K-Means Clustering

KATO PAUL 2023/U/MMU/BCS/00104
MPAGI DERRICK BRAIR 2023/U/MMU/BCS/00101
NAMONYE KENNETH 2023/U/MMU/BCS/01668
BEINOMUGISHA WYCLIEF 2023/U/MMU/BCS/01681
KARAMAGI HENRY 2023/U/MMU/BCS/00523

K MEANS

K-Means Clustering is an Unsupervised Machine Learning algorithm which groups the unlabeled dataset into different clusters. Here K defines the number of pre-defined clusters or groups that need to be created in the process, as if K=5, there will be five clusters, and for K=10, there will be ten clusters, and so on.

Understanding K-means Clustering

K-means clustering is a technique used to organize data into groups based on their similarity. For example, an online store uses K-Means to group customers based on purchase frequency and spending, creating segments like Budget Shoppers, Frequent Buyers, and Big Spenders for personalized marketing.

The algorithm works by first randomly picking some central points called centroids, and each data point is then assigned to the closest centroid forming a cluster. After all the points are assigned to a cluster, the centroids are updated by finding the average position of the points in each cluster. This process repeats until the centroids stop changing, forming clusters. The goal of clustering is to divide the data points into clusters so that similar data points belong to the same group.

How K-Means Clustering Works?

The algorithm will categorize the items into k groups or clusters of similarity. To calculate that similarity, we will use the Euclidean distance as a measurement. The algorithm works as follows:

Step 1: **Initialize** - Randomly select ”k” centroids, called means or cluster centroids.

Step 2: **Assign** - For each data point, calculate the Euclidean distance to all centroids and assign it to the nearest one forming a cluster. The Euclidean distance between a data point x_i and a centroid c_j is computed as:

$$d(x_i, c_j) = \sqrt{\sum_{k=1}^{d} (x_{i,k} - c_{j,k})^2}$$

where $x_{i,k}$ is the k-th feature of data point x_i and $c_{j,k}$ is the k-th feature of centroid c_j.

Step 3: Assign each data point x_i to the cluster j whose centroid is closest (i.e., the centroid with the smallest distance).

Step 4: **Update** - Recalculate each centroid as the mean of all points assigned to it.

$$c_j = \frac{1}{|C_j|} \sum_{x_i \in C_j} x_i$$

where C_j is the set of points assigned to cluster j, and $|C_j|$ is the number of points in cluster j.

Step 5: **Check Convergence** - If centroids change significantly, repeat Steps 2-3; if no change occurs, convergence has happened.

MATHEMATICAL CONCEPT

Step 1: The Elbow Method

The Elbow method is the best way to determine the number of clusters. It involves running K-Means clustering on the dataset.

Next, we use within-sum-of-squares (WSS) as a measure to find the optimum number of clusters for a given dataset. WSS is defined as the sum of the squared distances between each member of the cluster and its centroid.

$$\text{WSS} = \sum_{i=1}^{m} (x_i - c_i)^2$$

Where x_i = data point and c_i = closest point to centroid

Figure 1: Elbow Curve: WSS vs Number of Clusters

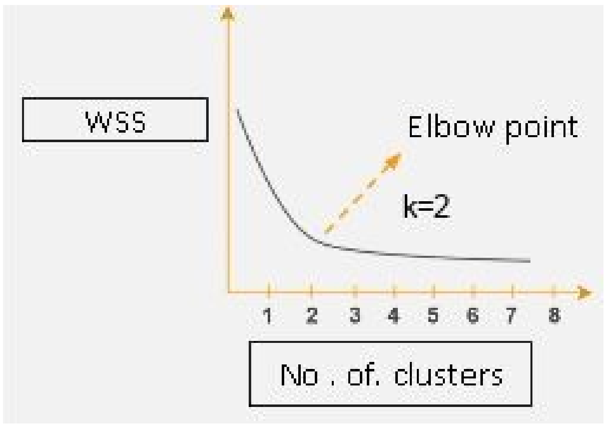


Figure 2: Elbow Curve: WSS vs Number of Clusters

The optimal value of K is typically found at the ”elbow” of the curve, where WSS stops decreasing significantly.

Step 2: Initial Centroids

We randomly initialize two points called cluster centroids.

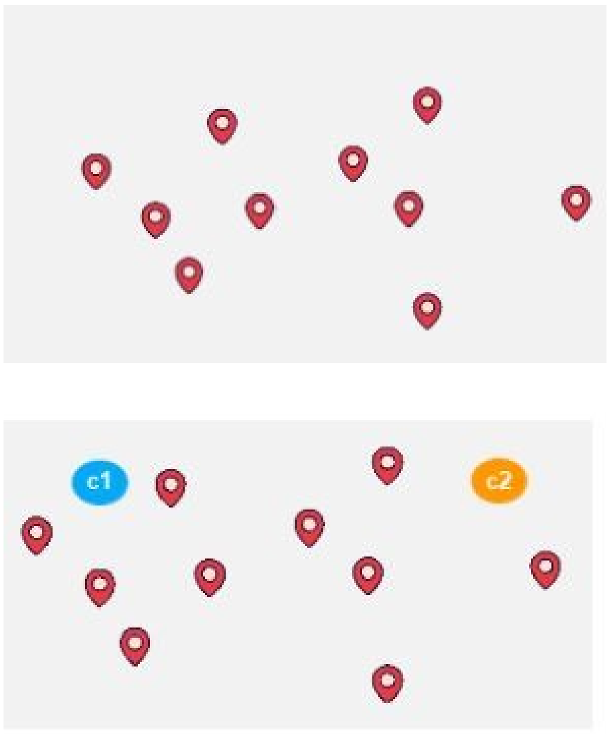


Figure 3: Initial Random Centroids C1 and C2

Step 3: Assigning Data Points to Clusters

Now, the distance of each location from the centroid is measured, and each data point is assigned to the nearest centroid.

