Clustering Comparison: K-Medoids vs DBSCAN

Introduction

This document compares two popular clustering algorithms: **K-Medoids** and **DBSCAN**.

- **K-Medoids:** This method uses actual data points as cluster centers (called medoids). It's better at handling outliers and works well for round-shaped clusters.
- **DBSCAN:** This algorithm finds clusters based on how dense the data points are. It can spot clusters of any shape and deals well with noise. You don't need to tell it how many clusters to look for.

Code Examples

K-Medoids

Following is an implementation of K-Medoids clustering algorithm. For convenance data points are taken form .csv file.

```
import pandas as pd
import numpy as np
import random

# Set the norm type for distance calculations
NORM_TYPE = 'L2'  # Options: 'L1', 'L2', 'L∞'

# Load and clean the data from a CSV file.
def load_clustering_data(file_path: str):
    # Convert all columns to numeric values and drop rows with missing values
    return pd.read_csv(file_path).apply(pd.to_numeric, errors='coerce').dropna()

# Compute distance between two points based on the selected norm type.
def compute_distance(point1, point2):
    if NORM_TYPE == 'L1':
        return np.sum(np.abs(point1 - point2))  # Manhattan distance (L1 norm)
    elif NORM_TYPE == 'L2':
        return np.linalg.norm(point1 - point2)  # Euclidean distance (L2 norm)
    elif NORM_TYPE == 'L∞':
        return np.max(np.abs(point1 - point2))  # Chebyshev distance (L∞ norm)

# Randomly select initial medoids from the dataset.
def generate_initial_random_medoids(data: pd.DataFrame, k: int):
    # Select k unique random indices from the data
    random_indices = random.sample(range(len(data)), k)
    # Extract the data points corresponding to the selected indices
    return [data.iloc[idx].values.tolist() for idx in random_indices]

# Assign each data point to the nearest medoid.
def assign_clusters(data: pd.DataFrame, medoids: list):
    cluster_assignment = []
    for _, row in data.iterrows():
        # Compute distances from the point to each medoid
        distances = [compute_distance(row.values, np.array(medoid)) for medoid in medoids]
        # Assign the point to the closest medoid
        closest_medoid = np.argmin(distances)
        cluster_assignment.append(int(closest_medoid))
    return cluster_assignment

# Calculate new medoids for each cluster based on the assigned points.
def calculate_new_medoids(data: pd.DataFrame, cluster_assignment: list, k: int):
    new_medoids = []
    for cluster_idx in range(k):
        # Get all points assigned to the current cluster
        cluster_points = data[np.array(cluster_assignment) == cluster_idx]
        if len(cluster_points) > 0:
            # Compute the total distance between each point and all other points in the clust
er
            distances = cluster_points.apply(lambda point: np.sum(
                [compute_distance(point.values, other_point)
                 for other_point in cluster_points.values]), axis=1)
            # Find the point with the minimum total distance to others (the new medoid)
            medoid_idx = distances.idxmin()
            new_medoids.append(cluster_points.loc[medoid_idx].values.tolist())
    return new_medoids

# Main K-Medoids clustering function.
def cluster_by_k_medoids(file_path: str, k: int, maxIter: int):
    # Load the dataset
    data = load_clustering_data(file_path)
    # Generate initial random medoids
    medoids = generate_initial_random_medoids(data, k)
    # Assign points to initial clusters based on the medoids
    cluster_assignment = assign_clusters(data, medoids)

    for _ in range(maxIter):
        # Recalculate medoids for each cluster
        medoids = calculate_new_medoids(data, cluster_assignment, k)
        # Reassign points to clusters based on the new medoids
        new_cluster_assignment = assign_clusters(data, medoids)
        # If cluster assignments don't change, stop iterating (convergence)
        if new_cluster_assignment == cluster_assignment:
            break
        # Update the cluster assignments
        cluster_assignment = new_cluster_assignment

    return medoids, cluster_assignment


if __name__ == "__main__":
    file_path = 'data.csv'  # Path to the dataset
    k = 3  # Number of clusters
    maxIter = 100  # Maximum number of iterations

    # Execute K-Medoids clustering
    final_medoids, final_assignment = cluster_by_k_medoids(file_path, k, maxIter)
    print("Final Medoids:", final_medoids)
    print("Final Cluster Assignment:", final_assignment)
```

