Documentation for Clustering Algorithm Implementation

Aliaksei Zimnitski
FIIT STU

November 7, 2024

Contents

1 How to Run the Code

1. Running the Script in PyCharm

   To execute the `execute.sh` script within PyCharm, follow these steps:

   (a) Ensure that your PyCharm project is open.

   (b) At the bottom of the PyCharm window, click on the `Terminal` tab to open the integrated terminal.

   (c) Type the following command and press `Enter`:

   ```
       ./execute.sh
   ```

2. The script will ask which algorithm you want to run, and you can respond with your choice.

```
echo "Choose a clustering algorithm:"
echo "c - k-means with centroids"
echo "m - k-means with medoids"
echo "d - divisive clustering"

read -p "Enter 'c', 'm', or 'd': " choice

# Validate input
while [[ "$choice" != "c" && "$choice" != "m"
  && "$choice" != "d" ]]; do
    echo "Invalid input. Please enter 'c', 'm', or 'd'."
    read -p "Try again: " choice
done
```

3. Next, the script runs `main.py`, which contains the algorithm code, using `pypy` for faster execution.

```
pypy main.py "$choice"
```

4. Once `main.py` finishes, `draw.py` is launched with `python3` to visualize the clusters produced by `main.py`. `pypy` cannot be used here as it does not support `matplotlib`.

```
python3 draw.py
```

5. After closing the visualization, the script completes.

2 Main Program (main.py)

2.1 User Input and Point Generation

1. At the beginning, the program receives the user's algorithm choice.

2. It creates 20 unique points on a map using the function:

```
def generate_20_coordinates():
    i = 0
    while i < 20:
        x = random.randint(min_coord, max_coord)
        y = random.randint(min_coord, max_coord)
        if (x, y) not in exist_coord:
            exist_coord.append((x, y))
            i += 1
```

3. After generating 20 random points, an additional 40,000 points are generated as follows:

   - Randomly choose one of the previously created points in the 2D space.
   - If the point is too close to the boundary, adjust the interval as specified in the next steps.
   - Generate random offsets `X_offset` and `Y_offset` within -100 to +100.
   - Add the new point to the 2D space, shifted by these offsets.

```
i = 0
while i < 40000:
    parent = random.choice(exist_coord)

    max_x_offset = 100
    max_y_offset = 100
    min_x_offset = -100
    min_y_offset = -100

    if parent[0] + max_x_offset > max_coord:
        max_x_offset = max_coord - parent[0]
    if parent[1] + max_y_offset > max_coord:
        max_y_offset = max_coord - parent[1]
    if parent[0] + min_x_offset < min_coord:
        min_x_offset = min_coord - parent[0]
    if parent[1] + min_y_offset < min_coord:
        min_y_offset = min_coord - parent[1]

    x_offset = random.randint(min_x_offset, max_x_offset)
    y_offset = random.randint(min_y_offset, max_y_offset)

    new_x = parent[0] + x_offset
    new_y = parent[1] + y_offset

    if (new_x, new_y) not in exist_coord:
        exist_coord.append((new_x, new_y))
        i += 1
```

4. Next, the selected algorithm is executed:

```
if choice == 'c':
    k = 20
    clusters = k_means(exist_coord, exist_coord[:k], k, 1)
elif choice == 'm':
    k = 20
    clusters = k_means(exist_coord, exist_coord[:k], k, 0)
else:
    k = 20
    clusters = divisive_clustering(exist_coord, k, 1)
```

5. The clusters are saved in JSON format for use in `draw.py`:

```
def save_clusters_to_file(clusters, filename="clusters.json"):
    with open(filename, 'w') as file:
        json.dump(clusters, file)
```

3 Explanation of Algorithms

3.1 K-means Algorithm

The K-means algorithm operates as follows:

1. The function `k_means` receives `points` (all points), `centers` (initial centers), `k` (number of clusters), and `which_kmeans` (1 for centroids, 0 for medoids).

```
def k_means(points, centers, k, which_kmeans):
```

2. A loop is started.

3. A `clusters` array is created to store each cluster's points and center.

```
clusters = [[[], centers[i]] for i in range(k)]
```

4. All points are assigned to the nearest center.

```
for point in points:
    distances = [euclidean_distance(point, center)
                                        for center in centers]
    closer_center = distances.index(min(distances))
    clusters[closer_center][0].append(point)
```

5. Current centers are saved.

```
last_centers = centers[:]
```

6. Centers are recalculated using `calculate_centroid` or `calculate_medoid`.

```
for i in range(k):
    if which_kmeans == 1:
      clusters[i][1] = calculate_centroid(clusters[i][0])
    else:
      clusters[i][1] = calculate_medoid(clusters[i][0])
    centers[i] = clusters[i][1]
```