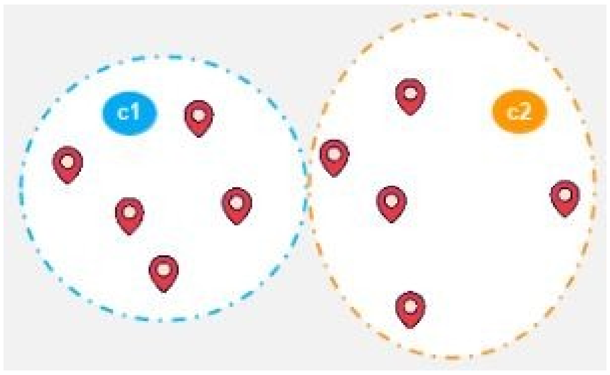


Figure 4: Initial Grouping of Data Points

Step 4: Computing New Centroids

Compute the actual centroid of data points for the first group.

Step 5: Repositioning Centroids

Reposition the random centroid to the actual centroid.

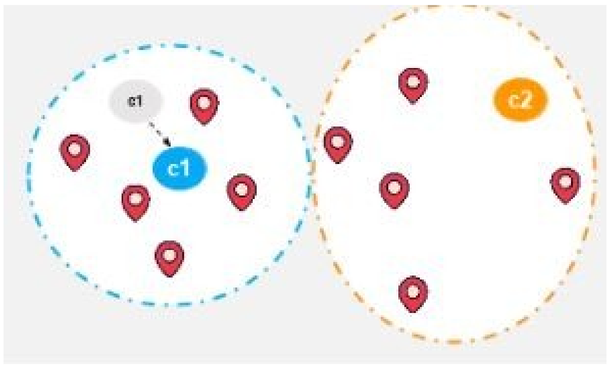


Figure 5: Repositioned Centroid for First Group

Step 6: Computing Second Centroid

Compute the actual centroid of data points for the second group.

Step 7: Repositioning Second Centroid

Reposition the random centroid to the actual centroid.

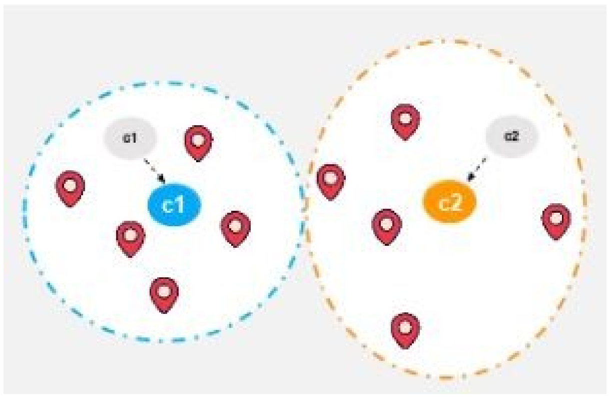


Figure 6: Repositioned Centroid for Second Group

Step 8: Final Clusters

Once the clusters become static, the K-Means algorithm has converged. The final cluster with centroids C1 and C2 is shown below:

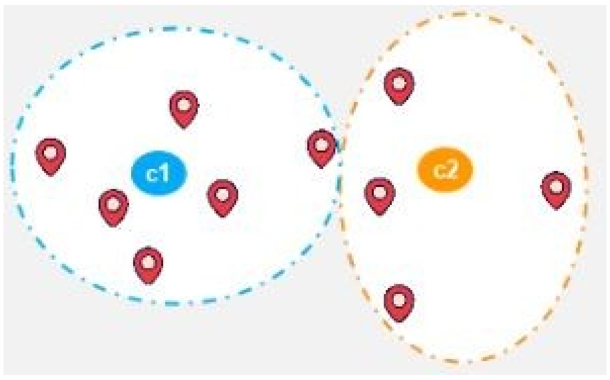


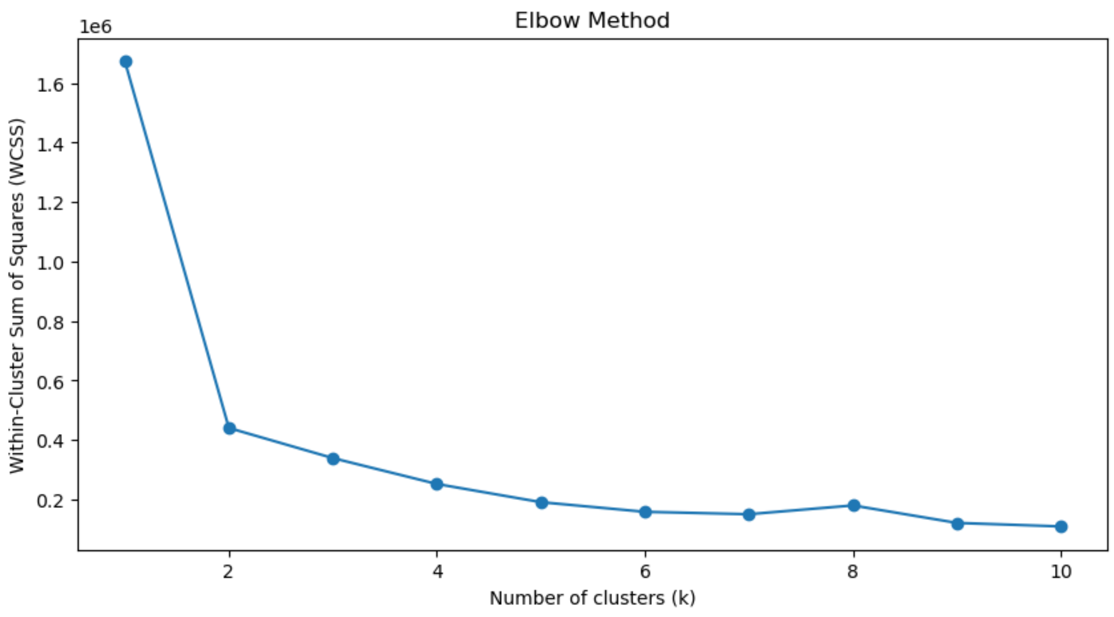
Figure 7: Final Clusters with Centroids C1 and C2