DBSCAN

Density-Based Spatial Clustering of Applications with Noise is a clustering algorithm based on intuitive notion of clusters and noise. The idea is that each cluster point neighborhood has to contain minimum number of points.

```
import pandas as pd
import numpy as np

# Set the norm type for distance calculation
NORM_TYPE = 'L2'  # Options: 'L1', 'L2', 'L∞'

# Load and preprocess the data from a CSV file
def load_clustering_data(file_path: str):
    # Load the dataset, convert columns to numeric, and drop rows with missing values
    return pd.read_csv(file_path).apply(pd.to_numeric, errors='coerce').dropna()

# Calculate the norm (distance) between two points based on the selected norm type
def calculate_norm(point1, point2):
    if NORM_TYPE == 'L1':
        return np.sum(np.abs(np.array(point1) - np.array(point2)))  # Manhattan distance
    elif NORM_TYPE == 'L2':
        return np.linalg.norm(np.array(point1) - np.array(point2))  # Euclidean distance
    elif NORM_TYPE == 'L∞':
        return np.max(np.abs(np.array(point1) - np.array(point2)))  # Chebyshev distance

# Find all neighbors of a point that are within the given epsilon (eps) radius
def find_neighbors(data: pd.DataFrame, point_index: int, eps: float):
    neighbors = []
    # Loop through all points and calculate distance to the point_index
    for idx in range(len(data)):
        if calculate_norm(data.iloc[point_index], data.iloc[idx]) <= eps:
            neighbors.append(idx)  # Append if within eps distance
    return neighbors

# Expand the cluster by visiting all density-reachable points
def expand_cluster(data: pd.DataFrame, cluster_assignment: list, point_index: int, neighbors:
```

```
list, cluster_id: int,
                   eps: float, min_pts: int):
    # Assign the current point to the current cluster
    cluster_assignment[point_index] = cluster_id
    i = 0
    # Expand by visiting all neighbors
    while i < len(neighbors):
        neighbor_idx = neighbors[i]
        if cluster_assignment[neighbor_idx] == -1:  # If previously marked as noise, add to c
luster
            cluster_assignment[neighbor_idx] = cluster_id
        elif cluster_assignment[neighbor_idx] == 0:  # If it's an unvisited point
            cluster_assignment[neighbor_idx] = cluster_id
            # Find neighbors of the current neighbor
            new_neighbors = find_neighbors(data, neighbor_idx, eps)
            # If new_neighbors are dense enough, include them in the expansion
            if len(new_neighbors) >= min_pts:
                neighbors.extend(new_neighbors)
        i += 1

# DBSCAN (Density-Based Spatial Clustering of Applications with Noise) algorithm
def cluster_by_dbscan(file_path: str, eps: float, min_samples: int):
    # Load the data
    data = load_clustering_data(file_path)
    # Initialize all points as unvisited (0)
    cluster_labels = [0] * len(data)  # 0 means unvisited
    cluster_id = 0

    # Iterate over each point in the dataset
    for point_idx in range(len(data)):
        if cluster_labels[point_idx] != 0:  # Skip already visited points
            contin
        # Find neighbors within epsilon distance
        neighbors = find_neighbors(data, point_idx, eps)
        if len(neighbors) < min_samples:  # Not enough neighbors to form a cluster, mark as n
oise (-1)
            cluster_labels[point_idx] = -1
        else:
            # Start a new cluster if neighbors meet the min_samples condition
            cluster_id += 1  # Increment cluster ID
            expand_cluster(data, cluster_labels, point_idx, neighbors, cluster_id, eps, min_s
amples)

    return cluster_labels  # Return the cluster labels for all points


if __name__ == '__main__':
    file_path = 'data.csv'  # Path to the dataset
    eps = 0.25  # Radius that defines the neighborhood
    min_pts = 10  # Minimum number of points to form a dense region (cluster)

    # Run DBSCAN clustering
    cluster_labels = cluster_by_dbscan(file_path, eps, min_pts)
    print(cluster_labels)  # Output the cluster labels
```

Comparison

Evaluation Criteria

To compare the clustering algorithms, we'll use both visual and numerical evaluations. The numerical evaluations will be based on two widely-used metrics:

- Davies-Bouldin Index: Measures the average similarity between each cluster and its most similar cluster. Lower values indicate better clustering.
- Calinski-Harabasz Score: Evaluates the cluster validity based on the average between- and within-cluster sum of squares. Higher scores indicate better-defined clusters.

