|  |
| --- |
| Slovenská Technická Univerzita Fakulta Informatiky a Informačných technológií |
| Clustering |
| Umelá Inteligencia, Zadanie 3 |

|  |
| --- |
| Norbert Matuška  12-8-2023 |

Contents

[Zadanie 1](#_Toc153187829)

[Riešenie 2](#_Toc153187830)

[Generovanie počiatočných bodov 2](#_Toc153187831)

[Pomocné funkcie 2](#_Toc153187832)

[Evaluácia úspešnosti algoritmov 2](#_Toc153187833)

[Vizualizácia 3](#_Toc153187834)

[Algoritmy 4](#_Toc153187835)

[K-means kde centroid je stred 4](#_Toc153187836)

[K-means kde medoid je stred 4](#_Toc153187837)

[Divízivné zhlukovanie 5](#_Toc153187838)

[Testovanie 7](#_Toc153187839)

[400 bodov 7](#_Toc153187840)

[K-means centroid 7](#_Toc153187841)

[Divisive clustering 7](#_Toc153187842)

[K-means medoid 8](#_Toc153187843)

[4000 bodov 8](#_Toc153187844)

[K-means centroid 8](#_Toc153187845)

[Divisive clustering 9](#_Toc153187846)

[K-means medoid 9](#_Toc153187847)

[40000 bodov 10](#_Toc153187848)

[K-means centroid 10](#_Toc153187849)

[Divisive clustering 10](#_Toc153187850)

[K-means medoid 11](#_Toc153187851)

[Záver 12](#_Toc153187852)

# 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

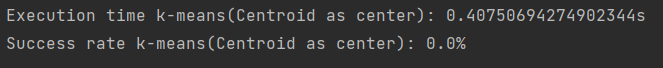
### K-means centroid



A grid of colored dots

Description automatically generated

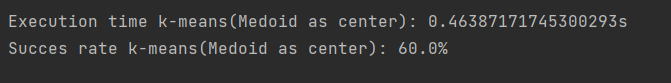
### Divisive clustering



A graph with different colored dots

Description automatically generated

### K-means medoid

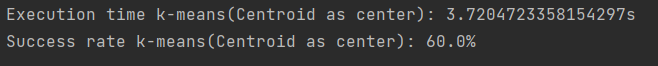


A graph with colored dots

Description automatically generated with medium confidence

## 4000 bodov

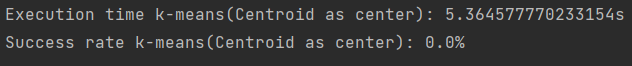
### K-means centroid



A white grid with colorful spots

Description automatically generated

### Divisive clustering



A graph with different colored clouds

Description automatically generated

### K-means medoid

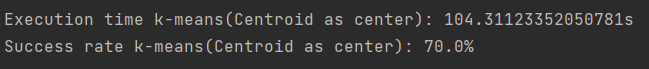


A chart with different colored spots

Description automatically generated

## 40000 bodov

### K-means centroid



A graph with different colored spots

Description automatically generated with medium confidence

### Divisive clustering



A grid with different colored spots

Description automatically generated

### K-means medoid



A diagram of cluster visualization

Description automatically generated

# 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.