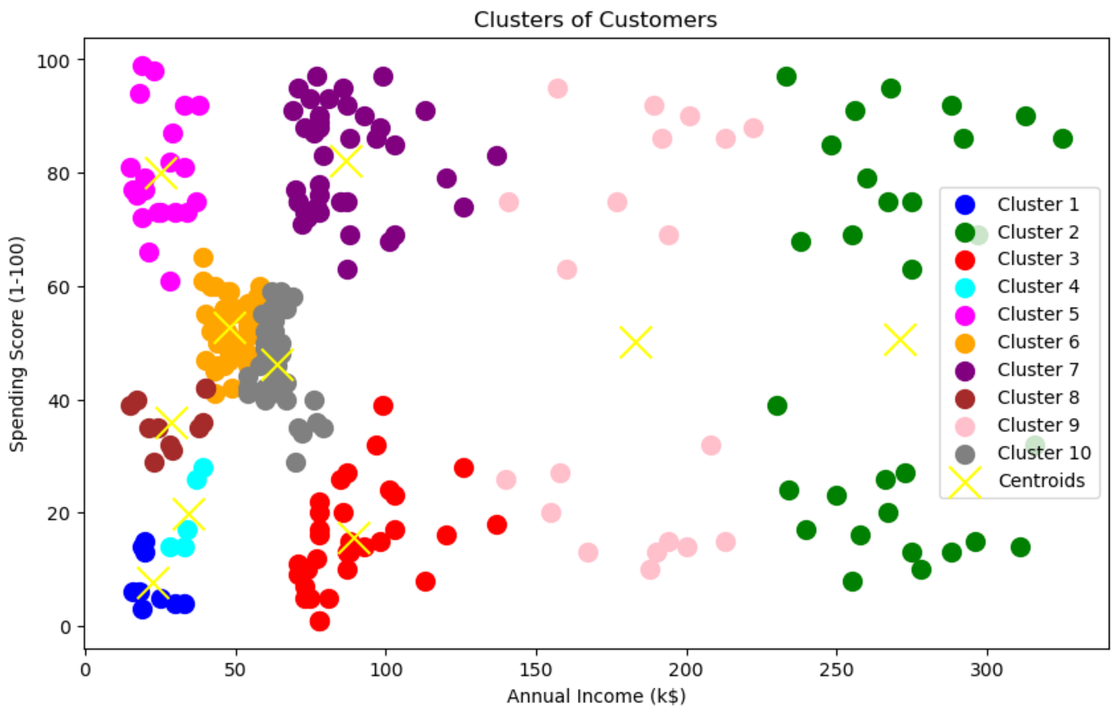
Untitled121

March 4, 2025

[3]:
```python
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd

class KMeansClustering:
    def __init__(self, k=5, max_iters=100):
        """
        Initialize K-Means Clustering

        Parameters:
        k (int): Number of clusters
        max_iters (int): Maximum number of iterations
        """
        self.k = k
        self.max_iters = max_iters
        self.centroids = None

    def euclidean_distance(self, x1, x2):
        """
        Calculate Euclidean distance between two points

        Parameters:
        x1 (array): First point
        x2 (array): Second point

        Returns:
        float: Euclidean distance
        """
        return np.sqrt(np.sum((x1 - x2) ** 2))

    def initialize_centroids(self, X):
        """
        Randomly initialize centroids

        Parameters:
        X (array): Input data
        Returns:
        array: Randomly selected initial centroids
        """
        n_samples, n_features = X.shape
        centroids = X[np.random.choice(n_samples, self.k, replace=False)]
        return centroids

    def assign_clusters(self, X, centroids):
        """
        Assign each data point to the nearest centroid

        Parameters:
        X (array): Input data
        centroids (array): Current centroids

        Returns:
        array: Cluster assignments for each data point
        """
        clusters = []
        for x in X:
            # Calculate distances to all centroids
            distances = [self.euclidean_distance(x, centroid) for centroid in␣
↪centroids]
            # Assign to the cluster with minimum distance
            cluster_idx = np.argmin(distances)
            clusters.append(cluster_idx)
        return np.array(clusters)

    def compute_centroids(self, X, clusters):
        """
        Compute new centroids as the mean of points in each cluster

        Parameters:
        X (array): Input data
        clusters (array): Current cluster assignments

        Returns:
        array: Updated centroids
        """
        centroids = []
        for k in range(self.k):
            # Find all points in this cluster
            cluster_points = X[clusters == k]
            # Compute centroid as mean of cluster points
            centroid = np.mean(cluster_points, axis=0)
            centroids.append(centroid)
        return np.array(centroids)
    def fit(self, X):
        """
        Perform K-Means clustering

        Parameters:
        X (array): Input data

        Returns:
        tuple: Final clusters and centroids
        """
        # Initialize centroids
        self.centroids = self.initialize_centroids(X)

        # Iterate until convergence or max iterations
        for _ in range(self.max_iters):
            # Assign clusters
            old_clusters = self.assign_clusters(X, self.centroids)

            # Compute new centroids
            new_centroids = self.compute_centroids(X, old_clusters)

            # Check for convergence
            if np.allclose(self.centroids, new_centroids):
                break

            self.centroids = new_centroids

        return old_clusters, self.centroids

    def elbow_method(self, X, max_k=10):
        """
        Compute Within-Cluster Sum of Squares (WCSS) for different k values

        Parameters:
        X (array): Input data
        max_k (int): Maximum number of clusters to try

        Returns:
        list: WCSS values for different k
        """
        wcss = []
        for k in range(1, max_k + 1):
            self.k = k
            clusters, centroids = self.fit(X)

            # Calculate WCSS
            total_wcss = 0
            for i in range(k):
                cluster_points = X[clusters == i]
                cluster_wcss = np.sum((cluster_points - centroids[i]) ** 2)
                total_wcss += cluster_wcss

            wcss.append(total_wcss)

        return wcss

# Load the dataset
dataset = pd.read_csv("C:\\Users\\hp\\Desktop\\ken\\Mall_Customers.csv")
X = dataset.iloc[:, [3, 4]].values

# Create K-Means instance
kmeans = KMeansClustering(k=5)

# Perform Elbow Method
wcss_list = kmeans.elbow_method(X)

# Plot Elbow Method results
plt.figure(figsize=(10, 5))
plt.plot(range(1, 11), wcss_list, marker='o')
plt.title('Elbow Method')
plt.xlabel('Number of clusters (k)')
plt.ylabel('Within-Cluster Sum of Squares (WCSS)')
plt.show()

# Perform clustering with 5 clusters
clusters, centroids = kmeans.fit(X)

# Visualize the clusters
plt.figure(figsize=(10, 6))

# Define a list of colors and labels
colors = ['blue', 'green', 'red', 'cyan', 'magenta', 'orange', 'purple',␣
 ↪'brown', 'pink', 'gray']
labels = ['Cluster 1', 'Cluster 2', 'Cluster 3', 'Cluster 4', 'Cluster 5',
          'Cluster 6', 'Cluster 7', 'Cluster 8', 'Cluster 9', 'Cluster 10']

# Plot each cluster with a different color
for i in range(kmeans.k):
    cluster_points = X[clusters == i]
    plt.scatter(cluster_points[:, 0], cluster_points[:, 1],
                c=colors[i], label=labels[i], s=100)

# Plot centroids
```

```
plt.scatter(centroids[:, 0], centroids[:, 1],
            c='yellow', label='Centroids', s=300, marker='x')

plt.title('Clusters of Customers')
plt.xlabel('Annual Income (k$)')
plt.ylabel('Spending Score (1-100)')
plt.legend()
plt.show()

# Print cluster assignments
print("Cluster Assignments:", clusters)
```





```
Cluster Assignments: [7 4 0 4 7 4 0 4 0 4 0 4 0 4 0 4 7 4 7 4 7 4 0 4 3 4 7 4 7
4 0 4 0 4 3 4 3
 4 3 4 7 4 7 5 3 5 5 5 7 7 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5
 5 5 5 5 5 9 5 5 9 9 5 5 5 5 5 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9
 9 9 9 9 9 9 9 9 9 9 9 9 6 9 6 9 6 2 6 2 6 9 6 2 6 2 6 2 6 2 6 9 6 2 6 9 6
 2 6 2 6 2 6 2 6 2 6 2 6 9 6 2 6 2 6 2 6 2 6 2 6 2 6 2 6 2 6 2 6 2 6 2 6 2
 6 2 6 2 6 2 6 2 6 2 6 2 6 2 6 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1]
```

[ ]:

Fuzzy C-Means Clustering

March 2025

1 Introduction

Fuzzy C-Means (FCM) clustering is a method of clustering that allows one piece of data to belong to more than one cluster. This is in contrast to traditional clustering methods like K-Means, where each data point is assigned to only one cluster. FCM is based on the concept of fuzzy sets, where each data point has a degree of belonging to each cluster rather than belonging exclusively to one cluster.

1.1 Definition of Fuzzy C-Means

The Fuzzy C-Means algorithm divides a set of data points into a specified number C of clusters. Each data point is assigned a membership value for each cluster, indicating the degree of association between the point and each cluster. The goal of FCM is to minimize the following objective function:

$$J(U, V) = \sum_{i=1}^{n} \sum_{c=1}^{C} (\mu_{ic})^m \cdot \|x_i - v_c\|^2$$

where:

- n is the number of data points,
- C is the number of clusters,
- μ_{ic} is the membership degree of data point x_i in cluster c,
- v_c is the center of cluster c,
- $\| \cdot \|$ is the Euclidean distance, and
- m is the fuzziness parameter, controlling the degree of fuzziness (typically $m > 1$).