The table below provides a general interpretation guide for these metrics:

Evaluation Type	Very Good	Good	Fair	Poor
Davies-Bouldin Index	0.0 to 0.3	0.3 to 0.8	0.8 to 2.0	> 2.0
Calinski-Harabasz Score	100+	100 to 50	30 to 10	10 to 1

This evaluation metrics are to be found under python `sklearn.metrics` library. As for visualization I am using `matplotlib` library.

Both Algorithms Perform Accurately

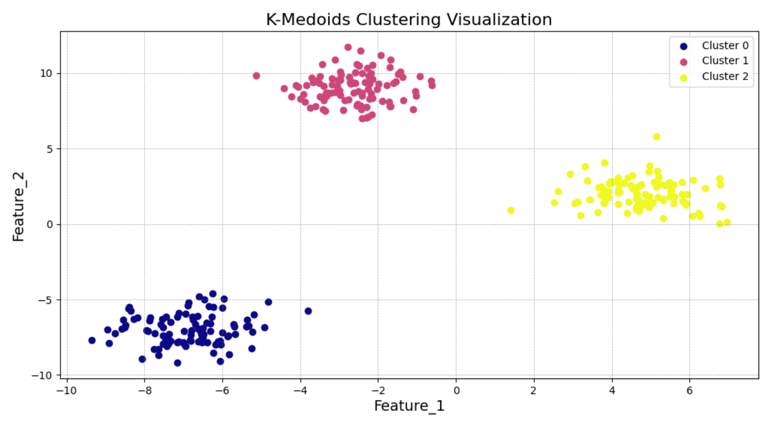
For this example, let's examine well-separated clusters of three distinct groups.

blob_dataset.csv

K-Medoids

`k = 3`

`max_iter = 100`



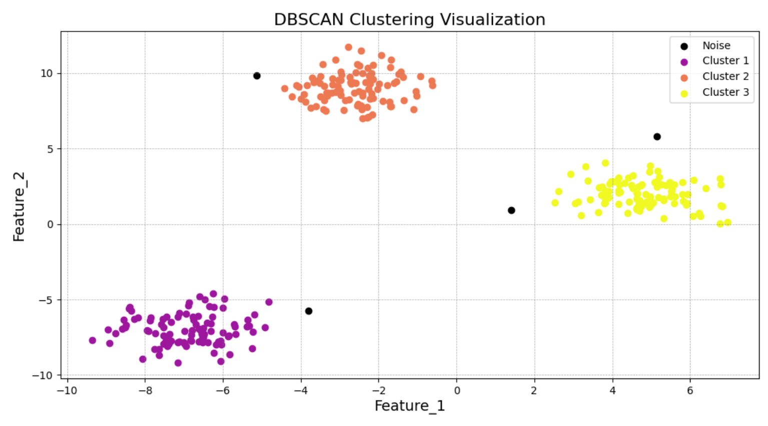
Evaluation Type	Score	Rating
Davies-Bouldin	0.207	Very Good
Calinski-Harabasz	4196.295	Very Good

This clustering represents the best possible outcome, as verified visually and supported by the numerical results.

DBSCAN

`eps = 1.1`

`min_pts = 5`



Evaluation Type	Score	Rating
Davies-Bouldin	0.212	Very Good
Calinski-Harabasz	3678.678	Very Good

The scores are nearly identical to those of K-medoids, and DBSCAN also identified outliers as noise.

DBSCAN Performs Better

DBSCAN is great for identifying clusters with irregular shapes and varying densities, like concentric circles. K-medoids, on the other hand, works best with compact, spherical clusters of similar size. Here are two examples:

