returns:
        list: A list of points representing all the clusters.
    """
    # Initialize an empty list to store all the points from all clusters
    all_points = []

    # Generate k random centers within the unit square [0, 1]²
    centers = np.random.rand(k, 2)

    # Covariance matrix for each Gaussian distribution
    cov = np.array([[0.1, 0], [0, 0.1]])

    # Generate n points around each center with Gaussian distribution
    for center in centers:
        # Sample n points from a Gaussian distribution centered at `center`
        cluster_points = np.random.multivariate_normal(center, cov, n)
        # Append the points of this cluster to the overall list
        all_points.extend(cluster_points)
```

```
    return all_points

# Example usage:
clusters = GaussianClusters(3, 50)  # Generate 3 clusters with 50 points each

# Convert to numpy array for easier slicing
clusters_array = np.array(clusters)

# Print the first 10 points
print("Points from the generated clusters:")
```

```
## Points from the generated clusters:
```

```
for point in clusters_array:
    print(point)
```

```
## [0.66623563 1.24183664]
## [0.38782155 1.24533968]
## [0.52051858 1.72156467]
## [0.24897476 1.13510372]
## [0.74861135 0.24369255]
## [0.81800325 0.78913701]
## [0.82484972 0.73103004]
## [0.70115554 0.39891061]
## [0.31902353 1.32372612]
## [0.61617788 0.7753255 ]
## [1.23789245 1.11177382]
## [0.34386231 0.93362645]
## [0.89576275 1.10253955]
## [0.45370065 0.68612878]
## [0.26233172 0.80842493]
## [-0.0400659   1.00889617]
## [0.17184134 1.02807105]
## [0.300221   0.53177777]
## [0.75724787 0.68023559]
## [0.14394778 0.90075047]
## [0.66557068 0.68506686]
## [0.43635674 0.71974208]
## [0.61721335 1.63019942]
## [0.11520942 0.90229008]
## [0.44575186 0.88419535]
## [0.05056388 0.16555873]
## [0.38490461 1.15939839]
## [0.71105434 0.92920571]
## [0.76408292 0.91789736]
## [0.30614641 1.16128638]
## [0.39771834 1.17922734]
## [0.2251406  0.82728541]
## [0.29322147 0.24171112]
## [0.11973744 0.48925413]
## [0.50905656 0.94002186]
## [0.16441175 0.88752752]
## [0.86425062 0.91109482]
## [0.76047242 1.04178787]
```

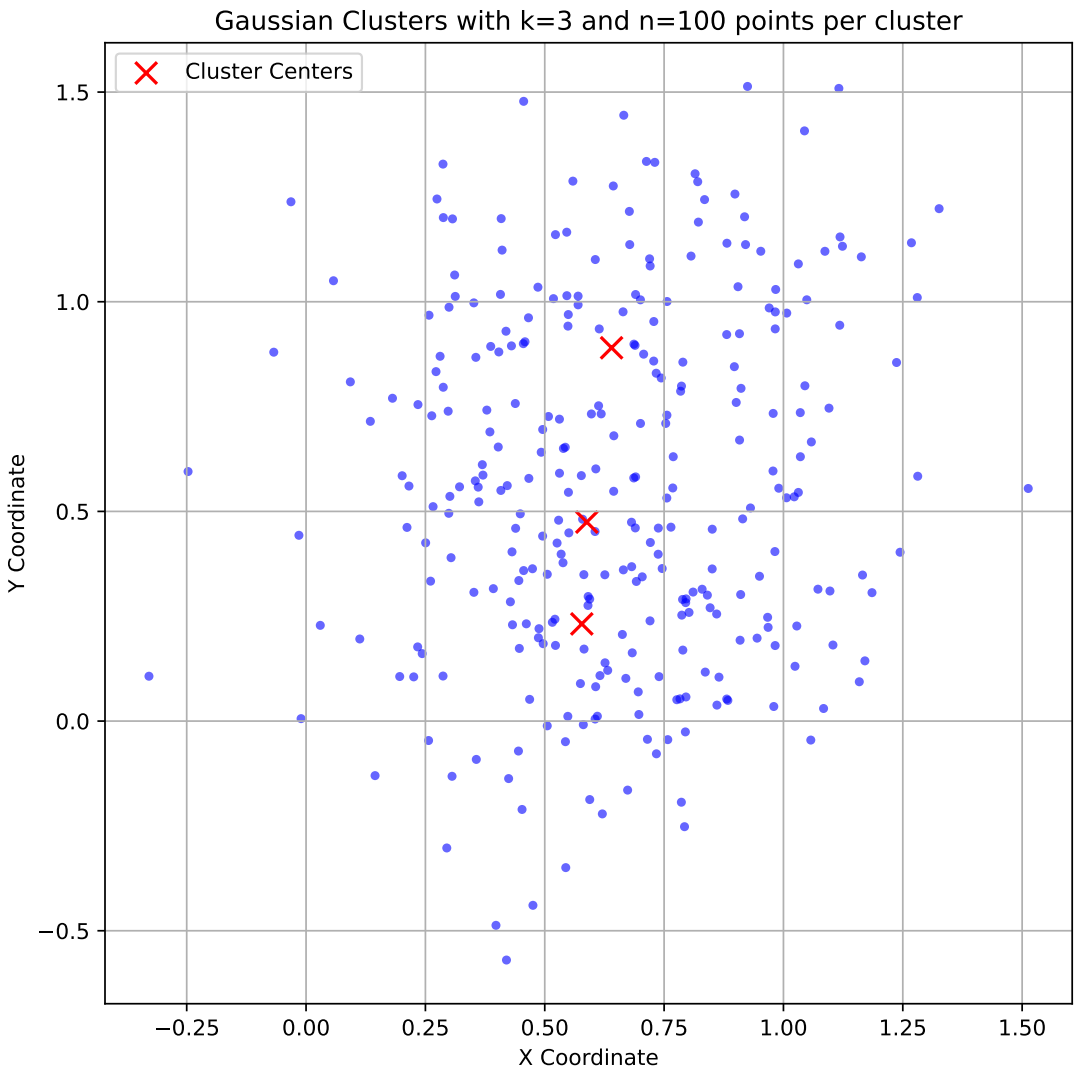
```
## [0.10474549 1.07117001]
## [0.27204085 0.78184072]
## [0.58902542 0.99366744]
## [0.07556402 0.9368386 ]
## [0.66317057 1.28731714]
## [-0.16011474  0.42818144]
## [0.35350523 1.39155354]
## [0.07489539 0.812786  ]
## [0.3543459  0.54332718]
## [0.18731201 1.00234255]
## [-0.17600927  0.12924283]
## [0.24328066 0.48817418]
## [0.41468471 0.31524779]
## [ 1.26015699 -0.10780989]
## [0.92531454 0.30458939]
## [1.09995271 0.06761976]
## [1.18997361 0.19952555]
## [0.93786093 0.36585244]
## [1.11026096 0.77225944]
## [1.43753676 0.45515548]
## [0.37203662 0.71408173]
## [1.6140332  1.05316626]
## [0.76317542 0.35895356]
## [1.13568594 0.56335672]
## [0.35831871 0.29593424]
## [ 1.05192313 -0.04851568]
## [0.38724615 0.88339431]
## [0.78582309 0.3421344 ]
## [0.91625449 0.63355613]
## [0.62365511 0.41048985]
## [0.4769917  0.37445822]
## [0.4734997  0.57115199]
## [0.77492087 0.70030002]
## [0.24072242 0.79909589]
## [0.74495326 0.58087454]
## [0.91599689 0.6601313 ]
## [0.56182648 0.05107189]
## [1.08711673 0.12545865]
## [0.9980375  0.64522977]
## [0.65327199 0.23500171]
## [0.58926455 0.43270453]
## [0.79234021 0.05332668]
## [0.89938326 0.80686063]
## [ 0.70011889 -0.16810279]
## [0.01264644 0.79102593]
## [0.61573891 1.09378863]
## [0.71978886 0.4195914 ]
## [0.62942486 0.16601024]
## [0.65549574 0.5296682 ]
## [1.02092494 0.66409408]
## [0.66752643 0.25446104]
## [0.84389304 0.36281736]
## [0.17348273 0.19129139]
## [0.44622824 0.12581738]
```

