Slovenská Technická Univerzita Fakulta Informatiky a Informačných technológií

Clustering

Umelá Inteligencia, Zadanie 3

Norbert Matuška
12-8-2023

Contents

Zadanie 1

Riešenie 2

- Generovanie počiatočných bodov 2
- Pomocné funkcie 2
  - Evaluácia úspešnosti algoritmov 2
  - Vizualizácia 3
- Algoritmy 4
  - K-means kde centroid je stred 4
  - K-means kde medoid je stred 4
  - Divízivné zhlukovanie 5

Testovanie 7

- 400 bodov 7
  - K-means centroid 7
  - Divisive clustering 7
  - K-means medoid 8
- 4000 bodov 8
  - K-means centroid 8
  - Divisive clustering 9
  - K-means medoid 9
- 40000 bodov 10
  - K-means centroid 10
  - Divisive clustering 10
  - K-means medoid 11

Záver 12

xmatuskan@stuba.sk 110849

Zadanie

Máme 2D priestor, ktorý má rozmery X a Y, v intervaloch od -5000 do +5000. Tento 2D priestor vyplňte 20 bodmi, pričom každý bod má náhodne zvolenú polohu pomocou súradníc X a Y. Každý bod má unikátne súradnice (t.j. nemalo by byť viacej bodov na presne tom istom mieste).

Po vygenerovaní 20 náhodných bodov vygenerujte ďalších 40000 bodov, avšak tieto body nebudú generované úplne náhodne, ale nasledovným spôsobom:

1. Náhodne vyberte jeden zo **všetkých doteraz vytvorených** bodov v 2D priestore. (nie len z prvých 20)
   Ak je bod príliš blízko okraju, tak zredukujete príslušný interval, uvedený v nasledujúcich dvoch krokoch.
2. Vygenerujte náhodné číslo X_offset v intervale od -100 do +100
3. Vygenerujte náhodné číslo Y_offset v intervale od -100 do +100
4. Pridajte nový bod do 2D priestoru, ktorý bude mať súradnice ako náhodne vybraný bod v kroku 1, pričom tieto súradnice budú posunuté o X_offset a Y_offset

Vašou úlohou je naprogramovať zhlukovač pre 2D priestor, ktorý zanalyzuje 2D priestor so všetkými jeho bodmi a rozdelí tento priestor na k zhlukov (klastrov). Implementujte rôzne verzie zhlukovača, konkrétne týmito algoritmami:

- k-means, kde stred je centroid
- k-means, kde stred je medoid
- divízne zhlukovanie, kde stred je centroid

Vyhodnocujte úspešnosť/chybovosť vášho zhlukovača. Za úspešný zhlukovač považujeme taký, v ktorom **žiaden klaster nemá priemernú vzdialenosť bodov od stredu viac ako 500**.

Vizualizácia: pre každý z týchto experimentov vykreslite výslednú 2D plochu tak, že označkujete (napr. vyfarbíte, očíslujete, zakrúžkujete) výsledné klastre.

Riešenie

Generovanie počiatočných bodov

Prvých 20 bodov generujem jednoduchou funkciou cez knižnicu random a pridávam vygenerované čísla do setu. Set používam z toho dôvodu, že sa v ňom nemôžu opakovať hodnoty. Funkcia vracia naspäť už pole a nie set.

```
def generate_starting_points(num_points=20, x_range=(-5000, 5000), y_range=(-5000, 5000)):
    starting_points = set()  # set, so the points dont repeat

    while len(starting_points) < num_points:
        x = random.randint(*x_range)
        y = random.randint(*y_range)
        starting_points.add((x, y))

    return starting_points
```

Následne generujem generujem ďalšie body pomocou funkcie **generate_offset_points()** kde zistím dolné a horné hranice pomocou max() a min() a medzi nimi vygenerujem náhodné číslo.

```
def generate_offset_points(base_points, num_of_new_points=40000, x_range=(-5000, 5000), y_range=(-5000, 5000)):
    all_points = set(base_points)
    offset_range = (-100, 100)

    while len(all_points) < num_of_new_points + len(base_points):
        base_x, base_y = random.choice(list(all_points))

        x_offset = random.randint(max(offset_range[0], x_range[0] - base_x), min(offset_range[1], x_range[1] - base_x))
        y_offset = random.randint(max(offset_range[0], y_range[0] - base_y), min(offset_range[1], y_range[1] - base_y))

        new_point = (base_x + x_offset, base_y + y_offset)
        all_points.add(new_point)

    return list(all_points)
```

