1. Implement k-means from scratch

You are given a dataset X whose rows represent different data points, you are asked to perform a k-means clustering on this dataset using the Manhattan Distance, k is chosen as 3.

X = [[5.7, 64], [4.7, 58], [6.1, 56], [4.6, 64], [5.4, 84], [4.9, 60], [5.0, 62], [6.4, 62], [5.1, 76], [6.0, 60]]

Note: The Manhattan Distance of 〖(x)〗_1,y_1) and 〖(x)〗_2,y_2) is calculated by Dist= |x_1- x_2 |+ |y_1- y_2 |.

a. Since first column and second column are not on the same scale. Before running K-means, this dataset needs to be preprocessed, Show the preprocessed dataset. (Answer in the format of [x1, x2], round your results to two decimal places, same as problems b and c)

In [1]:
```
import numpy as np

# Given dataset X
X = np.array([[5.7, 64], [4.7, 58], [6.1, 56], [4.6, 64], [5.4, 84], [4.9, 60], [5.0, 62

# Preprocess the dataset
min_vals = np.min(X, axis=0)
max_vals = np.max(X, axis=0)

X_scaled = (X - min_vals) / (max_vals - min_vals)

# Round the results to two decimal places
X_scaled = np.round(X_scaled,2)

print(X_scaled)
```

```
[[0.61 0.29]
 [0.06 0.07]
 [0.83 0.  ]
 [0.   0.29]
 [0.44 1.  ]
 [0.17 0.14]
 [0.22 0.21]
 [1.   0.21]
 [0.28 0.71]
 [0.78 0.14]]
```

b. Suppose the initial centroids of the clusters are μ_1=[5.6,60],μ_2=[5.9,60], 〖μ〗_3=[5.2,75] . What's the center of the second cluster after two iterations?

In [2]:
```
# Given initial centroids
mu_1 = np.array([5.6, 60])
mu_2 = np.array([5.9, 60])
mu_3 = np.array([5.2, 75])

# Function to calculate Manhattan distance
def manhattan_distance(point1, point2):
    return np.sum(np.abs(point1 - point2))

# Function to assign each point to the nearest centroid
def assign_to_clusters(X, centroids):
```

```
    clusters = []
    for point in X:
        distances = [manhattan_distance(point, centroid) for centroid in centroids]
        cluster = np.argmin(distances)
        clusters.append(cluster)
    return np.array(clusters)

# Function to update centroids based on the mean of points in each cluster
def update_centroids(X, clusters, k, current_centroids):
    new_centroids = []
    for i in range(k):
        cluster_points = X[clusters == i]
        if len(cluster_points) > 0:
            new_centroid = np.mean(cluster_points, axis=0)
            new_centroids.append(new_centroid)
        else:
            new_centroids.append(current_centroids[i])
    return np.array(new_centroids)

# K-Means iterations
max_iterations = 100
for iteration in range(max_iterations):
    # Assign points to clusters
    clusters = assign_to_clusters(X_scaled, [mu_1, mu_2, mu_3])

    # Update centroids
    new_mu_1, new_mu_2, new_mu_3 = update_centroids(X_scaled, clusters, 3, [mu_1, mu_2,

    # Check for convergence
    if np.array_equal(new_mu_1, mu_1) and np.array_equal(new_mu_2, mu_2) and np.array_eq
        break

    mu_1, mu_2, mu_3 = new_mu_1, new_mu_2, new_mu_3

# Results after convergence
print("Centroids after convergence:")
print("Cluster 1:", mu_1)
print("Cluster 2:", mu_2)
print("Cluster 3:", mu_3)
print("Number of iterations:", iteration + 1)
```

```
Centroids after convergence:
Cluster 1: [0.439 0.306]
Cluster 2: [ 5.9 60. ]
Cluster 3: [ 5.2 75. ]
Number of iterations: 2
```

c. What's the center of the third cluster when the clustering converges?