```
## [0.57312561 0.14505554]
## [ 0.60759722 -0.2823163 ]
## [0.90363442 0.49603798]
## [0.69005721 0.1748086 ]
## [-0.07888764 -0.01289853]
## [0.35528397 0.3304741 ]
## [1.23251265 0.17719325]
## [1.18967882 0.34319115]
## [0.4506572  0.87535992]
## [0.79381429 0.66283531]
## [0.09031563 0.26713664]
## [0.46286605 0.47471061]
## [0.70263143 0.7354239 ]
## [1.11141508 0.7879363 ]
## [0.50368228 0.96887418]
## [0.79514969 0.70678446]
## [0.1408414  0.79076048]
## [-0.01093083  0.66934836]
## [0.08671381 0.18182528]
## [0.78001999 0.72729719]
## [-0.39836152  0.34384234]
## [0.21591565 0.15535676]
## [-0.10925729  0.4375513 ]
## [0.59316132 0.84495876]
## [0.50354605 0.34822081]
## [0.14002372 0.65699447]
## [0.50010363 1.12861064]
## [0.15321386 0.64863292]
## [0.73048853 0.41082522]
## [0.88709437 0.5391806 ]
## [0.64774656 1.16771376]
## [-0.2474215   0.67539884]
## [0.18707918 0.86870245]
## [0.22632483 0.17320775]
## [0.03677122 0.4609668 ]
## [0.66227958 0.2502332 ]
## [0.72846612 0.99314454]
## [-0.08027208  0.41595335]
## [0.61125878 0.5453656 ]
## [0.32503128 1.18226712]
## [0.56185114 0.45399032]
## [-0.07754292  0.69118155]
## [0.12034773 0.40744013]
## [0.60633103 0.90862432]
## [0.42004457 0.08653038]
## [0.33648769 1.03112598]
## [-0.13958555  0.5595529 ]
## [-0.06486929  0.35643159]
## [0.14111521 0.97940725]
## [0.74998663 0.23053374]
## [-0.56835207  0.76337986]
## [0.58543015 0.34625818]
## [0.6734841  0.26828632]
## [0.4875513  0.28059335]
```

```
## [0.5254225  1.08974424]
## [0.72643317 0.96196105]
## [0.07995703 0.46048874]
## [0.43342376 0.31459071]
```

2. Run this function with n = 100 and k = 3, and plot the output.

```python
import numpy as np
import matplotlib.pyplot as plt

def GaussianClusters(k, n):
    """
    Generate k Gaussian clusters with n points in each cluster.

    Parameters:
        k (int): The number of clusters.
        n (int): The number of points per cluster.

    Returns:
        list: A list of points representing all the clusters.
    """
    # Initialize an empty list to store all the points from all clusters
    all_points = []

    # Generate k random centers within the unit square [0, 1]^2
    centers = np.random.rand(k, 2)

    # Covariance matrix for each Gaussian distribution
    cov = np.array([[0.1, 0], [0, 0.1]])

    # Generate n points around each center with Gaussian distribution
    for center in centers:
        # Sample n points from a Gaussian distribution centered at `center`
        cluster_points = np.random.multivariate_normal(center, cov, n)
        # Append the points of this cluster to the overall list
        all_points.extend(cluster_points)