Pomocné funkcie

Evaluácia úspešnosti algoritmov

Pri evaluácii prechádzam už hotovými clusterami a vypočítavam euklidovskú vzdialenosť pomocou funkcie np.linalg.norm() pre každý bod v clusteri a pripočítam to do premennej success_rate. Na konci to len predelím počtom všetkých bodov a zistím z toho percentuálnu úspešnosť algoritmu.

```
def evaluate_clusters(centroids, clusters):
    success_rate = 0

    for centroid, points in zip(centroids, clusters.values()):
        distances = [np.linalg.norm(np.array(point) - np.array(centroid)) for point in points]
        average_distance = np.mean(distances)

        if average_distance <= 500:
            success_rate += 1
```

```
    success_percentage = (success_rate / len(clusters)) * 100
    return success_percentage
```

Vizualizácia

Na vizualizáciu používam knižnicu matplotlib. Vo funkcii prechádzam cez všetky clustre zatiaľ čo priraďujem každému rôzne farby a vykresľujem to na vytvorenú „figure“.

```
def visualize_cluster(centroids, clusters):
    cmap = plt.get_cmap('tab20')  # 'tab20' has 20 distinct colors

    plt.figure(figsize=(10, 10))

    # Generate the plot for each cluster using a color from the colormap
    for i, cluster in clusters.items():
        # Separate the points into x and y lists
        xs, ys = zip(*cluster)
        color = cmap(i % cmap.N)  # Use modulo to cycle through the colormap
if necessary
        plt.scatter(xs, ys, c=[color], label=f'Cluster {i}', alpha=0.6)

    # Plot centroids
    cent_xs, cent_ys = zip(*centroids)
    plt.scatter(cent_xs, cent_ys, c='k', marker='x', s=100,
label='Centroids')

    plt.title('Cluster Visualization')
    plt.xlabel('X Coordinate')
    plt.ylabel('Y Coordinate')
    plt.legend()
    plt.grid(True)
    plt.show()
```

Algoritmy

K-means kde centroid je stred

Na začiatku si nainicializujem *k* centroidov náhodne zo všetkých bodov, kde *k* je počet clusterov. Ďalej inicializujem clusters ako slovník s *k* kľúčmi. No a ďalej iterujem cez všetky body pričom pri každom počítam euklidovskú vzdialenosť kde nájdem najmenšiu vzdialenosť pre určitý centroid a priradím bod k nemu. Následne vypočítam nový centroid v clusteri, ktorý má už priradené body a na konci skontrolujem, či algoritmus skonvergoval alebo nie. Ak nebola žiadna zmena oproti poslednej iterácii, cyklus sa zruší. Funkcia vracia pole centroidov a slovník clusterov.

```
def k_means(points, k=10, max_iterations=100):
    # initialize centroids
    centroids = random.sample(points, k)
    clusters = None

    for _ in range(max_iterations):
        # assign points to the nearest centroid
        clusters = {i: [] for i in range(k)}
        for point in points:
            distances = [np.linalg.norm(np.array(point) - np.array(centroid))
for centroid in centroids]
            min_distance_idx = distances.index(min(distances))
            clusters[min_distance_idx].append(point)

        # update centroids
        new_centroids = []
        for idx in range(k):
            cluster_points = clusters[idx]
            new_centroid = np.mean(cluster_points, axis=0)
            new_centroids.append(new_centroid)

        # check for convergence (if centroids don't change)
        if all([np.array_equal(new_centroids[i], centroids[i]) for i in
range(k)]):
            break

        centroids = new_centroids

    return centroids, clusters
```

K-means kde medoid je stred

Tento algoritmus má niektoré veci spoločne s predošlým algoritmom. V tomto prípade ale používame medoidy, ktoré sú reálne body v clusteri. Na začiatku si takisto nainicializujem medoidy z bodov, ktoré sa vygenerovali a clustre ako slovník. Takisto ďalej počítam euklidovskú vzdialenosť medzi bodmi v clusteri a priraďujem bod s najmenšou vzdialenosťou k medoidu. Rozdiel prichádza pri update medoidov. Na začiatku vyberiem všetky body z clustera, ale nový medoid vyberám ako bod, ktorý minimalizuje súčet vzdialeností v clusteri. Nakoniec skontrolujem, či algoritmus skonvergoval a buď sa loop ukončí alebo ďalej iterujem.

```
def k_means_medoids(points, k=10, max_iterations=100):
    # initialize medoids
    medoids = random.sample(points, k)
    clusters = None
```

```
    for _ in range(max_iterations):
        # assign points to the nearest medoid
        clusters = {i: [] for i in range(k)}
        for point in points:
            distances = [np.linalg.norm(np.array(point) - np.array(medoid))
for medoid in medoids]
            min_distance_idx = distances.index(min(distances))
            clusters[min_distance_idx].append(point)

        # update medoids
        new_medoids = []
        for idx in range(k):
            cluster_points = clusters[idx]
            # find the point in the cluster that minimizes the sum of
distances to all other points in the cluster
            medoid = min(cluster_points, key=lambda p:
sum(np.linalg.norm(np.array(p) - np.array(other)) for other in
                                                        cluster_points))
            new_medoids.append(medoid)

        # check for convergence (if medoids don't change)
        if all([medoids[i] == new_medoids[i] for i in range(k)]):
            break

        medoids = new_medoids

    return medoids, clusters
```