(a) `calculate_centroid` finds the average point in a cluster.

```
def calculate_centroid(points):
    sum_x = sum(point[0] for point in points)
    sum_y = sum(point[1] for point in points)
    centroid_x = sum_x / len(points)
    centroid_y = sum_y / len(points)
    return (int(centroid_x), int(centroid_y))
```

(b) `calculate_medoid` finds the point with the smallest average distance to all other points.

```
def calculate_medoid(points):
    medoids = []

    for current_medoid in points:
        cur_sum_path = sum(euclidean_distance(current_medoid,
                                point2) for point2 in points)
        medoids.append((cur_sum_path, current_medoid))

    return min(medoids)[1]
```

7. The loop ends when centers stabilize.

```
if last_centers == centers:
    break
```

3.2 Divisive Clustering

1. The function is recursive.

2. Initially, it takes all points, the required number of clusters, and the current number of clusters (starting from 1).

```
def divisive_clustering(clusters, k, k_clusters):
```

3. If only one cluster exists, we work with it; otherwise, we select the cluster with the largest spread and remove it from the list to split it into two.

```
if k_clusters == 1:
    cluster = clusters
else:
    max_clusters_index = find_biggest_cluster(clusters)
    cluster = clusters.pop(max_clusters_index)[0]
```

4. Finding the cluster with the largest spread: For each cluster:

   - Calculate the distance of each point to the cluster center.
   - Sum the squares of these distances.
   - Select the cluster with the largest value.

```
def find_largest_spread(clusters):
    average_distance = []
    for cluster in clusters:
        points = cluster[0]
        centroid = cluster[1]

     distances = sum(euclidean_distance(point, centroid) ** 2
                                             for point in points)
     spreads.append(distances)

    return spreads.index(max(spreads))
```

5. Determine initial centroids for `k-means` by selecting the two most distant points.

```
res = (0, [(0, 0), (0, 0)])
for point in cluster:
    for point2 in cluster:
        dist = euclidean_distance(point, point2)
        if dist > res[0]:
            res = (dist, [point, point2])

centroids = res[1]
```

6. If only one cluster is present, use `k_means` to split it; otherwise, add the new clusters to the existing list.

```
if k_clusters == 1:
    clusters = k_means(cluster, centroids, 2, 1)
else:
    new_clusters = k_means(cluster, centroids, 2, 1)
    clusters.append(new_clusters[0])
    clusters.append(new_clusters[1])
```

7. Increment the current cluster count.

```
k_clusters += 1
```

8. If the required number of clusters is reached, return them. Otherwise, recursively call the function, passing the list of clusters and centroids.

```
    if k_clusters == k:
        return clusters
    else:
        return divisive_clustering(clusters, k, k_clusters)
```

4 Visualization Program (draw.py)

This program calculates the average distance from all points to the centroid of each cluster, displaying the value next to each centroid on the visualization. This helps verify clustering success.

5 Examples of Clustering Results

To illustrate the effectiveness of each clustering algorithm, here are three example visualizations for each method: k-means with centroids, k-means with medoids, and divisive clustering. In each plot, the average distance of points to their cluster centroid is labeled, providing a quantitative assessment of clustering accuracy.