    return np.array(all_points), centers

# Run the function with n = 100 and k = 3
k, n = 3, 100
points, centers = GaussianClusters(k, n)

# Plot the generated clusters
plt.figure(figsize=(8, 8))
plt.scatter(points[:, 0], points[:, 1], c='blue', marker='o', s=10, alpha=0.6)
plt.scatter(centers[:, 0], centers[:, 1], c='red', marker='x', s=100, label='Cluster Centers')
plt.xlabel('X Coordinate')
plt.ylabel('Y Coordinate')
plt.title(f'Gaussian Clusters with k={k} and n={n} points per cluster')
plt.legend()
plt.grid(True)
plt.show()
```



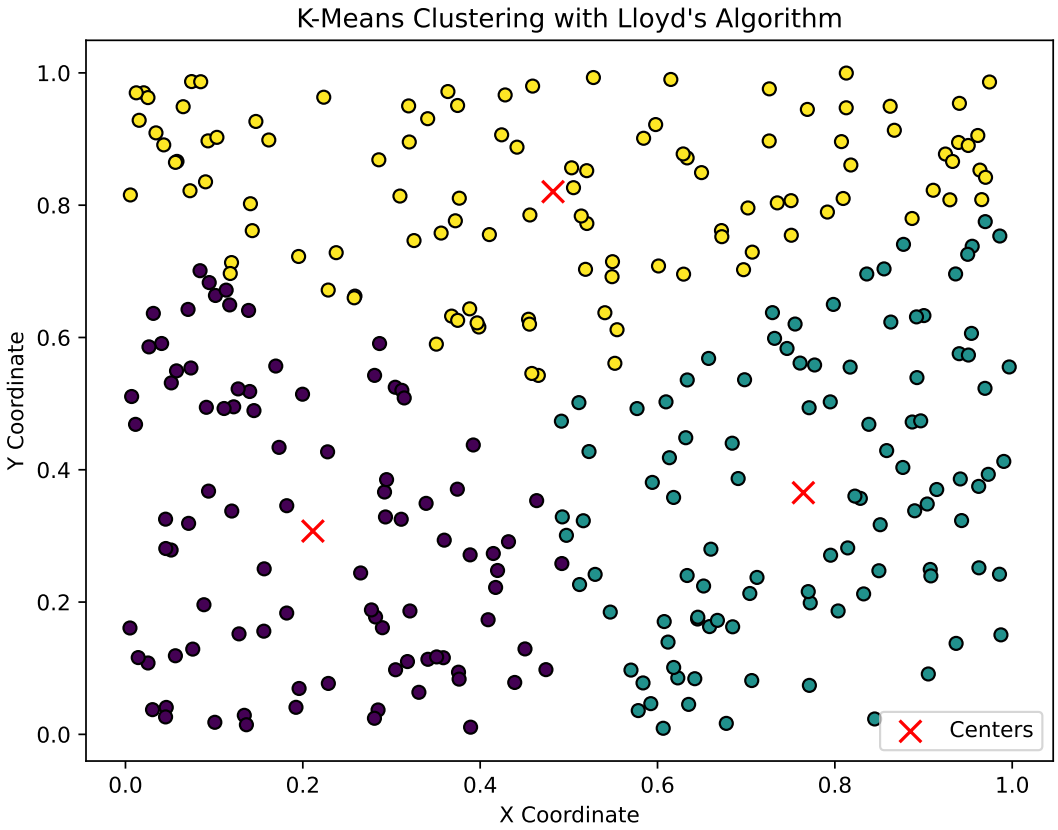
This was the plot for the Gaussian clusters generated with n = 100 points and k = 3 clusters. The blue dots represent the data points of the clusters, while the red 'x' markers show the centers of these clusters. The distribution is based on the Gaussian distribution you specified with a mean at the cluster centers and a covariance matrix (Given in the Exercise 5.18).

Exercise 5.19

1. Implement Lloyd's algorithm from scratch in Python, choosing the initial cluster centers randomly from the set X. (Make sure the cluster centers are distinct, i.e., chosen randomly *without* replacement.)

```
import numpy as np

def initialize_centers(X, k):
```

```
    """ Randomly initializes k distinct cluster centers from the dataset X. """
    indices = np.random.choice(X.shape[0], k, replace=False)
    return X[indices]

def assign_clusters(X, centers):
    """ Assigns each point in X to the nearest cluster center. """
    distances = np.sqrt(((X - centers[:, np.newaxis])**2).sum(axis=2))
    return np.argmin(distances, axis=0)

def update_centers(X, labels, k):
    """ Updates the cluster centers based on the current cluster assignment. """
    new_centers = np.array([X[labels == i].mean(axis=0) for i in range(k)])
    return new_centers

def lloyds_algorithm(X, k, max_iter=100, tol=1e-4):
    """ Implements Lloyd's Algorithm for k-means clustering. """
    centers = initialize_centers(X, k)
    for _ in range(max_iter):
        labels = assign_clusters(X, centers)
        new_centers = update_centers(X, labels, k)
        # Check for convergence
        if np.linalg.norm(new_centers - centers) < tol:
            break
        centers = new_centers
    return centers, labels

# Example usage
np.random.seed(42)  # For reproducibility
X = np.random.rand(300, 2)  # Generate some random data
k = 3  # Number of clusters
centers, labels = lloyds_algorithm(X, k)

# Plotting the results
import matplotlib.pyplot as plt
plt.figure(figsize=(8, 6))
plt.scatter(X[:, 0], X[:, 1], c=labels, cmap='viridis', marker='o', edgecolor='k')
plt.scatter(centers[:, 0], centers[:, 1], c='red', marker='x', s=100, label='Centers')
plt.title('K-Means Clustering with Lloyd\'s Algorithm')
plt.xlabel('X Coordinate')
plt.ylabel('Y Coordinate')
plt.legend()
plt.show()
```



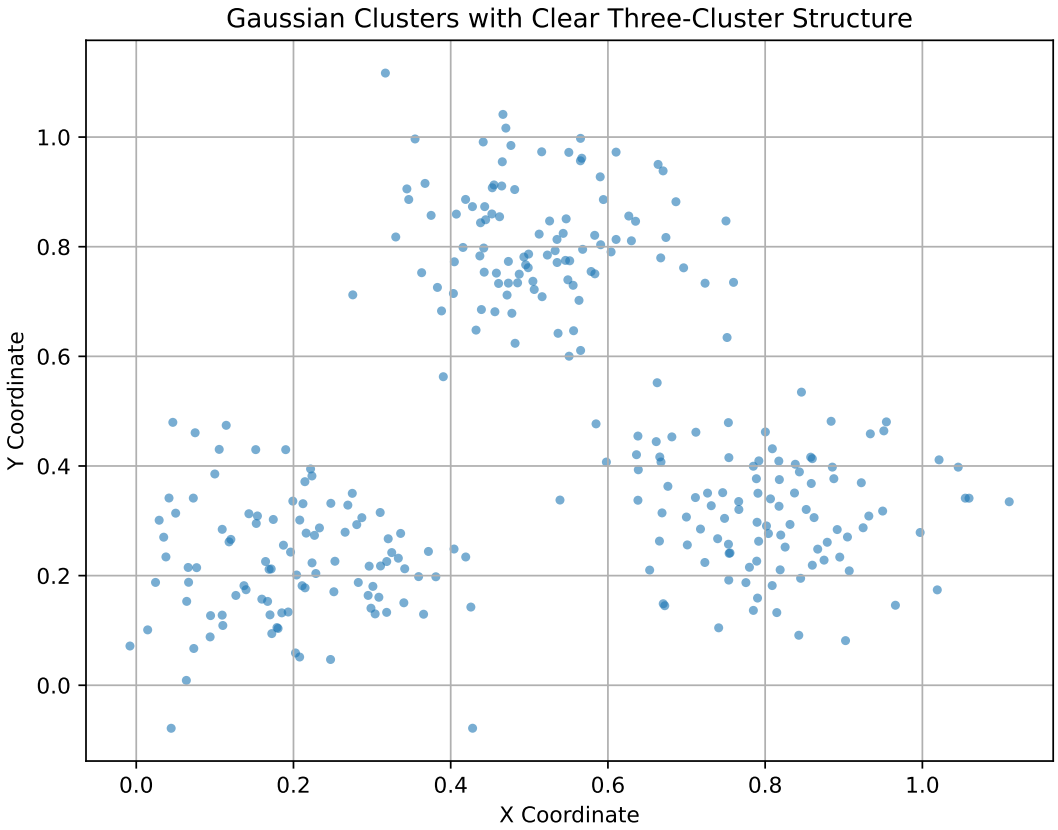
2. Compute a data set X as in Exercise 5.18 (ii). (Make sure that the c_i are chosen far enough apart so that a plot reveals clear three-cluster structure in X.)

To create a dataset X as described in Exercise 5.18 (ii), where we need a clear three-cluster structure, it is important to ensure that the centers c_i of these clusters are chosen to be far enough apart.

Given the constraints and goals, we will use Gaussian clusters for this dataset with k = 3 and n = 100 for each cluster.