Divízivné zhlukovanie

Divízivné zhlukovanie v skratke funguje na princípe iteratívneho rozdelovania clusterov. Na začiatku nainicializujem potrebné premenné ako jeden veľký cluster so všetkými vygenerovanými bodmi a centroid pre všetky body. Ďalej vojdem do loopu, kde na začiatku hľadám najväčší cluster, ktorý odstránim z clusterov a pridám ho do premennej largest_cluster pre ktorú následne vypočítam centroid. Implementoval som miernu optimalizačnú pomôcku, kde si nájdem dva najvzdialenejšie body. Následne rozdelím cluster na dva nové clustre pomocou spomínaných najvzdialenejších bodov. Ďalej hľadám nové centroidy pre clustre a potencionálne pridelím body k inému centroidu. Na konci priďelujem dva nové clustere k existujúcim.

```
def divisive_clustering(points, max_clusters=10):
    clusters = {0: points}
    centroids = [np.mean(points, axis=0)]

    while len(clusters) < max_clusters:
        # find the largest cluster
        largest_cluster_id = max(clusters, key=lambda k: len(clusters[k]))
        largest_cluster = clusters.pop(largest_cluster_id)
        cluster_centroid = np.mean(largest_cluster, axis=0)

        # choose seed points
        farthest_point = max(largest_cluster, key=lambda p:
np.linalg.norm(np.array(p) - cluster_centroid))
        second_farthest_point = max(largest_cluster, key=lambda p:
np.linalg.norm(np.array(p) -

np.array(farthest_point)))
```

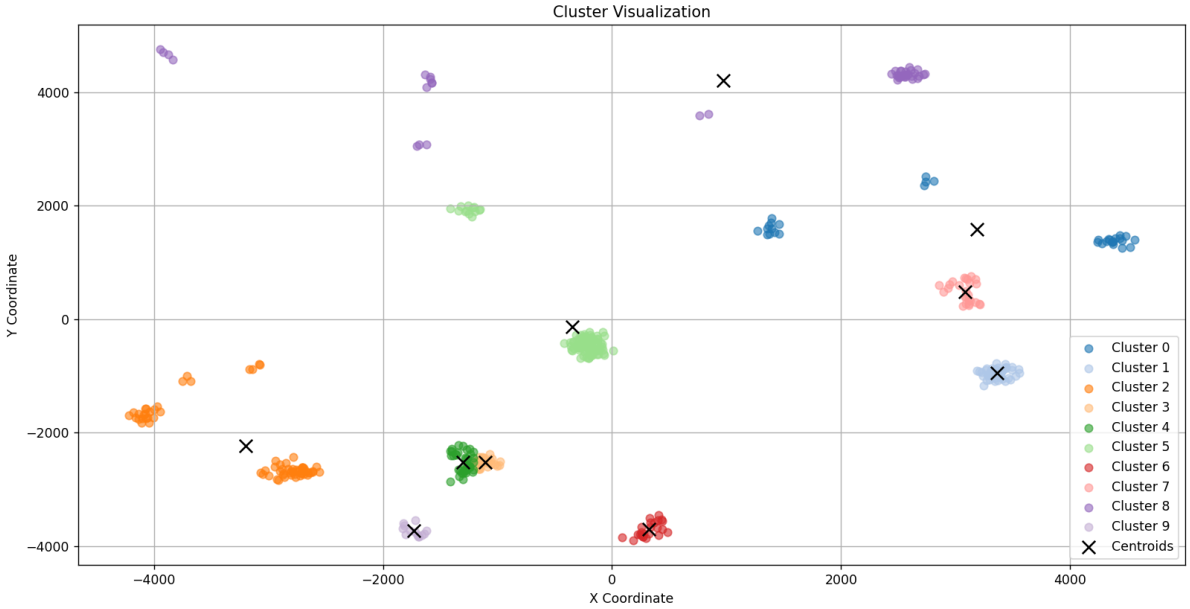
```
        # initial split
        new_clusters = {0: [], 1: []}
        for point in largest_cluster:
            distances = [np.linalg.norm(np.array(point) - np.array(seed)) for
seed in [farthest_point,

second_farthest_point]]
            closest_cluster = distances.index(min(distances))
            new_clusters[closest_cluster].append(point)

        # refine clusters
        for _ in range(50):  # number of iterations for refinement
            new_centroids = [np.mean(new_clusters[i], axis=0) for i in
new_clusters]
            stable = True

            for i in new_clusters:
                updated_cluster = []
                for point in new_clusters[i]:
                    distances = [np.linalg.norm(np.array(point) -
np.array(centroid)) for centroid in new_centroids]
                    closest_cluster = distances.index(min(distances))
                    if closest_cluster != i:
                        stable = False
                    updated_cluster.append(point)
                new_clusters[i] = updated_cluster

            if stable:
                break

        # add new clusters
        cluster_ids = list(range(len(centroids), len(centroids) + 2))
        for new_id, cluster_points in zip(cluster_ids,
new_clusters.values()):
            if len(centroids) < max_clusters - 1:  # Adjust the condition to
account for two new centroids
                clusters[new_id] = cluster_points
                centroids.append(np.mean(cluster_points, axis=0))
            elif len(centroids) < max_clusters:
                # if adding both centroids would exceed the limit, only add
one
                clusters[new_id] = cluster_points
                centroids.append(np.mean(cluster_points, axis=0))
                return centroids, clusters
    return centroids, clusters
```