The algorithm minimizes this objective function by adjusting the membership values μ_{ic} and the cluster centers v_c iteratively.

1.2 How Fuzzy C-Means Differs from K-Means

The key difference between Fuzzy C-Means and K-Means lies in how data points are assigned to clusters:

- **K-Means Clustering:** In K-Means, each data point is assigned to exactly one cluster, based on its proximity to the nearest cluster center. The assignment is crisp, meaning a point either belongs to a cluster or it does not.
- **Fuzzy C-Means Clustering:** In contrast, FCM assigns each data point a degree of membership to every cluster. A data point can belong to multiple clusters with varying degrees of membership. This is particularly useful when the data has inherent overlapping characteristics.

The fuzziness parameter m in FCM controls how soft or hard the assignments are. When m is set to 1, the algorithm reduces to a crisp classification like K-Means. As m increases, the algorithm becomes fuzzier, allowing for more overlap between clusters.

2 Fuzzy C-Means Clustering Algorithm: Step-by-Step Explanation

2.1 Step 1: Define the Problem

We have four data points in 2D space:

$$x_1 = (1, 3), \quad x_2 = (1.5, 3.2), \quad x_3 = (1.3, 2.8), \quad x_4 = (3, 1)$$

Here, $x_k = (x_{k1}, x_{k2})$ represents the k-th data point in a 2-dimensional space, where x_{k1} and x_{k2} are the first and second features of the point.

2.2 Step 2: Set Algorithm Parameters

- Number of Clusters (C): $C = 2$. This is the number of clusters to form.
- Fuzziness Parameter (m): $m = 2$. This controls the degree of overlap between clusters.
- Convergence Criterion (ϵ): $\epsilon = 0.01$. This is the threshold for stopping the algorithm.

2.3 Step 3: Initialize the Membership Matrix

We initialize the membership matrix U_0 as:

$$U_0 = \begin{bmatrix} 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

In this initial matrix:

- The first three points belong to Cluster 1.
- The fourth point belongs to Cluster 2.

2.4 Step 4: Calculate Cluster Centers

The cluster centers V are calculated using the formula:

$$V_{ij} = \frac{\sum_{k=1}^{n} (\mu_{ik})^m \cdot x_{kj}}{\sum_{k=1}^{n} (\mu_{ik})^m}$$

where:

- μ_{ik} is the membership of the k-th data point in the i-th cluster raised to the power m,
- x_{kj} is the j-th feature of the k-th data point,
- n is the total number of data points.

2.4.1 Cluster 1 Center (V_1)

$$V_{11} = \frac{(1^2 \cdot 1) + (1^2 \cdot 1.5) + (1^2 \cdot 1.3) + (0^2 \cdot 3)}{1^2 + 1^2 + 1^2 + 0^2} = \frac{1 + 1.5 + 1.3 + 0}{3} = 1.26$$

$$V_{12} = \frac{(1^2 \cdot 3) + (1^2 \cdot 3.2) + (1^2 \cdot 2.8) + (0^2 \cdot 1)}{1^2 + 1^2 + 1^2 + 0^2} = \frac{3 + 3.2 + 2.8 + 0}{3} = 3.0$$

So, $V_1 = (1.26, 3.0)$.

2.4.2 Cluster 2 Center (V_2)

$$V_{21} = \frac{(0^2 \cdot 1) + (0^2 \cdot 1.5) + (0^2 \cdot 1.3) + (1^2 \cdot 3)}{0^2 + 0^2 + 0^2 + 1^2} = \frac{0 + 0 + 0 + 3}{1} = 3$$

$$V_{22} = \frac{(0^2 \cdot 3) + (0^2 \cdot 3.2) + (0^2 \cdot 2.8) + (1^2 \cdot 1)}{0^2 + 0^2 + 0^2 + 1^2} = \frac{0 + 0 + 0 + 1}{1} = 1$$

So, $V_2 = (3, 1)$.

2.5 Step 5: Calculate Distances

Compute the Euclidean distance of each data point from the cluster centers.

2.5.1 Distance Formula

$$d_{ik} = \sqrt{(x_{kj} - V_{ij})^2}$$

where x_{kj} is the j-th feature of the k-th data point, and V_{ij} is the j-th feature of the center of cluster i.

2.5.2 Calculations

- Distance from $x_1 = (1, 3)$:

$$d_{11} = \sqrt{(1 - 1.26)^2 + (3 - 3)^2} = \sqrt{(-0.26)^2 + 0} = 0.26$$

$$d_{21} = \sqrt{(1 - 3)^2 + (3 - 1)^2} = \sqrt{(-2)^2 + (2)^2} = \sqrt{4 + 4} = 2.82$$

- Distance from $x_2 = (1.5, 3.2)$:

$$d_{12} = \sqrt{(1.5 - 1.26)^2 + (3.2 - 3)^2} = \sqrt{(0.24)^2 + (0.2)^2} = \sqrt{0.0576 + 0.04} = 0.31$$

$$d_{22} = \sqrt{(1.5 - 3)^2 + (3.2 - 1)^2} = \sqrt{(-1.5)^2 + (2.2)^2} = \sqrt{2.25 + 4.84} = 2.66$$