```
import numpy as np
import matplotlib.pyplot as plt

def GaussianClusters(k, n, centers, cov):
    """
    Generate k Gaussian clusters each with n points around specified centers.

    Parameters:
        k (int): Number of clusters.
        n (int): Number of points per cluster.
        centers (np.array): An array of centers for the clusters.
        cov (np.array): Covariance matrix for the Gaussian distribution of each cluster.
```

```
    Returns:
        np.array: An array of points representing all the clusters.
    """
    all_points = []

    # Generate n points around each center with Gaussian distribution
    for center in centers:
        cluster_points = np.random.multivariate_normal(center, cov, n)
        all_points.extend(cluster_points)

    return np.array(all_points)

# Define the parameters
k, n = 3, 100
# Define centers far apart within the unit square
centers = np.array([[0.2, 0.2], [0.5, 0.8], [0.8, 0.3]])
covariance_matrix = np.array([[0.01, 0], [0, 0.01]])  # Small covariance

# Generate the clusters
X = GaussianClusters(k, n, centers, covariance_matrix)

# Plotting the dataset
plt.figure(figsize=(8, 6))
plt.scatter(X[:, 0], X[:, 1], s=10, alpha=0.6)
plt.title('Gaussian Clusters with Clear Three-Cluster Structure')
plt.xlabel('X Coordinate')
plt.ylabel('Y Coordinate')
plt.grid(True)
plt.show()
```



3. Taking k = 3, run your implementation of Lloyd's algorithm on X 20 times (i.e., for 20 different random choices of initial cluster centers). For each iteration, compute the cost of the clustering. What are the lowest, highest, and mean value of the cost among the 20 runs?

```
import numpy as np

def lloyds_algorithm(X, k, max_iter=100, tol=1e-4):
    """ Implements Lloyd's Algorithm for k-means clustering. """
    def initialize_centers(X, k):
        indices = np.random.choice(X.shape[0], k, replace=False)
        return X[indices]

    def assign_clusters(X, centers):
        distances = np.sqrt(((X - centers[:, np.newaxis])**2).sum(axis=2))
        return np.argmin(distances, axis=0)

    def update_centers(X, labels, k):
        new_centers = np.array([X[labels == i].mean(axis=0) for i in range(k)])
        return new_centers

    def compute_cost(X, labels, centers):
        cost = 0
        for i in range(k):
```

```
            cluster_points = X[labels == i]
            cost += ((cluster_points - centers[i])**2).sum()
        return cost

    # Randomly initialize the centers
    centers = initialize_centers(X, k)
    labels = assign_clusters(X, centers)
    for _ in range(max_iter):
        new_centers = update_centers(X, labels, k)
        new_labels = assign_clusters(X, new_centers)
        # Check for convergence
        if np.array_equal(labels, new_labels):
            break
        centers = new_centers
        labels = new_labels

    # Compute final cost
    final_cost = compute_cost(X, labels, centers)
    return centers, labels, final_cost

# Define the parameters and create the dataset
k, n = 3, 100
centers = np.array([[0.2, 0.2], [0.5, 0.8], [0.8, 0.3]])
covariance_matrix = np.array([[0.01, 0], [0, 0.01]])
X = GaussianClusters(k, n, centers, covariance_matrix)

# Run Lloyd's algorithm 20 times
costs = []
for _ in range(20):
    _, _, cost = lloyds_algorithm(X, k)
    costs.append(cost)