Testovanie

400 bodov

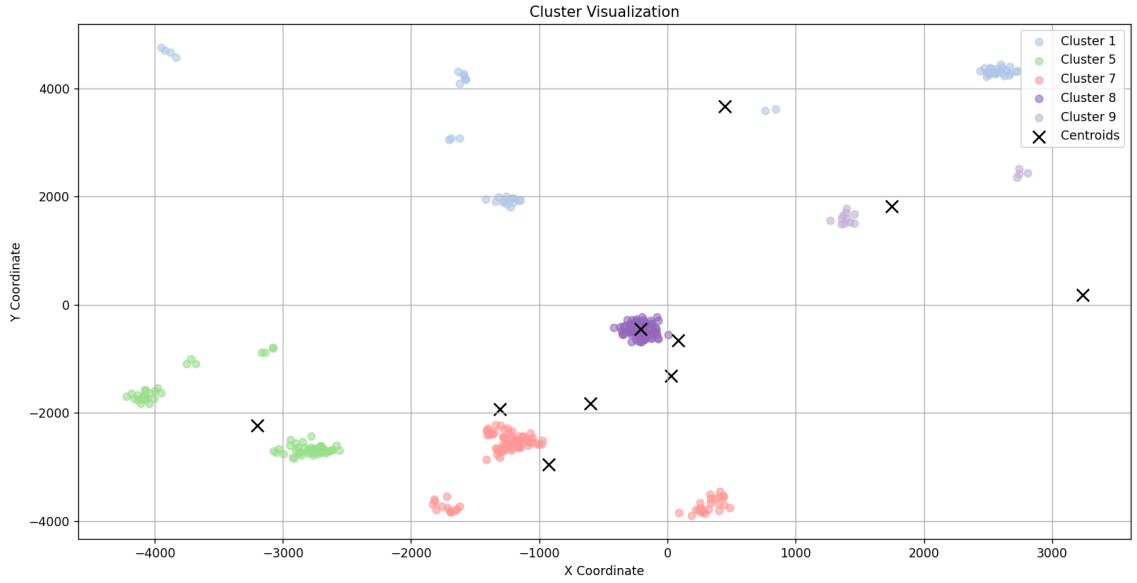
K-means centroid

```
Execution time k-means(Centroid as center): 0.15989184379577637s
Success rate k-means(Centroid as center): 60.0%
```