5.1 K-means with Centroids

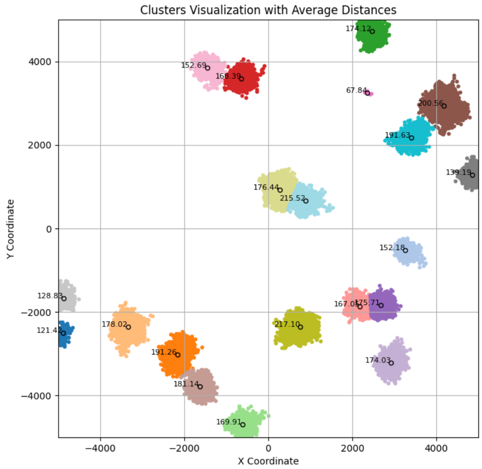


Figure 1: K-means with centroids - Example 1

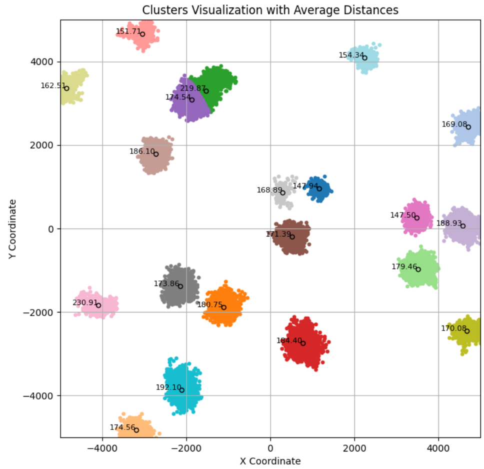


Figure 2: K-means with centroids - Example 2

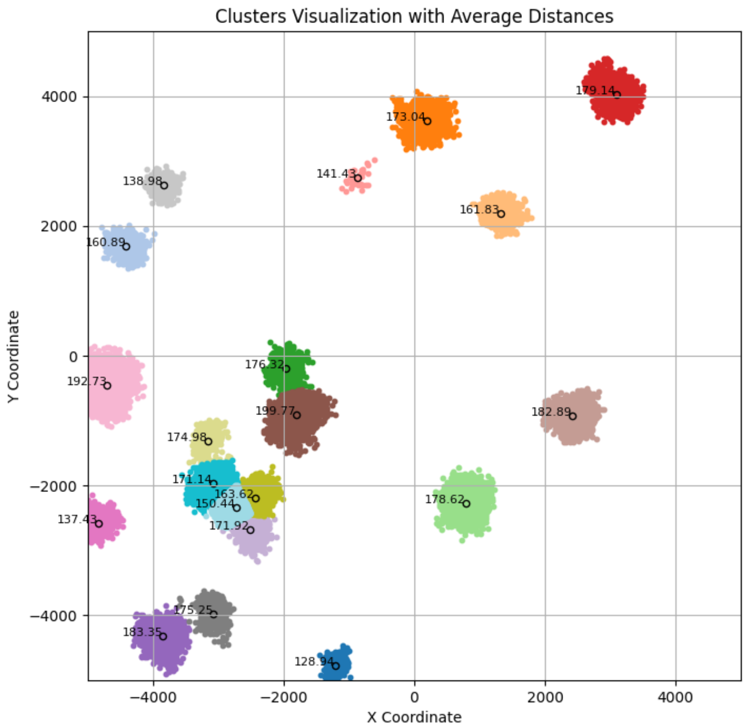


Figure 3: K-means with centroids - Example 3

5.2 K-means with Medoids

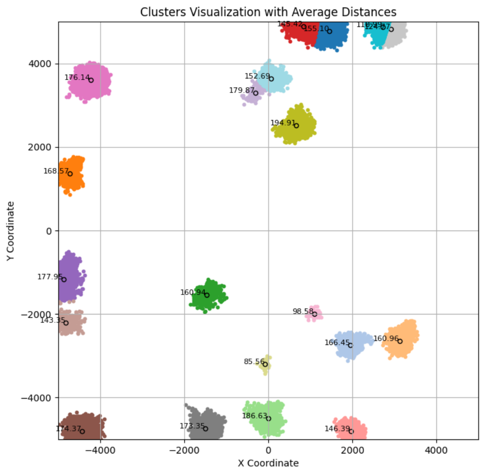


Figure 4: K-means with medoids - Example 1

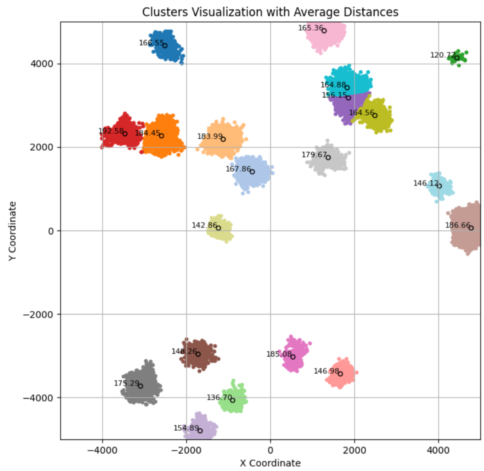