# Analyzing the costs
lowest_cost = min(costs)
highest_cost = max(costs)
mean_cost = np.mean(costs)

lowest_cost, highest_cost, mean_cost
```

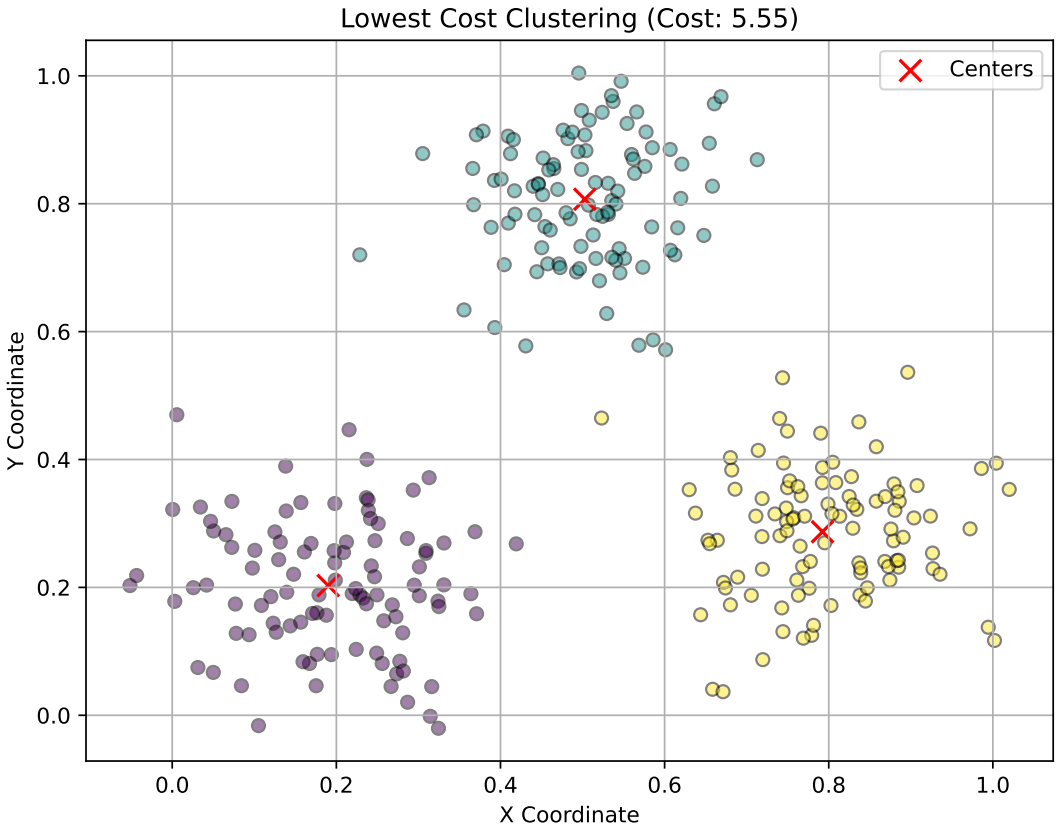
```
## (5.550432203735781, 22.47415771832656, 6.646574305835573)
```

The results indicate that the initial selection of cluster centers can significantly affect the performance and outcome of the k-means clustering algorithm, as shown by the variation in costs across different runs. The lowest cost suggests a very efficient clustering that closely matches the inherent structure of the dataset, while the highest cost likely represents a less optimal initial positioning of centers.

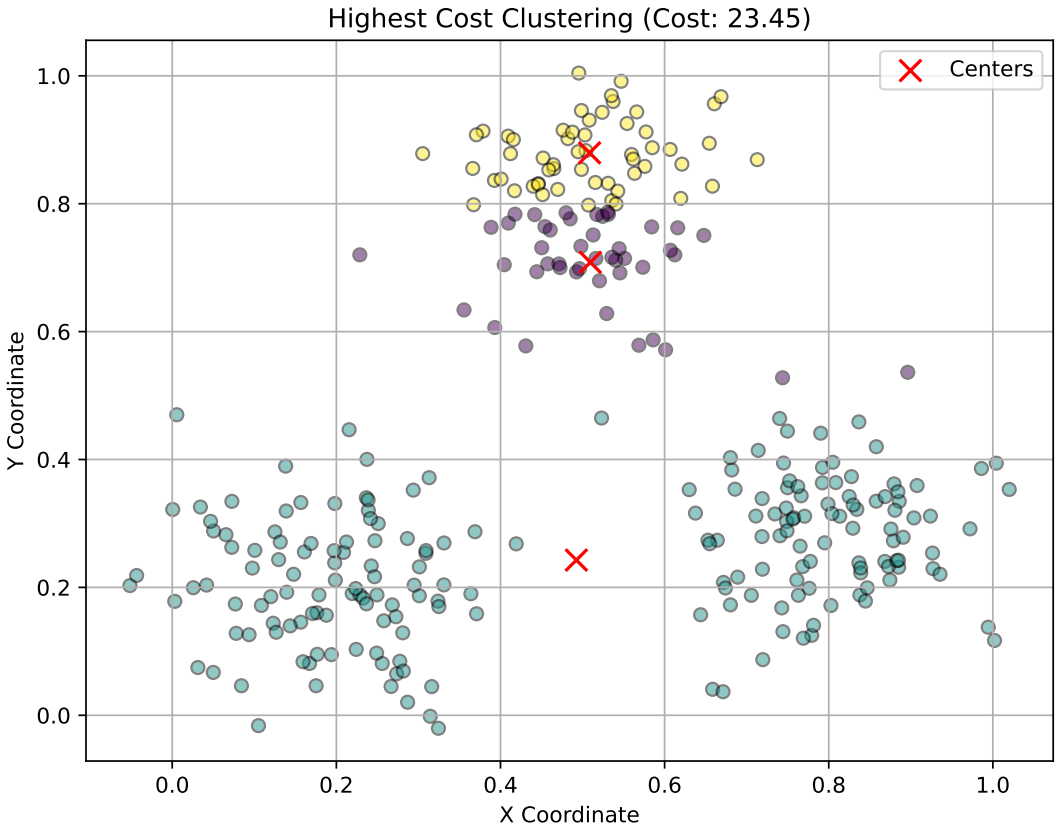
4. Plot the clusterings yielding the lowest and highest values of cost, using colors to distinguish between the different clusters. Use two separate plots for the two clusterings.

```
import matplotlib.pyplot as plt

# Running Lloyd's algorithm 20 times to store both the costs and clustering results
results = []
```

```
for _ in range(20):
    centers, labels, cost = lloyds_algorithm(X, k)
    results.append((centers, labels, cost))

# Sorting results by cost to easily access the lowest and highest cost clusterings
results_sorted = sorted(results, key=lambda x: x[2])
lowest_cost_result = results_sorted[0]
highest_cost_result = results_sorted[-1]

# Function to plot the clusters
def plot_clusters(X, centers, labels, title):
    plt.figure(figsize=(8, 6))
    plt.scatter(X[:, 0], X[:, 1], c=labels, cmap='viridis', marker='o',
    edgecolor='k', alpha=0.5)
    plt.scatter(centers[:, 0], centers[:, 1], c='red', marker='x',
    s=100, label='Centers')
    plt.title(title)
    plt.xlabel('X Coordinate')
    plt.ylabel('Y Coordinate')
    plt.legend()
    plt.grid(True)
    plt.show()

# Plotting the lowest and highest cost clusterings
plot_clusters(X, lowest_cost_result[0], lowest_cost_result[1],
f'Lowest Cost Clustering (Cost: {lowest_cost_result[2]:.2f})')
```