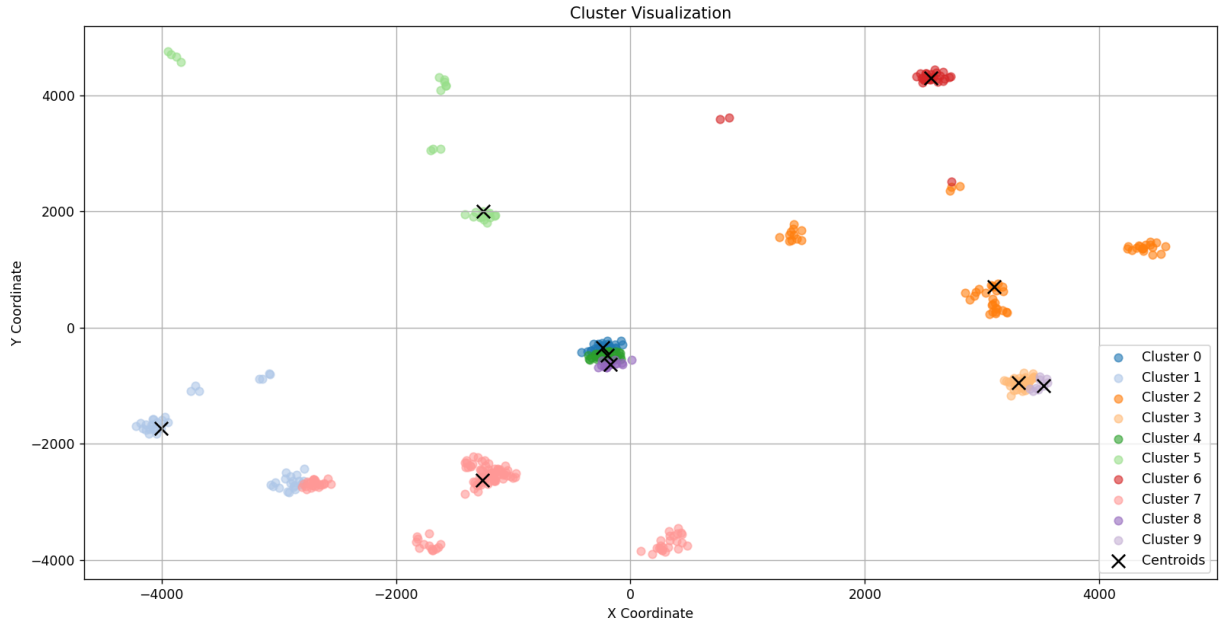
Divisive clustering

```
Execution time k-means(Centroid as center): 0.407506942749023445s
Success rate k-means(Centroid as center): 0.0%
```



K-means medoid

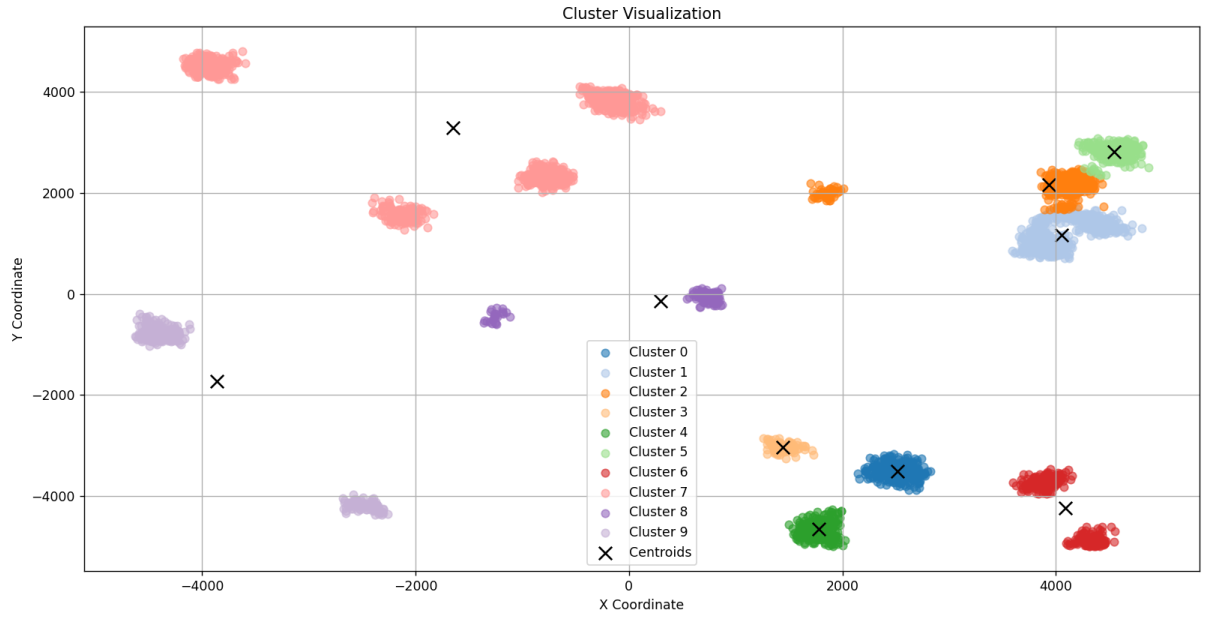
```
Execution time k-means(Medoid as center): 0.46387171745300293s
Succes rate k-means(Medoid as center): 60.0%
```