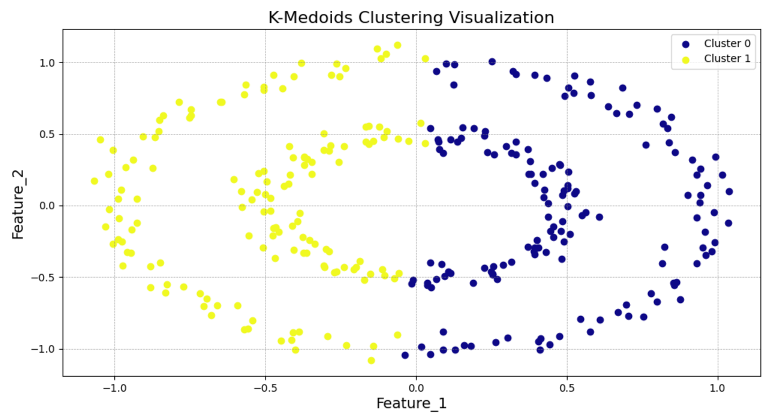
concentric_circles.csv

moons.csv

K-Medoids

k = 2

max_iter = 100



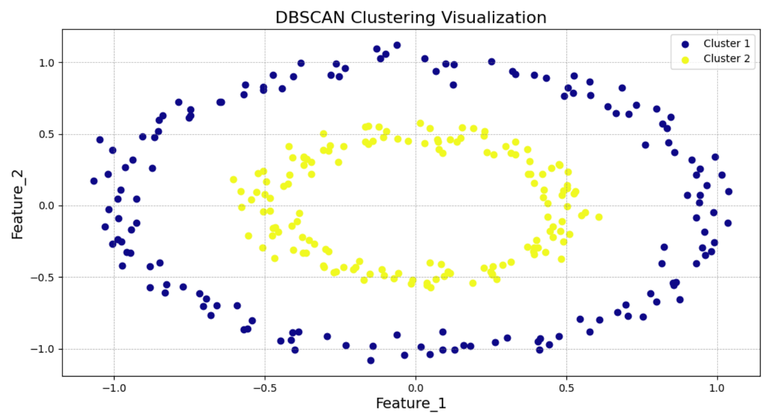
Evaluation Type	Score	Rating
Davies-Bouldin	1.574	Fair
Calinski-Harabasz	45.562	Good

As expected, k-medoids clusters data based on centroids, thus splitting data points in half.

DBSCAN

eps = 0.1

min_pts = 5



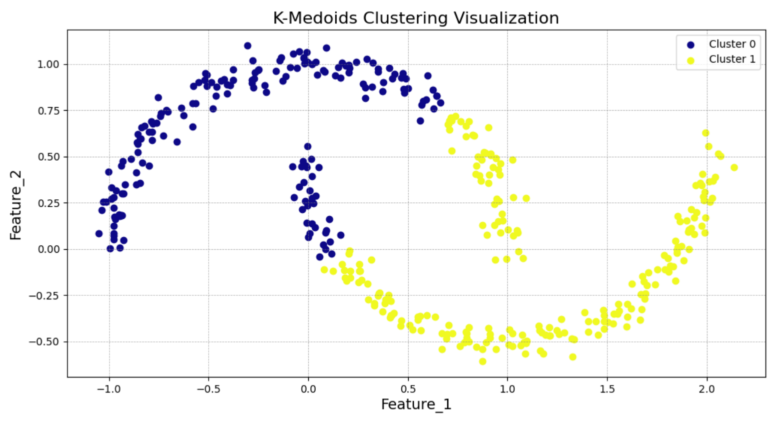
Evaluation Type	Score	Rating
Davies-Bouldin	0.009	Very Good
Calinski-Harabasz	175.144	Very Good

Since DBSCAN is good for clustering irregular shapes. It clustered this data well. One ring for each cluster.

K-Medoids

k = 2

max_iter = 100



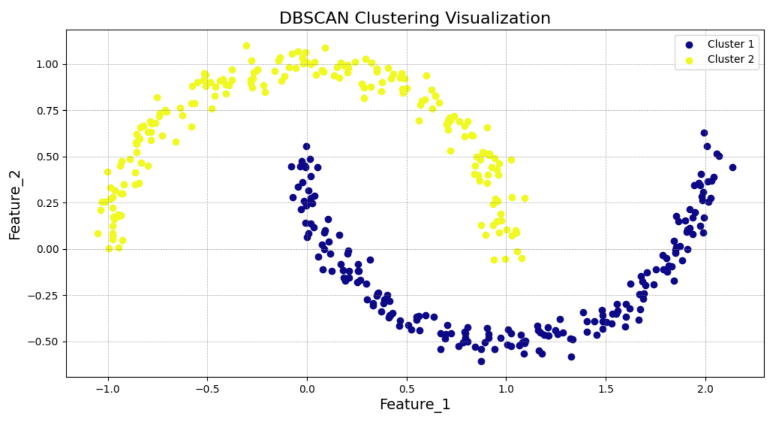
Evaluation Type	Score	Rating
Davies-Bouldin	0.875	Fair
Calinski-Harabasz	57.665	Good

As expected, k-medoids clusters data based on centroids, thus drawing middle line between two moons.