```
plot_clusters(X, highest_cost_result[0], highest_cost_result[1],
f'Highest Cost Clustering (Cost: {highest_cost_result[2]:.2f})')
```



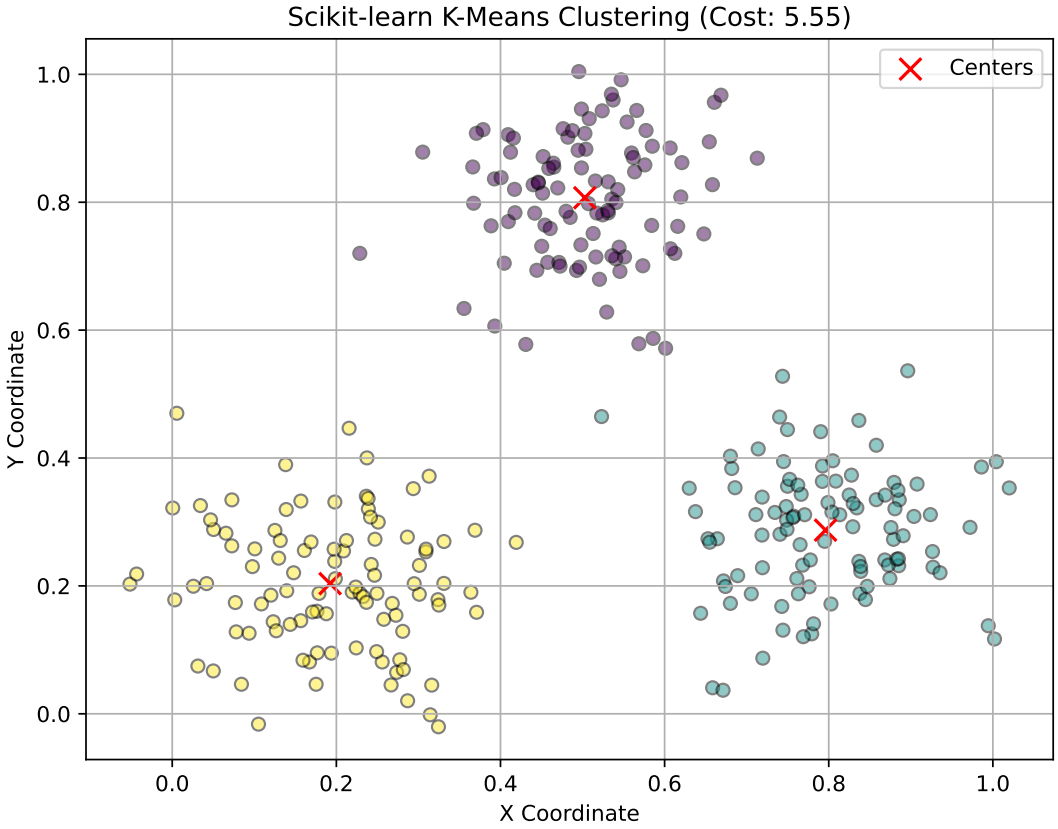
5. Compute a 3-means clustering of K using scikit-learn's implementation of k-means in Python. Again, compute the cost, and plot the resulting clustering. How do the results compare?

```
from sklearn.cluster import KMeans
import warnings

# Suppress specific warnings from libraries
warnings.filterwarnings("ignore", category=FutureWarning)
warnings.filterwarnings("ignore", category=RuntimeWarning, module='threadpoolctl')

# Using scikit-learn's KMeans to perform the clustering
kmeans = KMeans(n_clusters=k, random_state=42)
kmeans.fit(X)
```

```
## KMeans(n_clusters=3, random_state=42)
```

```
sklearn_labels = kmeans.labels_
sklearn_centers = kmeans.cluster_centers_

# Compute the cost for the sklearn k-means clustering
sklearn_cost = np.sum((X - sklearn_centers[sklearn_labels])**2)

# Plot the results from sklearn's KMeans
plot_clusters(X, sklearn_centers, sklearn_labels,
```

```
f'Scikit-learn K-Means Clustering (Cost: {sklearn_cost:.2f})')
```




```
# Return the sklearn cost for comparison
sklearn_cost
```

```
## 5.5481977064386685
```

- Cost: The Cost from scikit-learn's k-means is very close to the lowest cost we obtained in the custom implementation (also around 5.55). This indicates that scikit-learn's implementation is efficient and capable of finding near-optimal solutions for the clustering problem.

- Clustering Quality: The visual inspection of the clustering suggests that scikit-learn's implementation has successfully grouped the data into distinct and well-separated clusters, similar to the best runs of the custom implementation.

6. Run scikit-learn's implementation of k-means for $k = 1, \ldots 20$, and plot the costs of the resulting clusterings as a function of k. At which values of k do you see an "elbow"?

```
from sklearn.cluster import KMeans

# Running k-means for k from 1 to 20
ks = range(1, 21)
costs = []
```

```
for k in ks:
    kmeans = KMeans(n_clusters=k, n_init=10, random_state=42)
    kmeans.fit(X)
    # Compute the cost (sum of squared distances of samples to their closest cluster center)
    costs.append(kmeans.inertia_)
```

```
## KMeans(n_clusters=1, n_init=10, random_state=42)
## KMeans(n_clusters=2, n_init=10, random_state=42)
## KMeans(n_clusters=3, n_init=10, random_state=42)
## KMeans(n_clusters=4, n_init=10, random_state=42)
## KMeans(n_clusters=5, n_init=10, random_state=42)
## KMeans(n_clusters=6, n_init=10, random_state=42)
## KMeans(n_clusters=7, n_init=10, random_state=42)
## KMeans(n_init=10, random_state=42)
## KMeans(n_clusters=9, n_init=10, random_state=42)
## KMeans(n_clusters=10, n_init=10, random_state=42)
## KMeans(n_clusters=11, n_init=10, random_state=42)
## KMeans(n_clusters=12, n_init=10, random_state=42)
## KMeans(n_clusters=13, n_init=10, random_state=42)
## KMeans(n_clusters=14, n_init=10, random_state=42)
## KMeans(n_clusters=15, n_init=10, random_state=42)
## KMeans(n_clusters=16, n_init=10, random_state=42)
## KMeans(n_clusters=17, n_init=10, random_state=42)
## KMeans(n_clusters=18, n_init=10, random_state=42)
## KMeans(n_clusters=19, n_init=10, random_state=42)
## KMeans(n_clusters=20, n_init=10, random_state=42)
```