In [3]:
```
# The center of the third cluster when the clustering converges
print("Center of the third cluster when converges:", mu_3)
```

```
Center of the third cluster when converges: [ 5.2 75. ]
```

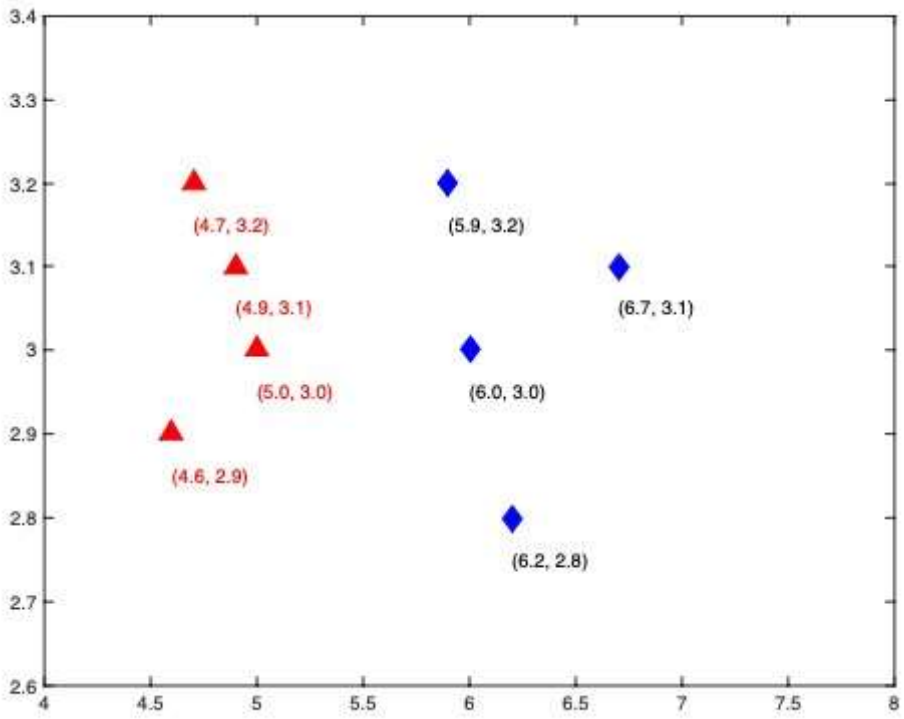
In [4]:
```
# Reset centroids to the initial values
mu_1 = np.array([5.6, 60])
mu_2 = np.array([5.9, 60])
mu_3 = np.array([5.2, 75])

# K-Means iterations
max_iterations = 100
converged = False
```

```
for iteration in range(max_iterations):
    # Assign points to clusters
    clusters = assign_to_clusters(X_scaled, [mu_1, mu_2, mu_3])

    # Update centroids
    new_mu_1, new_mu_2, new_mu_3 = update_centroids(X_scaled, clusters, 3, [mu_1, mu_2,

    # Check for convergence
    if np.array_equal(new_mu_1, mu_1) and np.array_equal(new_mu_2, mu_2) and np.array_eq
        converged = True
        break

    mu_1, mu_2, mu_3 = new_mu_1, new_mu_2, new_mu_3

# Print the results
if converged:
    print("Clusters converged after", iteration + 1, "iterations.")
    print("Centroids after convergence:")
    print("Cluster 1:", mu_1)
    print("Cluster 2:", mu_2)
    print("Cluster 3:", mu_3)
else:
    print("Clusters did not converge within the maximum number of iterations.")
```

```
Clusters converged after 2 iterations.
Centroids after convergence:
Cluster 1: [0.439 0.306]
Cluster 2: [ 5.9 60. ]
Cluster 3: [ 5.2 75. ]
```

2. Hierarchical Clustering

Suppose there are two clusters A (red) and B (blue), each has four members and plotted in Figure below, compute the distance between two clusters using Euclidean distance.



a. What is the distance between the two farthest members (Complete-link) (round to four decimal places here, and next 2 problems)?

Answer:

Finding the maximum Euclidean distance between any pair of members—where each member is from a different cluster—is necessary in order to determine the distance between two clusters using complete-linkage (or complete-link) hierarchical clustering. The maximum distance between any two members is therefore the distance between two clusters.

In [8]:
```
import numpy as np
# Cluster A (red)
A = np.array([[4.7, 3.2], [4.9, 3.1], [5.0, 3.0], [4.6, 2.9]])
# Cluster B (blue)
B = np.array([[5.9, 3.2], [6.0, 3.0], [6.2, 2.8], [6.7, 3.1]])

# Function to calculate Euclidean distance between two points
def euclidean_distance(point1, point2):
    return np.sqrt(np.sum((point1 - point2)**2))

# Calculate the complete-link distance between the two clusters
complete_link_distances = []
for point_A in A:
    for point_B in B:
        distance = euclidean_distance(point_A, point_B)
        complete_link_distances.append(distance)

# Find the maximum distance (complete-link)
max_complete_link_distance = np.max(complete_link_distances)

# Round the result to four decimal places
max_complete_link_distance = round(max_complete_link_distance, 4)
print("The distance between the two farthest members (Complete-link):", max_complete_lin
```

```
The distance between the two farthest members (Complete-link): 2.1095
```

b. What is the distance between the two closest members (Single-link)?