DBSCAN

eps = 0.1

min_pts = 5



Evaluation Type	Score	Rating
Davies-Bouldin	0.166	Very Good
Calinski-Harabasz	258.025	Very Good

Since DBSCAN is good for clustering irregular shapes. It clustered this data well. One moon for each cluster.

Results

K-Medoids and DBSCAN both performed well on datasets with well-separated blobs, accurately identifying clusters. However, DBSCAN struggled with scattered plots where clusters were not distinctly separated, failing to form coherent groups due to varying densities. In contrast, K-Medoids managed to assign points to clusters in these situations. Conversely, DBSCAN excelled with irregularly shaped clusters, leveraging its density-based approach, while K-Medoids may struggle since it relies on defined centroids. Overall, K-Medoids is better for well-separated clusters, while DBSCAN is more suited for irregular shapes, though it faces challenges with scattered data with its density difference.

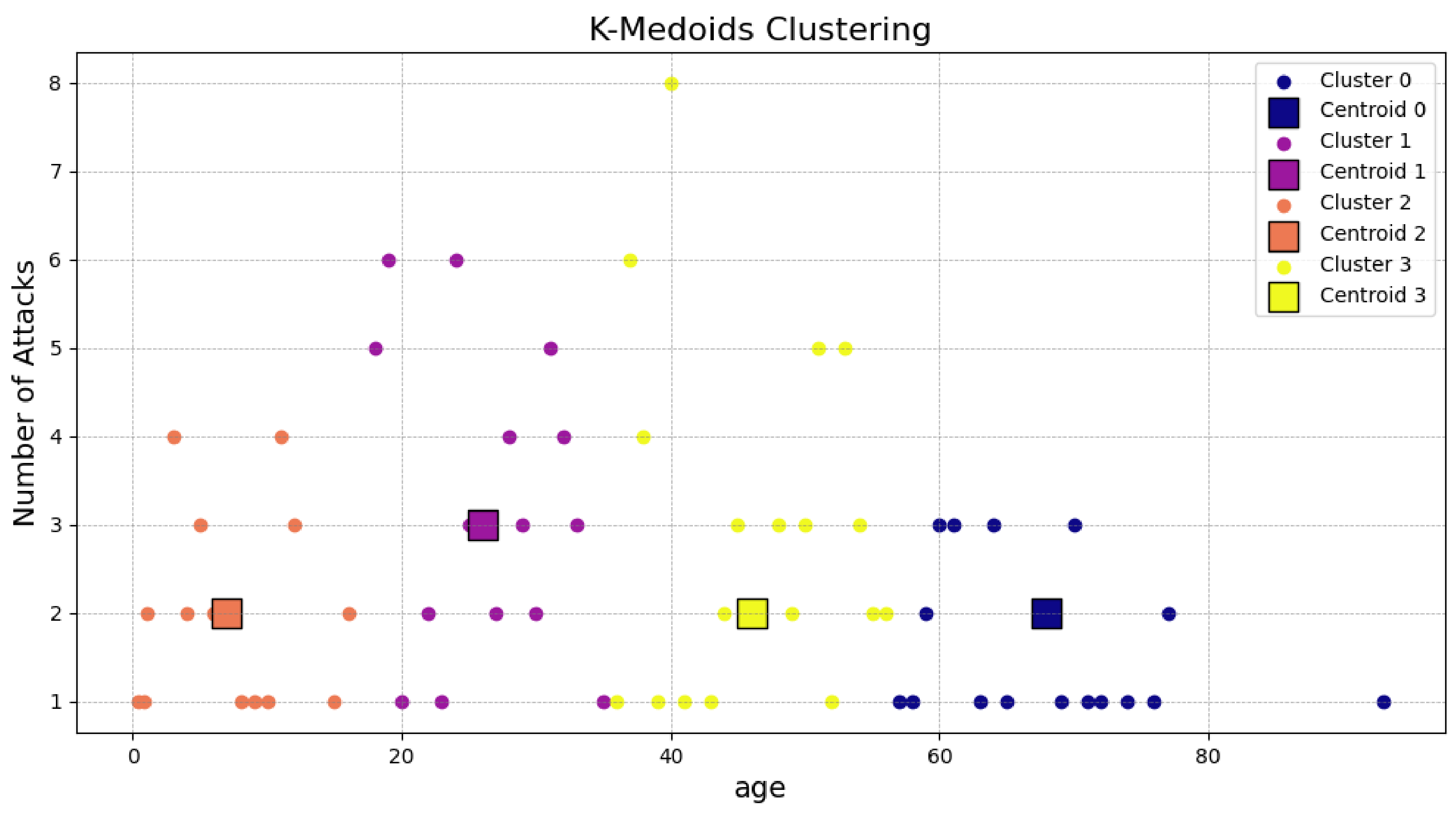
Real World Data

For the real world data example I took recorded fatal bear accidents in USA since 1900 to present day, source: dataset. I grouped entries according to age groups, giving me correlation between ages and number of attacks.