4000 bodov

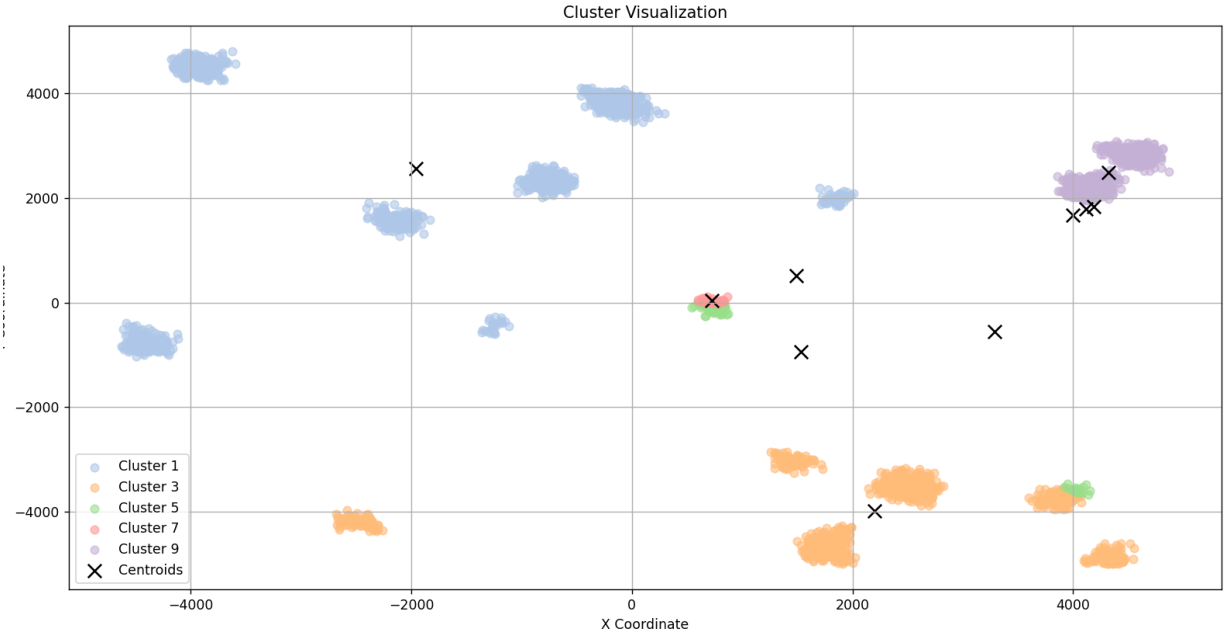
K-means centroid

```
Execution time k-means(Centroid as center): 3.7204723358154297s
Success rate k-means(Centroid as center): 60.0%
```



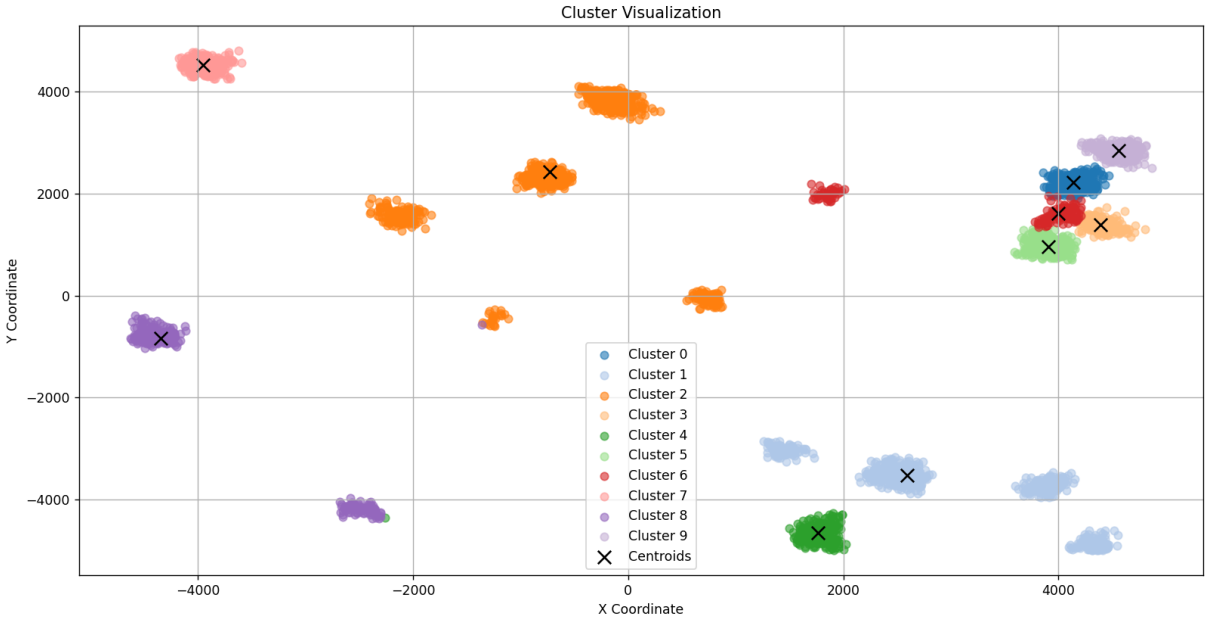
Divisive clustering

```
Execution time k-means(Centroid as center): 5.364577770233154s
Success rate k-means(Centroid as center): 0.0%
```