Figure 5: K-means with medoids - Example 2

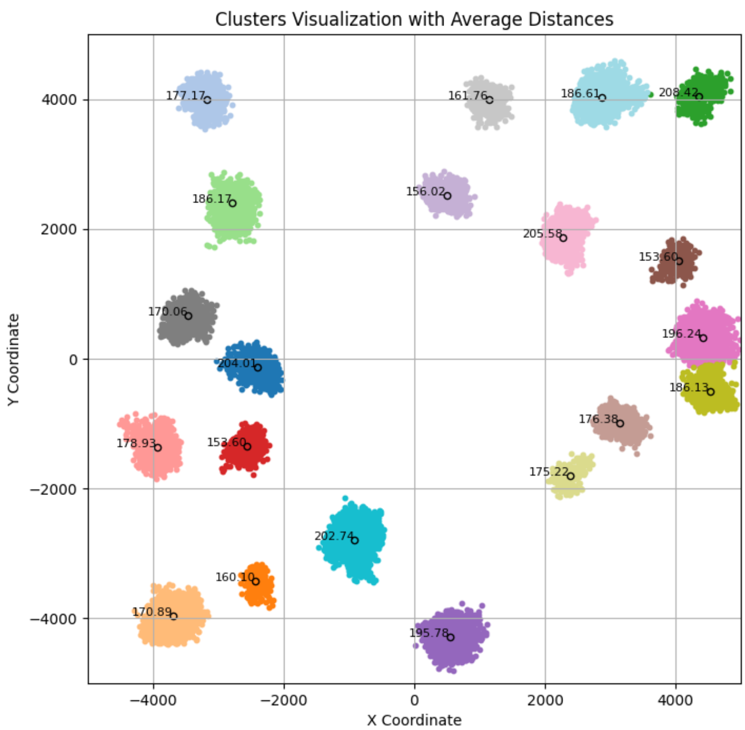


Figure 6: K-means with medoids - Example 3

5.3 Divisive Clustering

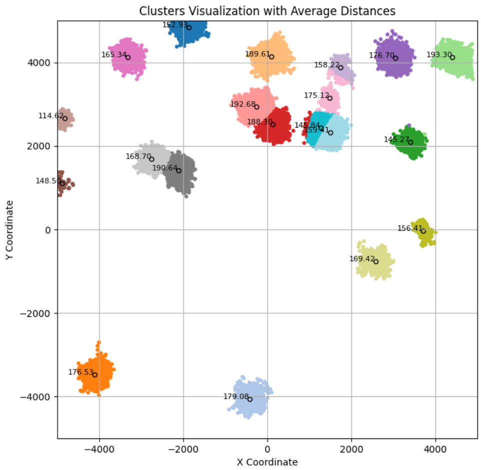


Figure 7: Divisive clustering - Example 1

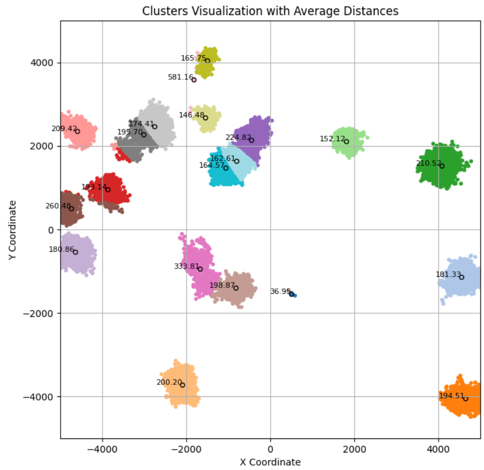


Figure 8: Divisive clustering - Example 2

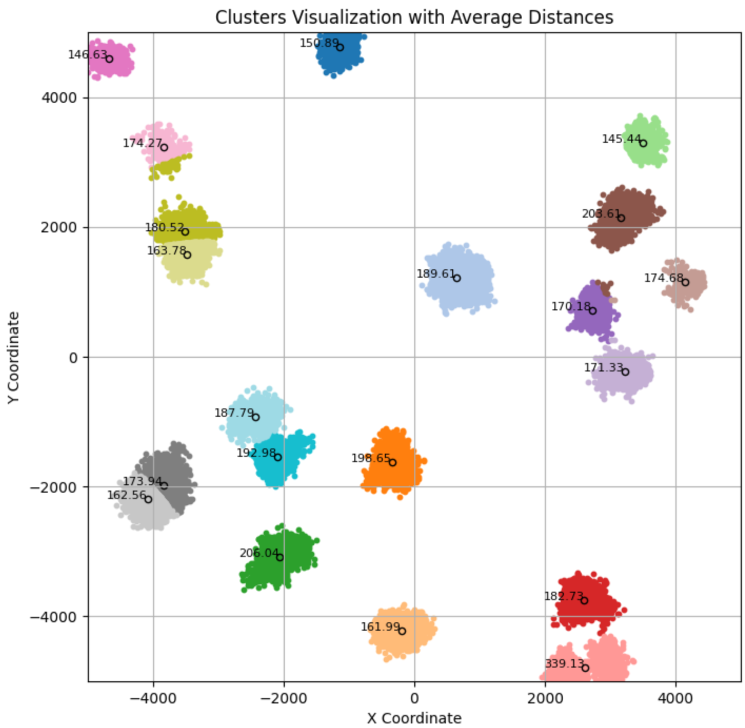
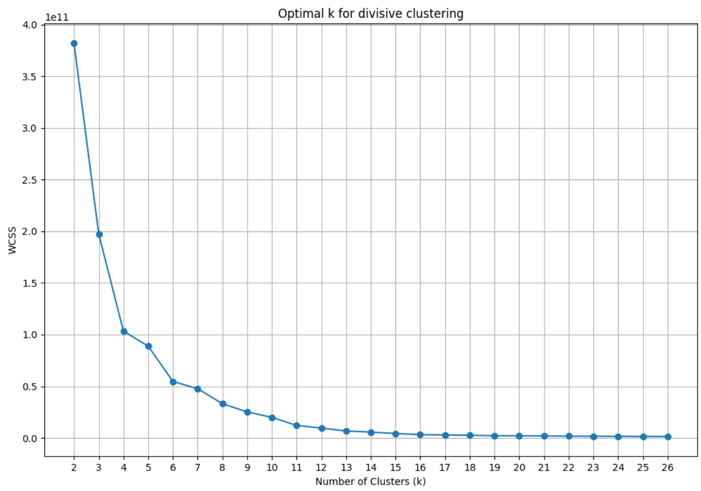