Answer: The distance between two clusters is then the minimum distance between any pair of members.

In [9]:
```
import numpy as np

# Cluster A (red)
A = np.array([[4.7, 3.2], [4.9, 3.1], [5.0, 3.0], [4.6, 2.9]])
# Cluster B (blue)
B = np.array([[5.9, 3.2], [6.0, 3.0], [6.2, 2.8], [6.7, 3.1]])

# Function to calculate Euclidean distance between two points
def euclidean_distance(point1, point2):
    return np.sqrt(np.sum((point1 - point2)**2))

# Calculate the single-link distance between the two clusters
single_link_distances = []
for point_A in A:
    for point_B in B:
        distance = euclidean_distance(point_A, point_B)
        single_link_distances.append(distance)
```

```
# Find the minimum distance (single-link)
min_single_link_distance = np.min(single_link_distances)
# Round the result to four decimal places
min_single_link_distance = round(min_single_link_distance, 4)

print("The distance between the two closest members (Single-link):", min_single_link_dis
```

```
The distance between the two closest members (Single-link): 0.922
```

c. What is the average distance between all pairs (Average-link)?

Answer: In order to find the distance between two clusters using average-linkage (or average-link) hierarchical clustering, you need to find the average Euclidean distance between all pairs of members, where each member is from a different cluster. The distance between two clusters is then the average distance between all pairs of members.

In [10]:
```
import numpy as np

# Cluster A (red)
A = np.array([[4.7, 3.2], [4.9, 3.1], [5.0, 3.0], [4.6, 2.9]])
# Cluster B (blue)
B = np.array([[5.9, 3.2], [6.0, 3.0], [6.2, 2.8], [6.7, 3.1]])

# Function to calculate Euclidean distance between two points
def euclidean_distance(point1, point2):
    return np.sqrt(np.sum((point1 - point2)**2))

# Calculate and print the distance for each pair
average_link_distances = []
print("Distance calculation for each pair:")
for i, point_A in enumerate(A):
    for j, point_B in enumerate(B):
        distance = euclidean_distance(point_A, point_B)
        average_link_distances.append(distance)
        print(f"Distance between A[{i}] and B[{j}]: {distance}")

# Calculate the average distance (average-link)
average_link_distance = np.mean(average_link_distances)
# Round the result to four decimal places
average_link_distance = round(average_link_distance, 4)

print("\nAverage distance between all pairs (Average-link):", average_link_distance)
```

```
Distance calculation for each pair:
Distance between A[0] and B[0]: 1.2000000000000002
Distance between A[0] and B[1]: 1.3152946437965904
Distance between A[0] and B[2]: 1.5524174696260025
Distance between A[0] and B[3]: 2.0024984394500787
Distance between A[1] and B[0]: 1.004987562112089
Distance between A[1] and B[1]: 1.1045361017187258
Distance between A[1] and B[2]: 1.3341664064126333
Distance between A[1] and B[3]: 1.7999999999999998
Distance between A[2] and B[0]: 0.9219544457292891
Distance between A[2] and B[1]: 1.0
Distance between A[2] and B[2]: 1.216552506059644
Distance between A[2] and B[3]: 1.7029386365926404
Distance between A[3] and B[0]: 1.3341664064126342
Distance between A[3] and B[1]: 1.4035668847618203
Distance between A[3] and B[2]: 1.6031219541881403
Distance between A[3] and B[3]: 2.109502310972899

Average distance between all pairs (Average-link): 1.4129
```

d. Among all three distances above, which one is robust to noise?

Answer:

In hierarchical clustering, three distances are used: complete-link, single-link, and average-link. Of these, the average-link distance is typically thought to be more noise-resistant than the complete-link and single-link distances.

This is because the average-link distance measures the average separation between every pair of points from every cluster. Compared to complete-link and single-link distances, which are impacted by the maximum and minimum distances, respectively, it is therefore less susceptible to outliers or noisy data points.

For this reason, when a dataset has noise or outliers, the average-link distance is frequently chosen. By taking into account the total distribution of point-to-point distances, it offers a more impartial estimate of cluster similarity.