K-means medoid

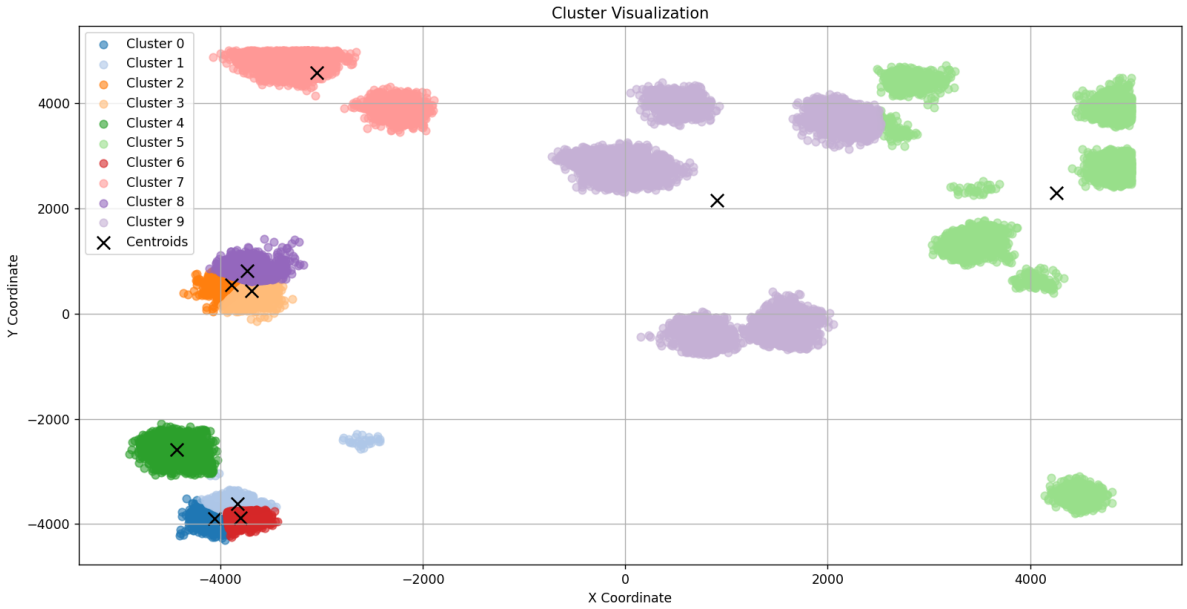
```
Execution time k-means(Medoid as center): 40.79468822479248s
Succes rate k-means(Medoid as center): 60.0%
```



40000 bodov

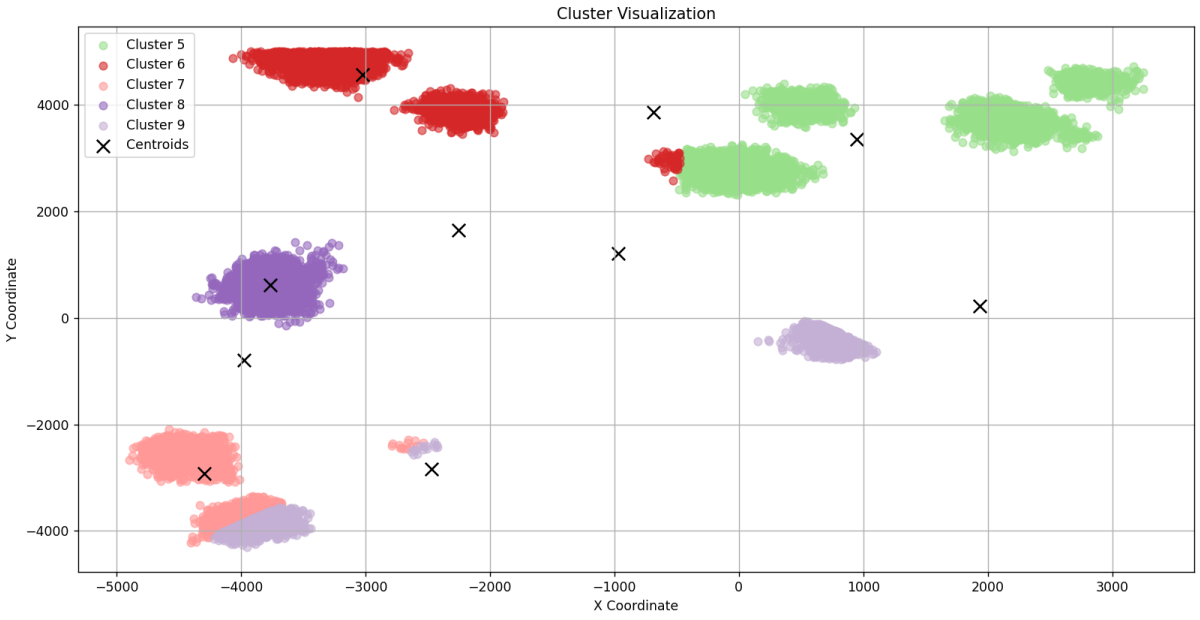
K-means centroid

```
Execution time k-means(Centroid as center): 104.31123352050781s
Success rate k-means(Centroid as center): 70.0%
```