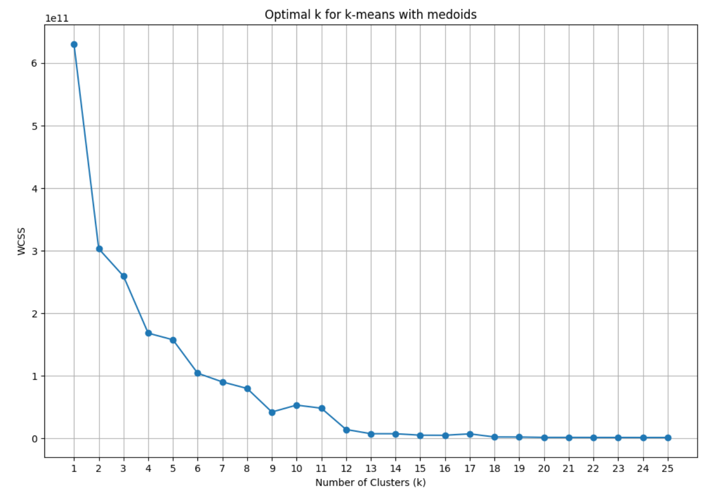
Figure 9: Divisive clustering - Example 3

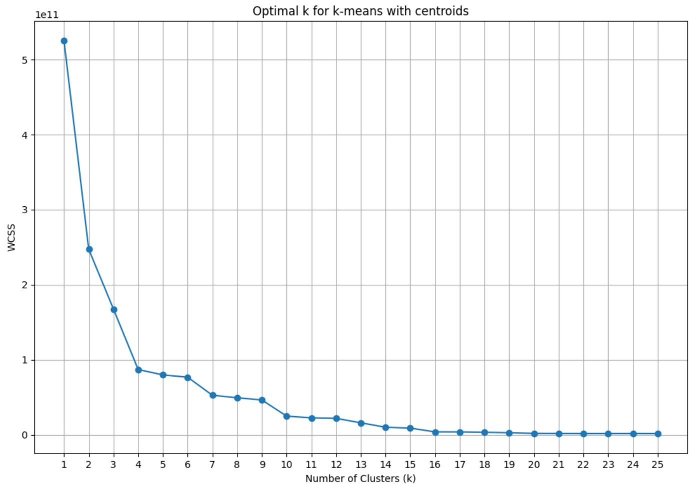
6 Elbow method

The optimal number of clusters for this dataset was determined using the elbow method. This method involves plotting the within-cluster sum of squares (WCSS) against the number of clusters k. The "elbow" point on this plot indicates a diminishing return in WCSS as k increases, suggesting the optimal number of clusters.

The following graphs illustrate the elbow method for different clustering algorithms:







Based on these plots, the optimal number of clusters was chosen as 20 for each algorithm. This selection ensures that clustering is effective without introducing excessive complexity.

7 Conclusion

In conclusion, all the algorithms required 20 clusters to ensure successful clustering. However, when comparing them in terms of execution time, k-means with centroids performed approximately twice as fast as the other algorithms. Unlike k-means with medoids, which needs to iterate over all points in quadratic time, and the divisive method, which searches for centroids iteratively, k-means with centroids directly identifies the necessary centroids more efficiently. Therefore, k-means with centroids is the most suitable choice for this task.