- Distance from $x_3 = (1.3, 2.8)$:

$$d_{13} = \sqrt{(1.3 - 1.26)^2 + (2.8 - 3)^2} = \sqrt{(0.04)^2 + (-0.2)^2} = \sqrt{0.0016 + 0.04} = 0.20$$

$$d_{23} = \sqrt{(1.3 - 3)^2 + (2.8 - 1)^2} = \sqrt{(-1.7)^2 + (1.8)^2} = \sqrt{2.89 + 3.24} = 2.47$$

- Distance from $x_4 = (3, 1)$:

$$d_{14} = \sqrt{(3 - 1.26)^2 + (1 - 3)^2} = \sqrt{(1.74)^2 + (-2)^2} = \sqrt{3.0276 + 4} = 2.65$$

$$d_{24} = \sqrt{(3 - 3)^2 + (1 - 1)^2} = \sqrt{0 + 0} = 0$$

2.6 Step 6: Update Membership Matrix

The updated membership values are calculated using:

$$\mu_{ik}^{r+1} = \left(\sum_{j=1}^{C} \left(\frac{d_{ik}}{d_{jk}} \right)^{\frac{2}{m-1}} \right)^{-1}$$

2.6.1 Calculations

- For $x_1 = (1, 3)$:

$$\mu_{11} = \left(\left(\frac{0.26}{0.26} \right)^2 + \left(\frac{0.26}{2.82} \right)^2 \right)^{-1} = (1 + 0.0085)^{-1} = 0.991$$

$$\mu_{21} = 1 - \mu_{11} = 1 - 0.991 = 0.009$$

- For $x_2 = (1.5, 3.2)$:

$$\mu_{12} = \left(\left(\frac{0.31}{0.31} \right)^2 + \left(\frac{0.31}{2.66} \right)^2 \right)^{-1} = (1 + 0.0138)^{-1} = 0.982$$

$$\mu_{22} = 1 - \mu_{12} = 1 - 0.982 = 0.018$$

- For $x_3 = (1.3, 2.8)$:

$$\mu_{13} = \left(\left(\frac{0.20}{0.20} \right)^2 + \left(\frac{0.20}{2.47} \right)^2 \right)^{-1} = (1 + 0.0325)^{-1} = 0.993$$

$$\mu_{23} = 1 - \mu_{13} = 1 - 0.993 = 0.007$$

- For $x_4 = (3, 1)$:

$$\mu_{14} = \left(\left(\frac{2.65}{2.65} \right)^2 + \left(\frac{2.65}{0} \right)^2 \right)^{-1} = (1 + \infty)^{-1} = 0$$