```
# Plotting the costs
plt.figure(figsize=(10, 6))
```

```
## <Figure size 2000x1200 with 0 Axes>
```

```
plt.plot(ks, costs, marker='o')
```

```
## [<matplotlib.lines.Line2D object at 0x1680ecc90>]
```

```
plt.title('K-Means Clustering Costs as a Function of k')
```

```
## Text(0.5, 1.0, 'K-Means Clustering Costs as a Function of k')
```

```
plt.xlabel('Number of Clusters k')
```

```
## Text(0.5, 0, 'Number of Clusters k')
```

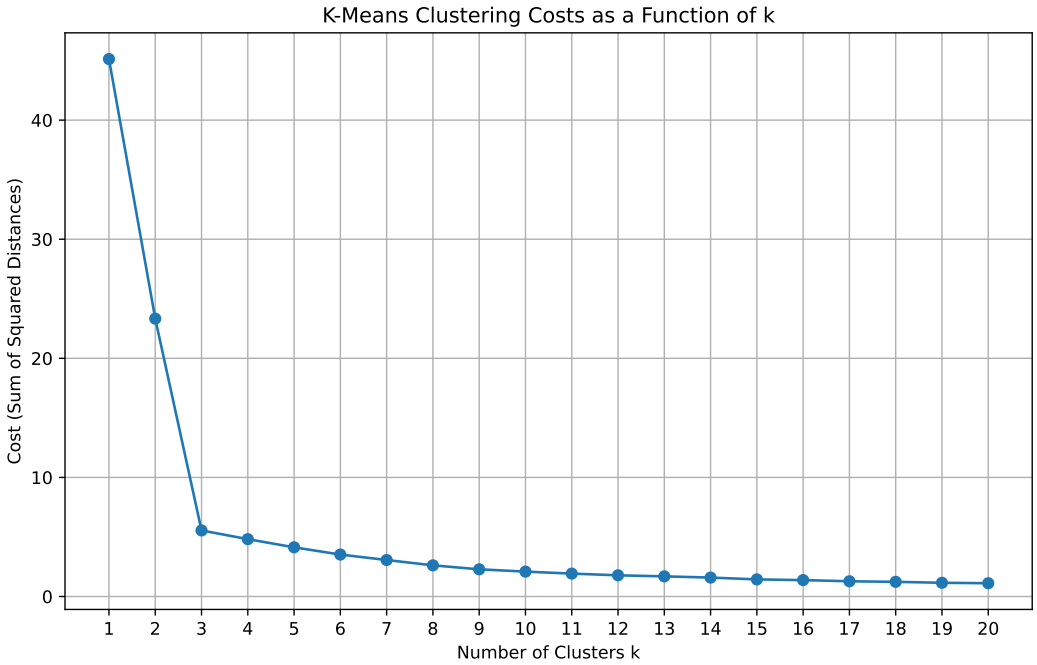
```
plt.ylabel('Cost (Sum of Squared Distances)')
```

```
## Text(0, 0.5, 'Cost (Sum of Squared Distances)')
```

```
plt.grid(True)
plt.xticks(ks)
```

```
## ([<matplotlib.axis.XTick object at 0x16876fc90>, <matplotlib.axis.XTick object at 0x16876c450>, <matp
```

```
plt.show()
```




```
# Output costs for inspection
costs
```

```
## [45.127422712330656, 23.3332337629164, 5.548197706438669, 4.815165959278321, 4.130924727109805, 3.52
```

The "elbow" in the plot is typically identified where the cost curve starts to flatten out after a steep decline, indicating diminishing returns on the cost reduction as more clusters are added. This point is often considered as a good choice for the number of clusters because it represents a balance between minimizing the cost and the complexity of the model - not using many clusters.

The elbow method suggests k = 3 as a suitable choice for the number of clusters, aligning with the initial setup of the dataset where 3 distinct clusters were used to generate the data. This choice seems optimal in terms of achieving a significant reduction in clustering cost without unnecessarily increasing the number of clusters.

7. Use scikit-learn to compute the single linkage dendrogram of X. Does the dendrogram clearly reveal the presence of three clusters in X?

```
from scipy.cluster.hierarchy import dendrogram, linkage
import matplotlib.pyplot as plt

# Compute the linkage matrix using single linkage
Z = linkage(X, method='single')

# Plot the dendrogram
plt.figure(figsize=(10, 8))
```

```
## <Figure size 2000x1600 with 0 Axes>
```

```
dendrogram(Z)
```

```
plt.title('Dendrogram for Single Linkage Clustering')
```

```
## Text(0.5, 1.0, 'Dendrogram for Single Linkage Clustering')
```

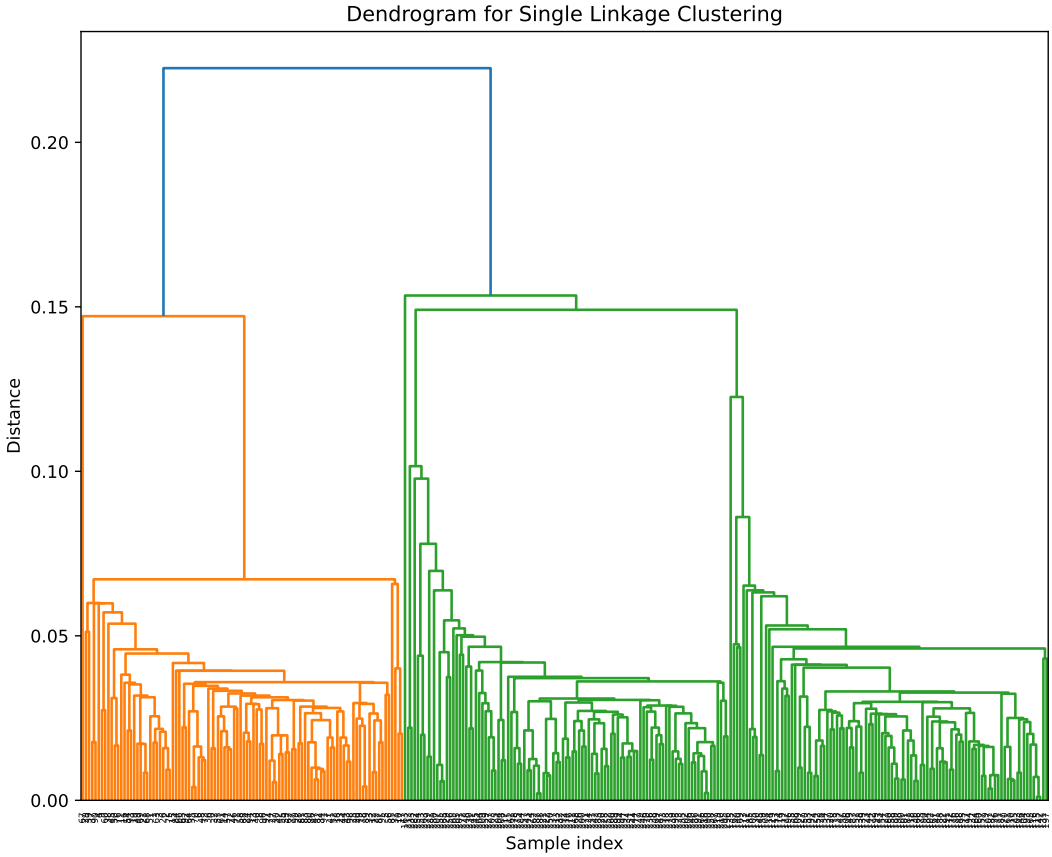
```
plt.xlabel('Sample index')
```