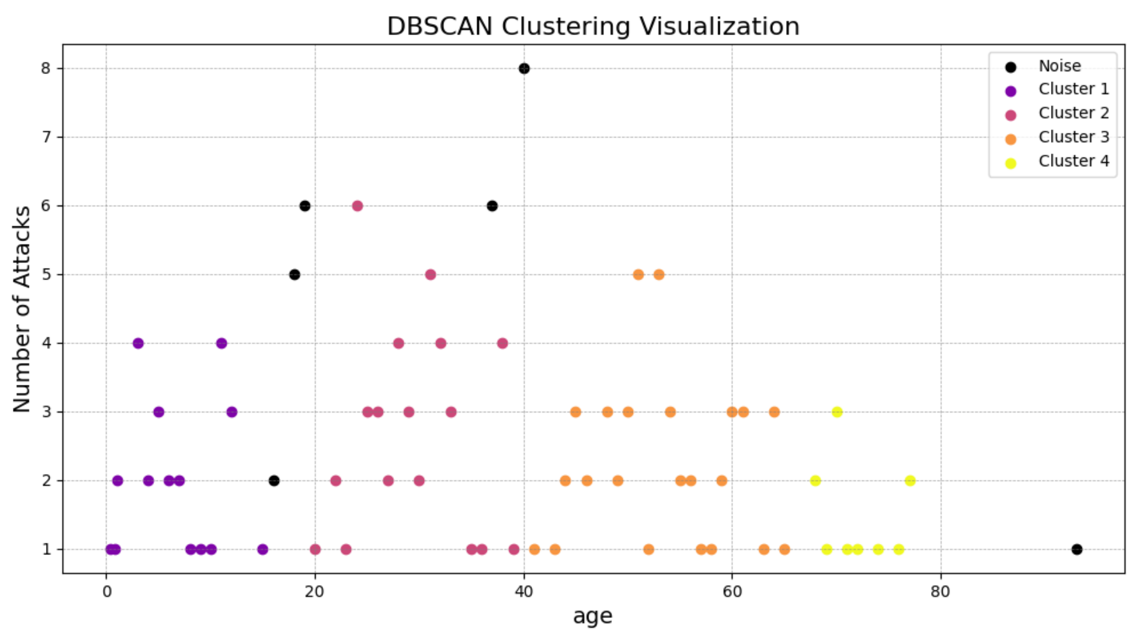
Processed_BearAttacks.csv

Here is how processed data looks like clustered via K-Medoids and DBSCAN.

K-Medoids



DBSCAN



DBSCAN actually consistently produced 4 clusters after changing its inputs couple of times and finalized on `eps = 3.5` and `min_pts = 5`. So I decided to keep K-Medoids cluster count to 4, that is `k = 4` and `max_iter = 100`. Based on results clustered data between these two algorithms is somewhat similar, with exception of couple points being different.

There are 4 clear clusters.

- Ages 0-10 years: With average beat attacks of 2.5.
- Ages 20-40 years: With average beat attacks of 3.
- Ages 40-60 years: With average beat attacks of 3.
- Ages 65-80 years: With average beat attacks of 1.

Since both algorithms led to similar clusters, it suggests that the structure of the data was naturally divided into distinct age groups, making it easier for both DBSCAN and K-Medoids to perform in this case.

The clustering results suggest that **age is a significant factor** in bear attack frequency. Specifically, younger individuals (0-10 years) and older individuals (65-80 years) experience fewer attacks compared to middle-aged individuals (20-60 years). This pattern could be linked to outdoor activity levels. **Middle-aged individuals**, who are typically more active in outdoor activities such as hiking, camping, and exploring wilderness areas, are likely to have more frequent encounters with bears. In contrast, **younger children and elderly adults** may engage less in these activities, resulting in fewer bear encounters. This indicates that activity levels associated with different age groups may play a critical role in bear attack risks.