$$\mu_{24} = 1 - \mu_{14} = 1 - 0 = 1$$

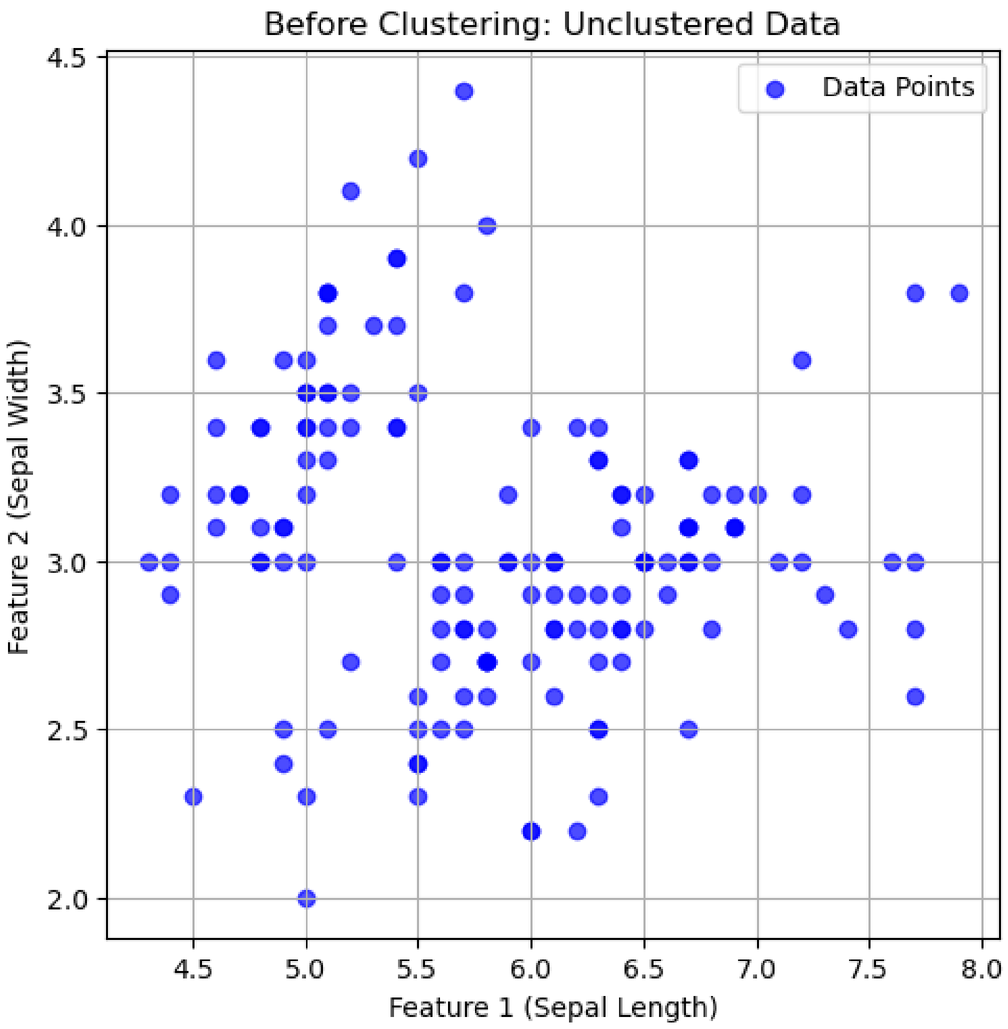
2.7 Step 7: Repeat Steps 4-6

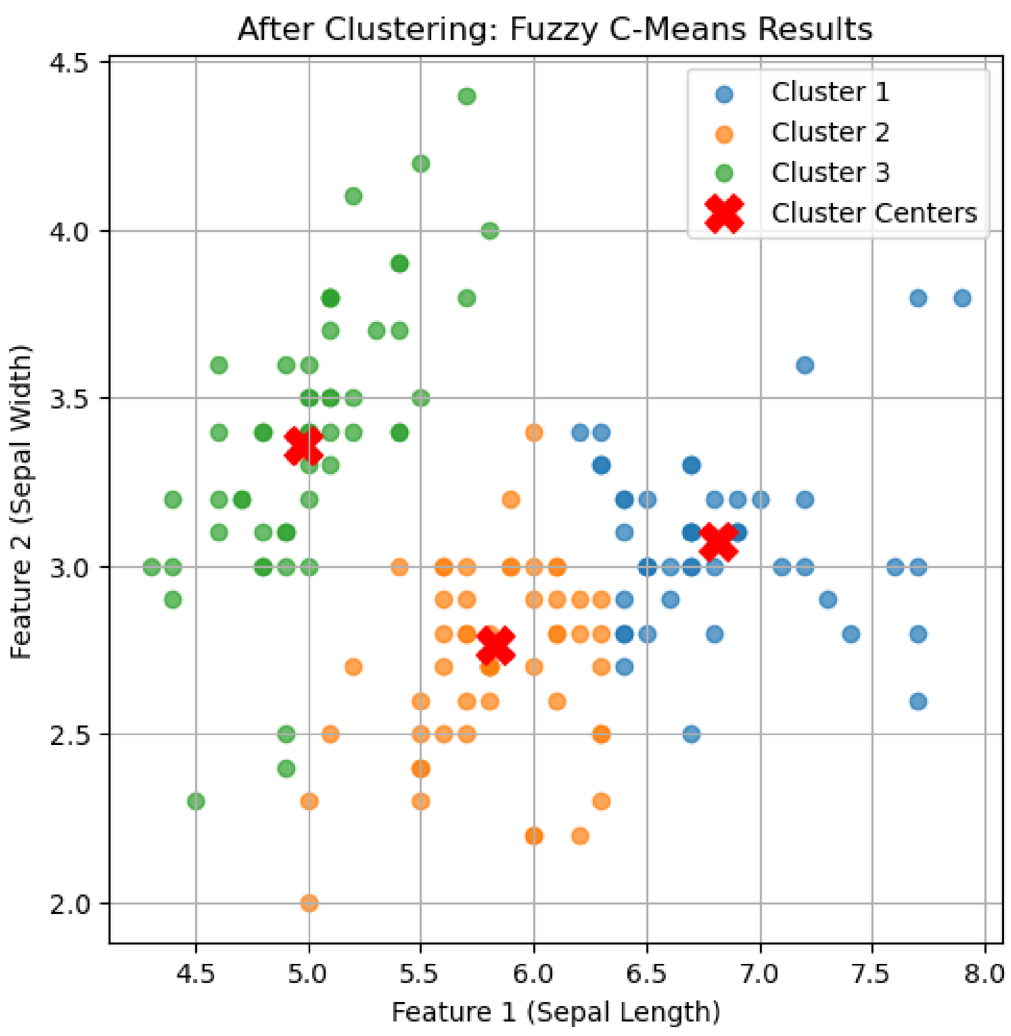
We repeat Steps 4-6 until the membership matrix converges, i.e., when the change in membership values is smaller than the convergence criterion ϵ.

