Indian Institute of Technology Patna

CS564: Foundation of Machine Learning

Assignment #1: K-Means and K-Medoid

Submission Date: 13th April 2024

Baskar Natarajan - 2403res19(IITP001799)

Jyotisman Kar – 240res35(IITP001751)

SEMESTER-1

MTECH AI & DSC

IIT PATNA.

1. Python Libraries used are

1. **Pandas** - to read the CSV file as Data Frame and do some manipulations
2. **NumPy** – Array functions and data manipulations. Here np= numpy
    a. *np.random.choice* - randomly pick the element
    b. *np.sqrt* - calculate squire route of a value
    c. *np.newaxis* - adds a new axis at the beginning/ending
    d. *np.argmin* - returns the indices of the minimum values along a specified axis
    e. *np.array* - create numpy array
    f. *np.allclose* - Check if all elements are close within a tolerance
    g. *np.copy* - copy the array to another numpy array
    h. *np.sum* - adding the elements of the array
    i. *np.array_equal* – check if arrays have the same shape and exact element-wise equality without any tolerance for numerical differences
    j. *np.where* - get the indices of the array element
    k. *np.all* - get all elements which matches the condition
    l. *np.bincount* - counts occurrences of each non-negative integer in an array
3. **Matplotlib**.pyplot - to draw scatter plot graph
4. **Time** – time functions like time.time() etc

2. K- Means Algorithm

a. Algorithm:

i. Choose K initial centroids randomly from the dataset
ii. Calculate the Euclidean distance between dataset element and centroids and get the distance array
iii. Repeat until convergence or max iterations:
iv. For each data point x:
    1. Find out the minimum distance indices as labels
        - Using the above labels calculate the mean of all points in the cluster which gives the new centroid
        - Check if the old centroid and new centroid are close
    2. If yes, break, else continue
        a. Calculate the Euclidean distance between dataset element and centroids and get the distance array

v. Output the final clusters and centroids.

b. Major Functions

- calculate_sse – Calculate Sum of Squire Error

Note:

- One change in the default algorithm which is,
- Pre-calculated the distances array before the convergence array or max_ittr array for performance reasons.

c. Use K-Means When

- You're new to clustering and want to understand the basic concepts.
- You're dealing with well-separated, spherical clusters.
- You have large datasets and require a fast and efficient clustering algorithm.

d. Avoid K-Means When

- The distance metric used in K-Means (Euclidean) isn't suitable for your data.
- You're unsure about the optimal number of clusters (k).
- Your clusters are not spherical in shape.
- Your data contains outliers that might skew the cluster centers.

e. Strengths and Weakness of K-Means

a. *Strengths*:
   - Well-suited for datasets with spherical clusters
   - Easy to understand and implement, often the first choice for beginners in clustering.
   - Simple and computationally efficient, making it suitable for large datasets.

b. *Weakness:*
   - Uses Euclidean distance, which might not be suitable for all data types.
   - Requires pre-defining the number of clusters (k) upfront, which can be challenging.
   - Not ideal for non-spherical clusters.

- Sensitive to outliers.

3. K-Medoid Algorithm:

a. Major Functions:

- calculate_sse – Calculate Sum of Squire Error

b. Algorithm:

i. Choose K initial medoids randomly from the dataset
ii. Calculate the Euclidean distance between dataset element and medoids which gives distance array.
iii. Repeat until convergence or max iterations:
    1. Compute labels based on the distance's matrix created
    1. For each cluster:
        a. Compute distances to all medoids
        b. Find the minimum distance medoid element
        c. Assign x to the cluster of the nearest medoid
iv. Check if the old medoid and new medoid are equal,
v. if yes break, else continue
    1. Assign new medoid
    1. Calculate the medoid indices again, based on new medoid
    1. Output the final clusters and medoids.

Note: The Distance matrix is precalculated to improve the performance.

c. Use K-Medoid When

- **Unsure About Number of Clusters** (k)
- K-Medoids **doesn't rely on a specific distance** metric like Euclidean distance (used in K-Means). It can work with various distance metrics like Manhattan distance or cosine similarity
- **Non-Spherical Clusters**: K-Medoids can effectively handle clusters with irregular shapes (elongated, crescent-shaped, etc.) because it chooses actual data points as medoids.
- **Dealing with Outliers**: K-Medoids is known for its robustness against outliers. Unlike K-Means, which uses the mean (average) as the center of

a cluster, K-Medoids selects the medoid - the data point that minimizes the total distance to all other points in the cluster.

d. Avoid K-Medoid When

- **Large Datasets and Performance**: K-Medoids can be computationally slower than K-Means, especially for very large datasets.
- The process of calculating distances between all data points to find the medoid can become time-consuming.
- If Euclidean distance is a good fit for your data, and you don't have concerns about outliers
- **Focus on Simplicity**: If you're new to clustering or need a very fast initial clustering solution

e. Strengths and Weakness of K-Means

i. *Strengths*:

1. **Robust to Outliers**: K-Medoids chooses medoids (data points themselves) as cluster centers. This makes it less susceptible to extreme values that can skew the center in K-Means.
2. **Flexible Distance Metrics**: K-Medoids isn't limited to Euclidean distance like K-Means. It can work with various distance metrics like Manhattan distance or cosine similarity, making it adaptable to different data types.
3. **Effective for Non-Spherical Clusters**: K-Medoids excels at handling clusters with irregular shapes (elongated, crescent-shaped, etc.) because it uses actual data points as medoids.

ii. Weakness:

1. **Computational Cost**: K-Medoids can be slower than K-Means, especially for large datasets. Calculating distances between all data points to find the medoid can be time-consuming.
2. **Initialization Dependence**: The initial placement of medoids can impact the final results. Experimentation with different initialization techniques might be necessary.
3. **Implementation Complexity**: K-Medoids can be slightly more complex to implement compared to the simpler K-Means algorithm.

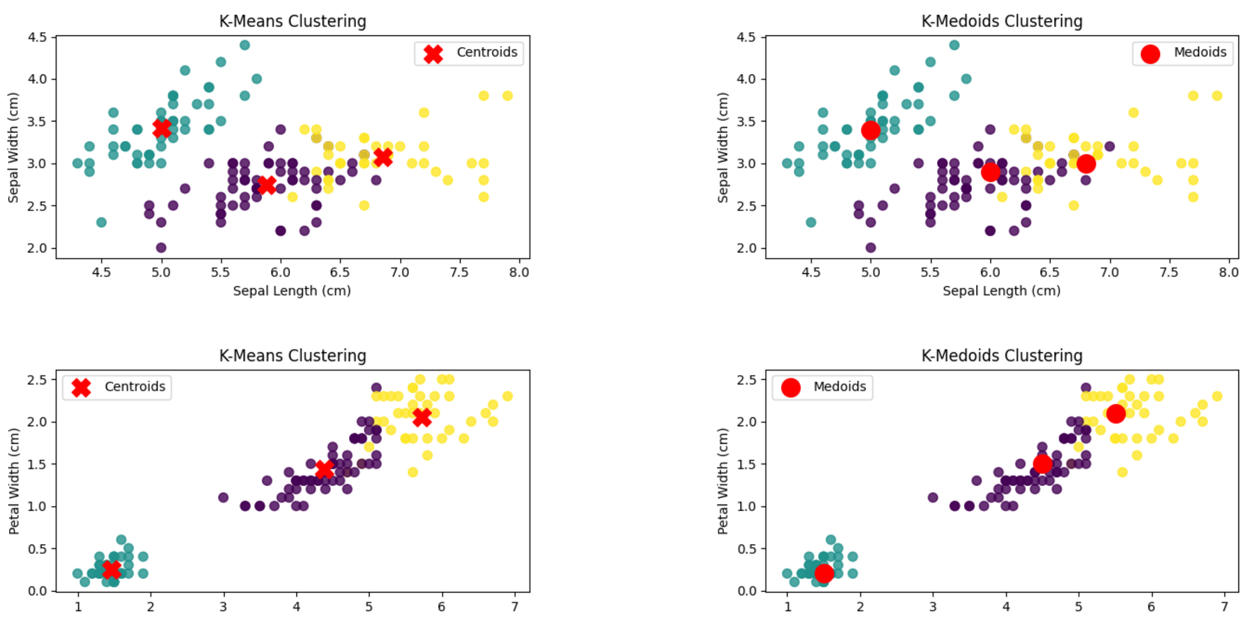
4. K- Means Vs K-Medoid Algorithm Results comparison

a. Choosing the Right Algorithm:

- For large datasets and well-separated spherical clusters, **K-Means** offers a fast and efficient solution.
- If you suspect outliers, non-spherical clusters, or need flexibility in distance metrics, **K-Medoids** is a better choice.
- Start with K-Means for its simplicity, especially if you're new to clustering. If limitations arise, consider K-Medoids.
- Experiment with both algorithms on your specific data to see which one yields better clustering results.

Output:

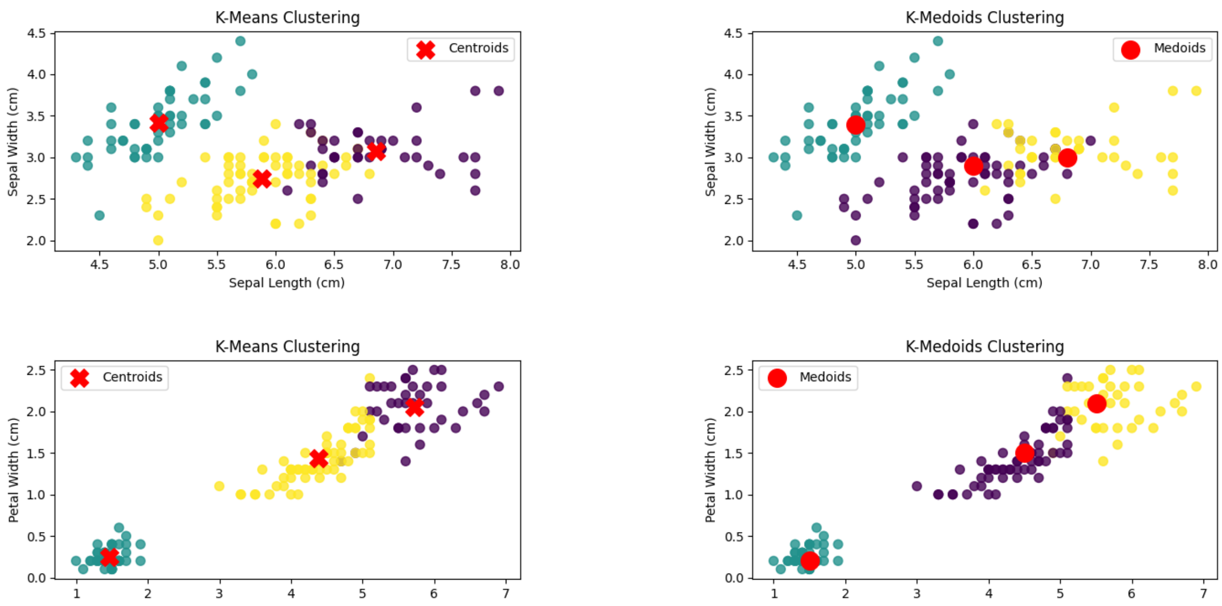
Sample 1:



- K-Means SSE: 78.94506582597731
- K-Medoids SSE: 84.67999999999999
- K-Means Cluster Counts: [61 50 39]
- K-Medoids Cluster Counts: [62 50 38]

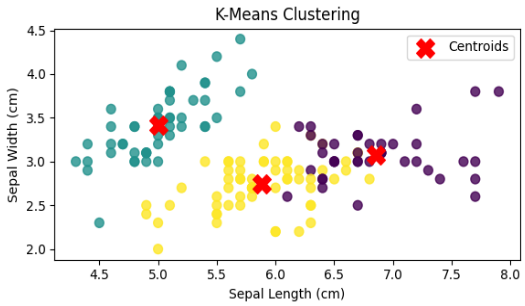
- K-Means Total Steps: 11
- K-Medoids Total Steps: 3
- K-Means Time Taken (milliseconds): 2.0
- K-Medoids Time Taken (milliseconds): 0.0

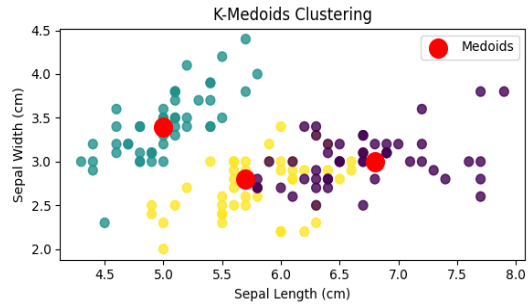
Sample 2:

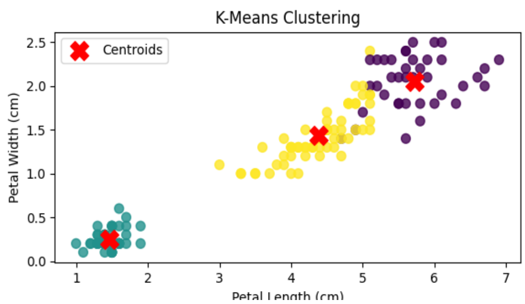


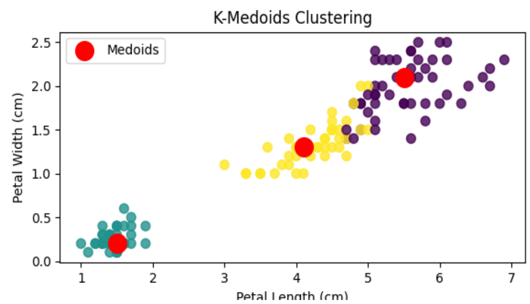
- K-Means SSE: 78.94506582597731
- K-Medoids SSE: 84.67999999999999
- K-Means Cluster Counts: [39 50 61]
- K-Medoids Cluster Counts: [62 50 38]
- K-Means Total Steps: 6
- K-Medoids Total Steps: 2
- K-Means Time Taken (milliseconds): 1.0
- K-Medoids Time Taken (milliseconds): 0.0

Sample 3:









- K-Means SSE: 78.94506582597731
- K-Medoids SSE: 85.17
- K-Means Cluster Counts: [39 50 61]
- K-Medoids Cluster Counts: [51 50 49]
- K-Means Total Steps: 11
- K-Medoids Total Steps: 2
- K-Means Time Taken (milliseconds): 0.99
- K-Medoids Time Taken (milliseconds): 0.0