```
## Text(0.5, 0, 'Sample index')
```

```
plt.ylabel('Distance')
```

```
## Text(0, 0.5, 'Distance')
```

```
plt.show()
```



- Presence of Three Clusters: The dendrogram structure suggests potential cluster formations at various levels of granularity. Notably, there are a few long vertical lines (large jumps in height) which typically indicate the formation of distinct clusters.

- Identifying Clusters: The dendrogram does show regions where points are grouped closely at lower heights, and these groupings merge at higher distances. This characteristic can be used to infer the

presence of clusters.

- Optimal Cluster Count: For a clear identification of three clusters, we would look for three vertical lines joining at a significantly higher height than others, which is not very distinct here. Single linkage often results in a "chaining effect", where clusters are elongated and might merge gradually, which can make the separation into exactly three clusters less obvious at a glance.

8. Do the same for the average linkage dendrogram of X. Does the dendrogram clearly reveal the presence of three clusters in X?

```
# Compute the linkage matrix using average linkage
Z_average = linkage(X, method='average')

# Plot the dendrogram for average linkage
plt.figure(figsize=(10, 8))
```

```
## <Figure size 2000x1600 with 0 Axes>
```

```
dendrogram(Z_average)
```

```
[illegible]
```

```
plt.title('Dendrogram for Average Linkage Clustering')
```

```
## Text(0.5, 1.0, 'Dendrogram for Average Linkage Clustering')
```

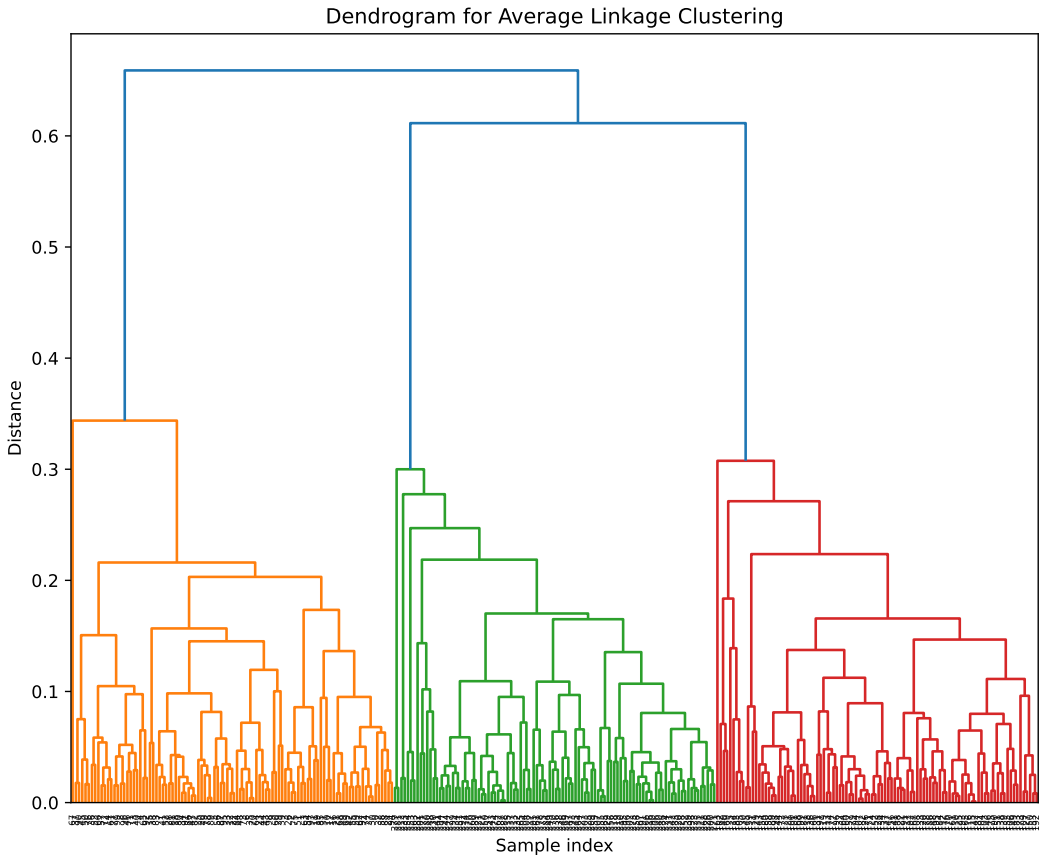
```
plt.xlabel('Sample index')
```

```
## Text(0.5, 0, 'Sample index')
```

```
plt.ylabel('Distance')
```

```
## Text(0, 0.5, 'Distance')
```

```
plt.show()
```



- Clearer Cluster Formation: The dendrigram with average linkage shows clearer and more distinct cluster formations than the single linkage dendrogram. This is because average linkage tends to mitigate the chaining effect seen in single linkage, providing a more balanced view of cluster distance.

- Presence of Three Clusters: The dendrogram reveals what appears to be three main clusters merging at higher levels. These clusters are indicated by the three major groupings of lines that merge at distinctly higher distances than the rest of the lower merges.

- Interpretation of Clusters: The vertical distances between the merges (height of the joins) are more uniform before combining into these three groups, suggesting that the data naturally forms into three clusters before any further merging occurs.