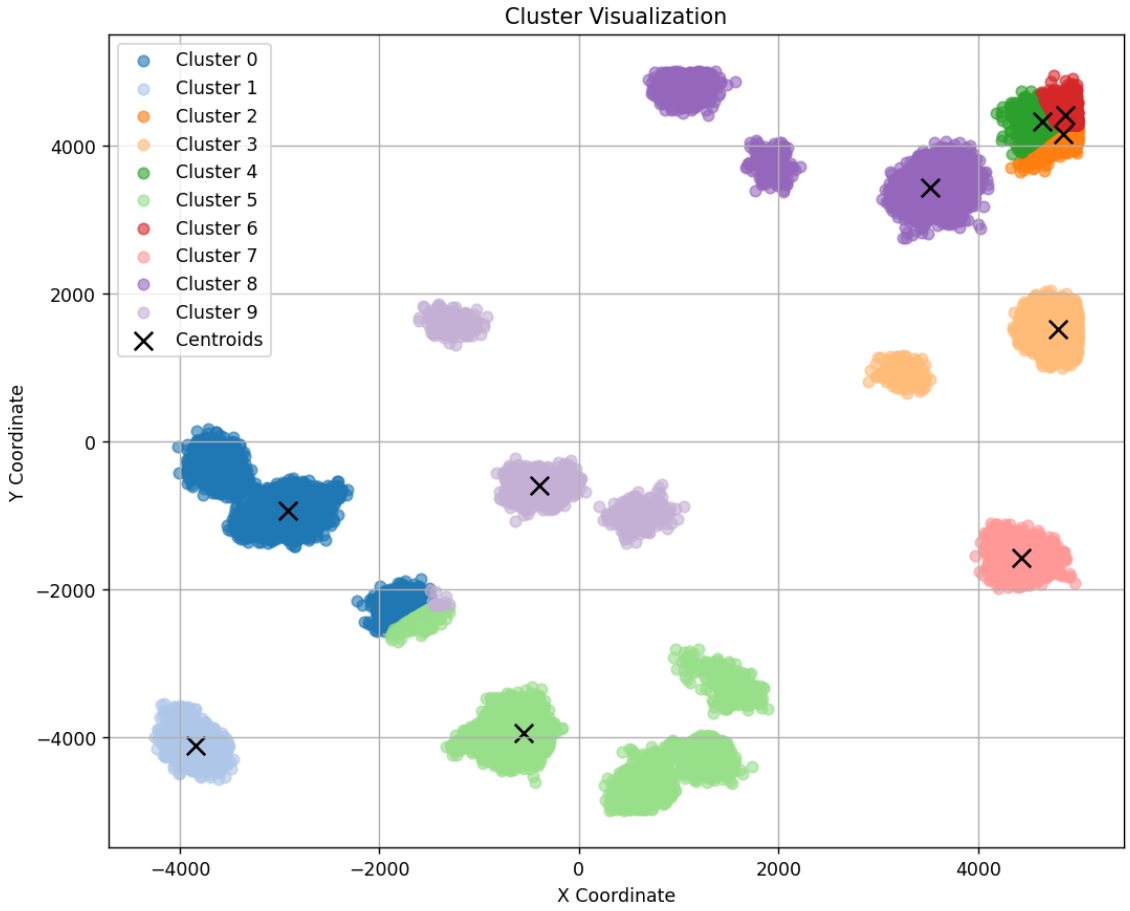
Divisive clustering

```
Execution time k-means(Centroid as center): 36.69831562042236s
Success rate k-means(Centroid as center): 0.0%
```



K-means medoid

```
Execution time k-means(Medoid as center): 17077.518937587738s
Succes rate k-means(Medoid as center): 60.0%
```



Záver

K-means s centroidmi vracalo najlepšie výsledky za reálny čas, zatiaľ čo Divisive clustering je síce rýchle ale výsledky sú úplne zlé. Neviem prečo to tak je, skúšal som rôzne parametre meniť a nič veľmi nepomáhalo. Na papiery znie Divisive Clustering dobre, a keď sa na to kukám v kóde, nevidím dôvod prečo by to nemalo fungovať dobre, ale nejak to vracia zlé výsledky. Nakoniec tu je ale K-means s medoidmi, kde časová zložitosť rastie exponencionálne a pri 40000 bodoch sa algoritmus vykonáva takmer 5 hodín čo nie je optimálne.