Untitled120

March 4, 2025

[3]:
```python
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import load_iris

class FuzzyCMeans:
    """
    Fuzzy C-Means Clustering Algorithm.
    Parameters:
    C : int
        The number of clusters.
    m : float
        The fuzziness parameter (m > 1).
    epsilon : float, optional
        The convergence criterion. Default is 0.01.
    max_iter : int, optional
        The maximum number of iterations. Default is 100.
    """
    def __init__(self, C, m, epsilon=0.01, max_iter=100):
        self.C = C
        self.m = m
        self.epsilon = epsilon
        self.max_iter = max_iter
        self.U = None
        self.V = None

    def fit(self, X):
        """
        Fit the Fuzzy C-Means model to the data.
        Parameters:
        X : numpy array, shape (n_samples, n_features)
            The input data to cluster.
        """
        n_samples, n_features = X.shape

        # Step 1: Initialize the membership matrix U randomly
        self.U = np.random.rand(n_samples, self.C)
        self.U = self.U / np.sum(self.U, axis=1, keepdims=True)
        for iteration in range(self.max_iter):
            # Step 2: Calculate cluster centers V
            U_m = self.U ** self.m
            self.V = np.dot(U_m.T, X) / np.sum(U_m.T, axis=1, keepdims=True)

            # Step 3: Calculate the distance matrix D
            D = np.zeros((n_samples, self.C))
            for i in range(self.C):
                D[:, i] = np.linalg.norm(X - self.V[i], axis=1)

            # Step 4: Update the membership matrix U
            U_new = np.zeros((n_samples, self.C))
            for i in range(self.C):
                for j in range(self.C):
                    U_new[:, i] += (D[:, i] / D[:, j]) ** (2 / (self.m - 1))
            U_new = 1 / U_new

            # Step 5: Check for convergence
            if np.linalg.norm(U_new - self.U) < self.epsilon:
                break

            self.U = U_new

    def predict(self, X):
        """
        Predict the cluster membership for new data points.
        Parameters:
        X : numpy array, shape (n_samples, n_features)
            The input data to predict.
        Returns:
        U : numpy array, shape (n_samples, C)
            The membership matrix for the input data.
        """
        n_samples, n_features = X.shape
        D = np.zeros((n_samples, self.C))
        for i in range(self.C):
            D[:, i] = np.linalg.norm(X - self.V[i], axis=1)

        U_new = np.zeros((n_samples, self.C))
        for i in range(self.C):
            for j in range(self.C):
                U_new[:, i] += (D[:, i] / D[:, j]) ** (2 / (self.m - 1))
        U_new = 1 / U_new
        return U_new

    def get_cluster_centers(self):
        """
        Get the cluster centers.
        Returns:
        V : numpy array, shape (C, n_features)
            The cluster centers.
        """
        return self.V

    def get_membership_matrix(self):
        """
        Get the membership matrix.
        Returns:
        U : numpy array, shape (n_samples, C)
            The membership matrix.
        """
        return self.U

# Load the Iris dataset
data = load_iris()
X = data.data[:, :2]  # Use only the first two features for visualization

# Step 1: Plot the unclustered data points
plt.figure(figsize=(6, 6))
plt.scatter(X[:, 0], X[:, 1], c='blue', label='Data Points', alpha=0.7)
plt.title('Before Clustering: Unclustered Data')
plt.xlabel('Feature 1 (Sepal Length)')
plt.ylabel('Feature 2 (Sepal Width)')
plt.legend()
plt.grid(True)
plt.show()

# Step 2: Apply Fuzzy C-Means Clustering
fcm = FuzzyCMeans(C=3, m=2, epsilon=0.01, max_iter=100)
fcm.fit(X)  # Fit the model to the data

# Get the results
U = fcm.get_membership_matrix()
V = fcm.get_cluster_centers()

# Assign each data point to the cluster with the highest membership
cluster_assignments = np.argmax(U, axis=1)

# Step 3: Plot the clustered data points
plt.figure(figsize=(6, 6))
for i in range(fcm.C):
    plt.scatter(
        X[cluster_assignments == i, 0],  # Feature 1
        X[cluster_assignments == i, 1],  # Feature 2
        label=f'Cluster {i + 1}',
        alpha=0.7
    )

# Plot the cluster centers
plt.scatter(
    V[:, 0],  # Cluster center feature 1
    V[:, 1],  # Cluster center feature 2
    s=200,  # Size of the markers
    c='red',  # Color
    marker='X',  # Marker style
    label='Cluster Centers'
)

# Add labels and legend
plt.title('After Clustering: Fuzzy C-Means Results')
plt.xlabel('Feature 1 (Sepal Length)')
plt.ylabel('Feature 2 (Sepal Width)')
plt.legend()
plt.grid(True)
plt.show()

# Step 4: Model Evaluation
def fuzzy_partition_coefficient(U):
    """Calculate the Fuzzy Partition Coefficient (FPC)."""
    return np.mean(U**2)

def fuzzy_partition_entropy(U):
    """Calculate the Fuzzy Partition Entropy (FPE)."""
    return -np.mean(U * np.log(U))

# Calculate evaluation metrics
fpc = fuzzy_partition_coefficient(U)
fpe = fuzzy_partition_entropy(U)

print("Model Evaluation:")
print(f"Fuzzy Partition Coefficient (FPC): {fpc:.4f}")
print(f"Fuzzy Partition Entropy (FPE): {fpe:.4f}")

# Interpretation of metrics
print("\nInterpretation:")
print("- FPC ranges from 0 to 1. A value closer to 1 indicates better␣
 ↪clustering.")
print("- FPE ranges from 0 to infinity. A value closer to 0 indicates better␣
 ↪clustering.")
```


Before Clustering: Unclustered Data
Data Points
Feature 1 (Sepal Length)
Feature 2 (Sepal Width)
4.5
5.0
5.5
6.0
6.5
7.0
7.5
8.0
2.0
2.5
3.0
3.5
4.0
4.5



```
Model Evaluation:
Fuzzy Partition Coefficient (FPC): 0.2342
Fuzzy Partition Entropy (FPE): 0.1783

Interpretation:
- FPC ranges from 0 to 1. A value closer to 1 indicates better clustering.
- FPE ranges from 0 to infinity. A value closer to 0 indicates better
clustering.
```

[ ]: