SLOVENSKÁ TECHNICKÁ UNIVERZITA V BRATISLAVE

Fakulta informatiky a informačných technológii

Umelá Inteligencia

Zadanie č.4:

Klastrovanie

Tomáš Socha

ID: 110896

Akademický rok 2021/2022

1. 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). Následne je generovaných ďalších 20000 bodov, kde každý musí byť od 1 ľubovoľne zvoleného už vygenerovaného bodu vzdialený nie viac ako o 100 po x a y súradnici.

Úlohou je naprogramovať zhlukovač pre 2D priestor, ktorý zanalyzuje 2D priestor so všetkými jeho bodmi a rozdelí tento priestor na k zhlukov (klastrov). Je potrebné implementovať rôzne verzie zhlukovača, konkrétne s týmito algoritmami:

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

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

2. Implementácia

Vo vlastnej implementácií som použil rôzne algoritmy, ktoré zabezpečia rozdelenie bodov do klastrov. Konkrétne som implementoval K-means kde stred bol **centroid** a **medoid** a taktiež **divízne a aglomeratívne zhlukovanie**. Následne som porovnal, ako vhodne dokážu tieto algoritmy rozdeliť množinu bodov na klastre, a ktorý je za akých podmienok vhodnejšie použiť. Na realizáciu som ako v predošlých zadaniach použil jazyk Python vo verzii 3.9.

a. Použité knižnice

Pre zabezpečenie fungovania programu bolo potrebné použiť nasledujúce knižnice:

- ***Random ->*** funkciu random na vygenerovanie náhodných čísel
- ***Timeit ->*** na zmeranie času vykonávania jednotlivých funkcií
- ***Math ->*** funkciu *sqrt()* použitú pri výpočte euklidovskej vzdialenosti.
- ***scipy.cluster.hierarchy ->*** funkcie na vykreslenie dendogramu
- ***matplotlib ->*** funkcie na vykreslenie grafov

b. Globálne premenné

Použil som globálne premenné ako parametre pre algoritmy. Konkrétne som použil premennú *centroids_pocet* na nastavenie počtu centroidov a medoidov pre K-means algoritmy. Taktiež som použil premennú *pocet_bodov* ktorá určuje **veľkosť množiny** vygenerovaných bodov používanú jednotlivými algoritmami.

Následne som použil premennú *colors*, ktorá načíta zo súboru vopred vygenerovanú paletu farieb[1]. Premenná *list_cely_graf* obsahuje všetky body z rozsahu grafu, ktoré sa neskôr zafarbujú podľa najbližšieho centroidu a slúžia ako pozadie na grafe. Premenná *cislo_grafu* slúži na uloženie jednotlivých vygenerovaných **grafov**, aby mal každý jedinečné meno.

```
colors = []
list_cely_graf = []
cislo_grafu = 1

centroids_pocet = 10
pocet_bodov = 20000
```

Taktiež som použil ďalšie globálne premenné, a to 4 listy pre uloženie hodnôt z jednotlivých funkcií pre vyhodnotenie testovania.

[1] https://github.com/taketwo/glasbey

c. Triedy

Pri tomto zadaní som taktiež použil zopár tried pre jednoduchšiu a prehľadnejšiu reprezentáciu premenných. Pre Uloženie **centroidu** som vytvoril triedu *Centroid*. Okrem suradníc a listu všetkých bodov patriacich pod daný centroid si taktiež udržuje sumu všetkých x a y suradníc bodv pre rýchlejšie vypočítanie nových súradníc a samozrejme aj priemernú vzdialenosť od jednotlivých bodov pre vyhodnotenie úspešnosti a rozmedzia v akom sa prislúchajúce body nachádzajú.

```
class Centroid:
    suradnice = None
    list_bodov = None
    sum_x_suradnic = 0
    sum_y_suradnic = 0
    avg_vzdialenost = 0
```

Pre uloženie **medoidu** som vytvoril novú triedu *Klaster* pretože si pri jeho reprezentácií nepotrebujem pamätať toľko vecí.

Túto triedu som použil aj pri **aglomeratívnom zhlukovaní** za účelom šetrenia pamäte pri ukladaní jednotlivých centroidov, pretože na začiatku je pre každý bod vytvorený nový klaster, čo by pri použití už vytvorenej triedy centroid zbytočne zaberalo viac pamäte.

```
class Klaster:
    suradnice = None
    list_bodov = None
```

Poslednou triedou ktorú som využil bola trieda pre zaznamenanie jednotlivých stavov pri divíznom zhlukovaní. Vďaka tomuto som sa mohol po vykonaní algoritmu vrátiť ku roztriedeniu pre hocijaký počet centroidov.

```
class Divisive_rozlozenie:
    list_klastrov = None
    predch_rozlozenie = None
```

d. K-means, kde stred je centroid

K-means je jednoduchý zhlukovací algoritmus, ktorý dostane na vstupe vopred stanovený počet zhlukov, na koľko má množinu bodov rozdeliť. Následne sú náhodne vygenerované pozície bodov(centroidy), kde každý vygenerovaný bod predstavuje jeden zhluk. Množina všetkých bodov sa prerozdelí medzi tieto zhluky a z každého zhluku sa zistia nové súradnice ich centroidu, čo predstavuje bod ktorý má priemernú vzdialenosť od ostatných bodov najbližšiu. Následne sa podľa nových súradníc cetroidov vytvoria nové zhluky a cyklus sa opakuje, kým sa tieto súradnice ustália.

e. K-means, kde stred je medoid

Ide o totožný algoritmus ako vyššie spomenutý, kde sa však pre zadelenie jednotivých bodov do zhlukov používa namiesto centroidu medoid. Ide o reálny bod z množiny bodov, ktorého priemerná vzdialenosť so zvyškom daného zhluku je najmenšia.

f. Aglomerativne zhlukovanie

Ide o komplikovanejší algoritmus ako K-means. Jeho princíp spočíva v tom, že na začiatku každý bod predstavuje jeden zhluk. V tejto množine zhlukov sa najde dvojica, ktorých vzdialenosť je najmenšia a spojí sa do 1 zhluku. Takýto postup sa opakuje až kým nedosiahneme nami požadovaný počet zhlukov, alebo nesplníme nejakú podmienku pre ktorú bude dané rozloženie dostačujúce.

g. Divízne zhlukovanie

Tento algoritmus funguje opačne ako aglomerativne zhlukovanie. Na začiatku je jeden zhluk kde sú všetky body, ktorý sa rozdelí na 2, a tie sú následne rozdelené na ďalšie.

3. Opis Riešenia

Mnou implementované riešenie sa spúšťa z hlavnej funkcie *main()* kde sa zavolá príslušná funkcia ktorej názov zodpovedá použitému algoritmu. Pre správne fungovanie sa však najskôr zavolá funkcia *nacitanie_farieb()* ktorá načíta uložené hexadec. Hodnoty farieb ktoré sa použijú pri vykresľovaní grafov. Následne sa pomocou funkcii *generator_start(), generator_continue()* a *graf_vsetky_body()* vygeneruje množina bodov ktoré sa budú rozdeľovať, a množina bodov zabezpečujúca vykreslenie grafov.

K_means_centroid()

Funkcia je rozdelená do 2 častí. V prvej je *while* cyklus slúžiaci na vygenerovanie potrebného počtu súradníc pre centroidy. Tieto centroidy nie sú generované úplne náhodne, ale vždy sa vyberú 2 náhodné body, zistí sa stred medzi nimi a z toho bodu sa vygeneruje náhodný posun max o +-500. čiže súradnice budú rovné strednému bodu zvečšeného o offset. Druhý while cyklus má v sebe for cyklus obsahujúci ďalší cyklus slúžiaci na prideľovanie jednotlivých bodov ku centroidom. Následne po rozdelení všetkých bodov sa pre každý zhluk bodov vypočítajú nové súradnice centoridov a proces sa opakuje kým nebude najväčší posun spomedzi súradníc centroidov menší ako 10.

K_means_medoid()

Funkcia pracuje na podobnom princípe ako pre centroid. V prvom while cykle sú náhodne vybrané body ktoré sa prehlásia za medoidy. Druhý while cyklus tiež obsahuje for cyklus pre prechádzanie všetkých bodov, kde sa rozdelia do jednotlivých zhlukov.

Následne v ďalšom for cykle sa prechádza cez všetky medoidy a volá sa funkcia *najdenie_medoidu(),* ktorá nájde nový medoid pre danú množinu bodov. To je vykonané pomocou vypočítania priemernej vzdialenosti pre každý bod a vybratia bodu s najmenšou vzdialenosťou. Pre nové medoidy sa následne cyklus rozdeľovania bodov opakuje. Cyklus skončí po 15 iteráciách, alebo keď by väčšina medoidov vstúpila do stavu v ktorom už niekedy boli, čiže sa už zacyklili a nenájdu lepšie riešenie.

Aglomerative_clustering()

Na začiatku sa pre každý bod vytvorí vlastný zhluk. Následne sa vytvorí matica vzdialeností medzi každou dvojicou bodov. Nasleduje while cyklus v ktorom sa deje všetko podstatné. Najskôr sa pomocou funkcie *min()* zistí najmenšia hodnota v jednom riadku. Táto hodnota sa zistí pre každý riadok a z toho dostaneme globálne minimum. Zoberú sa indexy tejto hodnoty, ktoré predstavujú 2 klastre, oni sa zlúčia do jedného, vymaže sa riadok a stĺpec zlúčeného klastra a aktualizujú sa hodnoty pre novo vzniknutý klaster. Tento postup sa opakuje kým nedostaneme 1 veľký klaster, kde je možné vybrať si počas behu programu rozloženie pre ľubovoľný počet klastrov a ten interpretovať.

Divisive_clustering()

Pri divíznom zhlukovaní sa v cykle volá funkcia pre rozdelenie vstupných dát vždy do 2 zhlukov. Tieto zhluky sú pridané do listu zhlukov, vytvorí sa aktuálny stav pre počet zhlukov, následne sa vyberie zhluk, ktorý má najväčšiu priemernú vzdialenosť, ten sa vyberie a pri ďalšej iterácií sa on bude deliť na dva.

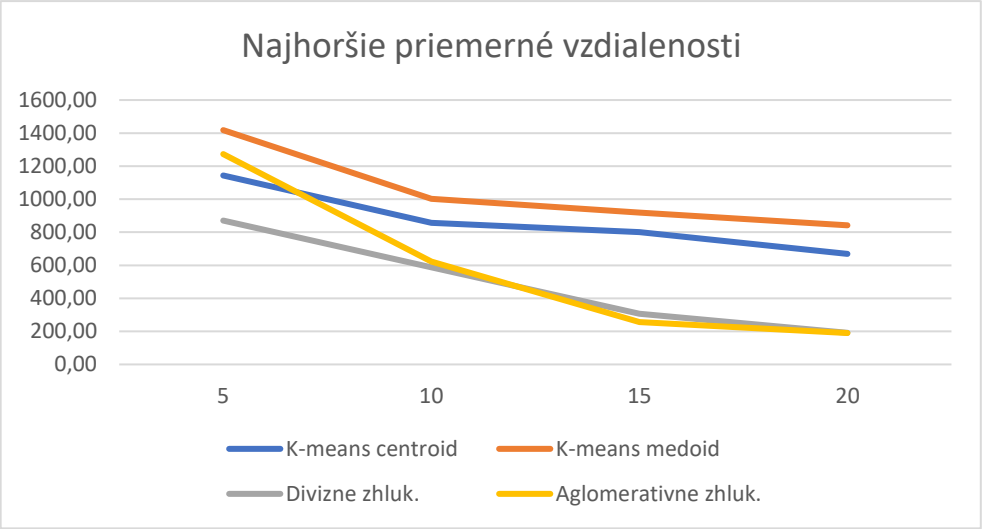
V programe sa taktiež nachádzajú funkcie pre vykreslenie grafov a dendogramu ako aj funkcia ktorá zabezpečila vykonanie jedného z testov kde sa tporovnávali všetky algoritmy.

4. Testovanie

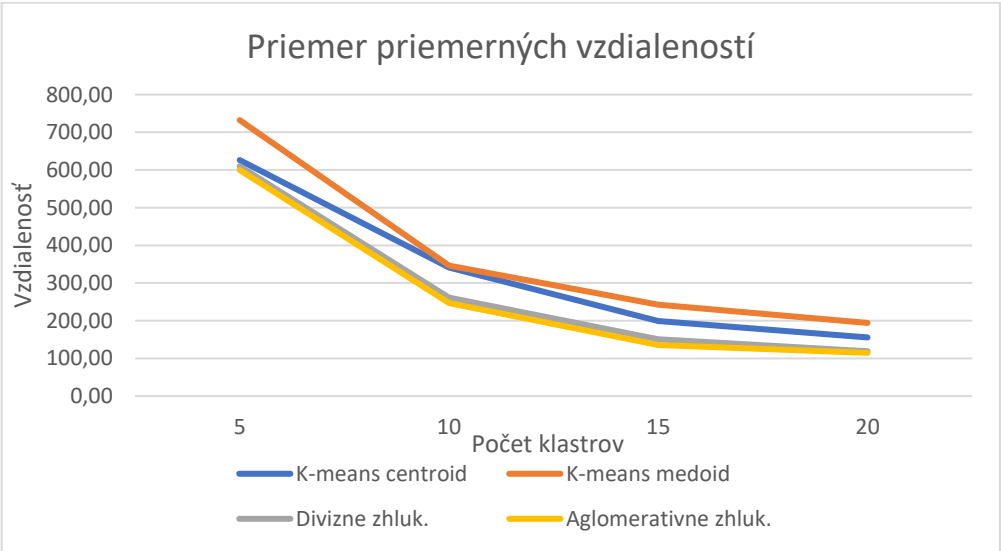
V nasledujúcom teste som sa pozrel na jednotlivé algoritmy a porovnal ich časy a úspešnosti zhlukovania pri rôznych počtov zhlukov. Test som vykonal s 2000 náhodne vygenerovanými bodmi na začiatku, ktoré boli postupne zhlukované do 5,10,15 a nakoniec 20 klastrov. Pre každú veľkosť som spravil 5 opakovaní a ich priemernú hodnotu premietol do grafov.

Podľa zadania sme mali vyhodnotiť úspešnosť jednotlivých algoritmov, kde za úspešné zhlukovanie sa považoval stav, kedy žiadny zhluk nemal priemernú vzdialenosť od bodov väčšiu ako 500. Táto hodnota je zobrazená v prvom grafe znázorňujúci najhoršie spomedzi priemerných vzdialeností všetkých klastrov.

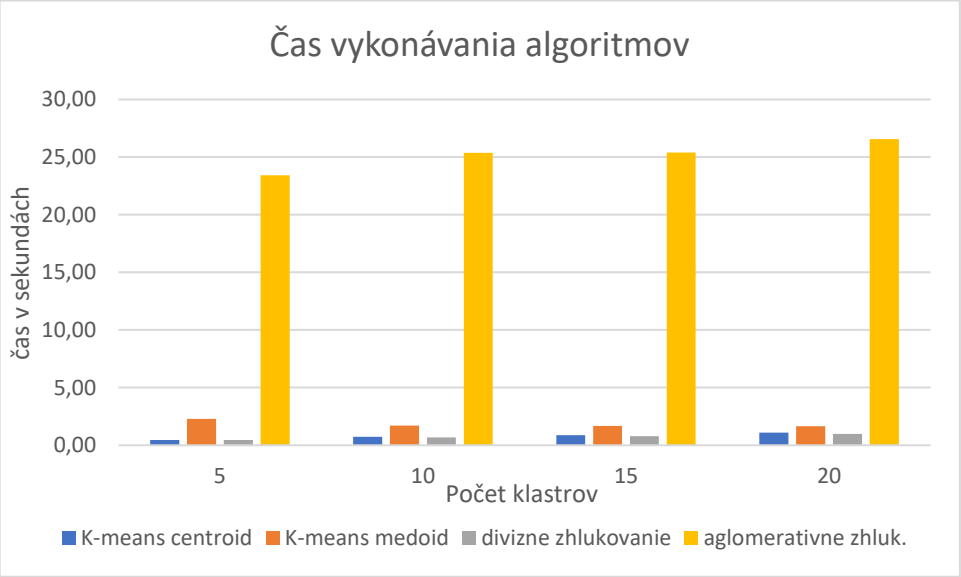
Túto hranicu sa nepodarilo dosiahnuť ani jednému K-means algoritmu, čo je spôsobené najmä náhodným generovaním počiatočných centroidov a medoidov. Naopak, veľmi slušne obstáli zostávajúce 2 algoritmy, ktoré dokázali veľmi presne identifikovať zhluky vo vygenerovanej množine.



Nasledujúci graf zobrazuje priemerné hodnoty priemerných vzdialeností jednotlivých klastrov. Na moje prekvapenie najhoršie vo všetkých prípadoch dopadol K-means medoid, čo mohlo byť zapríčinené nie úplne náhodným generovaním centroidov ako som spomínal v opise riešenia.



Z časového hľadiska dopadlo najhoršie aglomeratívne zhlukovanie, čo nieje žiadným prekvapením vzhľadom na spôsob jeho fungovania. Časy jednotlivých algoritmov zodpovedajú okrem samotného vykonania algoritmu taktiež aj vykresleniu grafu dosiahnutého výsledku.



Na základe dosiahnutých hodnôt môžem konštatovať, že najlepšie dopadol algoritmus divízneho zhlukovania pretože dosahoval jednoznačne najlepšiu kombináciu času a výsledku. Pri ňom si môžeme taktiež na grofoch všimnúť tzv. elbow point ktorý sa nachádza na 15 klastroch.

5. Používateľské prostredie

Môj program neobsahuje interaktívne prostredie, no je možné meniť jeho správanie na základe globálnych premenných umiestnených na začiatku programu. Je možné si nastaviť počet centroidov ako aj veľkosť množiny vygenerovaných bodov.

Obrázok nižšie ukazuje výpis programu po jeho vykonaní.

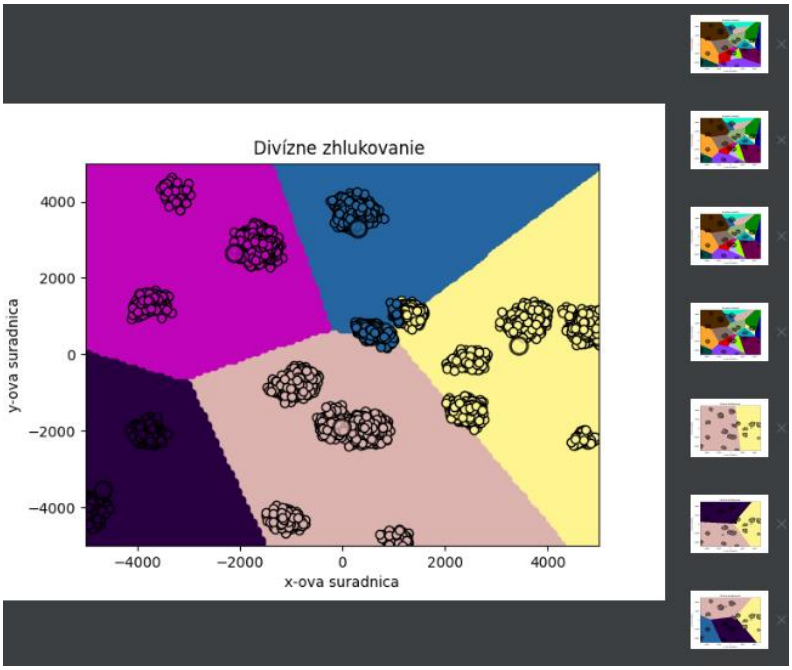
```
K-means centroid
------------
Priemer priemerných vzdialeností bodov od stredov klastrov:  154.65
Najhoršia priemerná vzdialenosť bodov od stredov klastrov:  668.94
1.0884 sekund

K-means medoid
------------
Priemer priemerných vzdialeností bodov od stredov klastrov:  194.12
Najhoršia priemerná vzdialenosť bodov od stredov klastrov:  841.87
1.6653 sekund

Aglomerativne zhlukovanie
------------
Priemer priemerných vzdialeností bodov od stredov klastrov:  114.7
Najhoršia priemerná vzdialenosť bodov od stredov klastrov:  190.06
26.5557 sekund

Divizne zhlukovanie
------------
Priemer priemerných vzdialeností bodov od stredov klastrov:  119.02
Najhoršia priemerná vzdialenosť bodov od stredov klastrov:  191.44
0.9997 sekund
```

Taktiež sa po vykonaní programu vykreslia grafy pre jednotlivé algoritmy.



6. Rozšíriteľnosť a optimalizácia

Program prešiel oproti prvej verzií viacerými úpravami. Oproti pôvodnej verzií som napríklad pri triede centroid použil premenné na zapamätanie celkovej sumy x a y súradníc už pri priraďovaní jednotlivých bodov do klastru, čim som ušetril nejaký čas. Ďalej som pri k-means centroid zmenil generovanie počiatočných súradníc centroidov. Úplne náhodné generovanie som zmenil na generovanie náhodného bodu s offsetom max 500 od stredu 2 náhodne zvolených bodov . Túto zmenu som aplikoval hlavne kvôli divíznemu zhlukovaniu, kedy sa stávalo že pri generovaní len 2 centroidov ,jeden centroid obsiahol celú množinu a druhý teda zostal prázdny. Tieto prípady nastávali hlavne keď sa už pracovalo s menšími množinami, ktoré predstavovali tak 1/8 z celej.

V prípade K-means medoid som musel zmeniť princíp zvolenia nového bodu za medoid. V prvej verzií som si zotriedil všetky body v danom klastri podľa vzdialenosti a následne vybral median, čiže strednú hodnotu. Toto riešenie však nebolo korektné nakoľko nie vždy našlo skutočný bod s najmenšou priemernou vzdialenosťou. To sa následne odrazilo aj na výslednom riešení. Preto som zvolil výpočtovo omnoho náročnejšie, avšak korektné riešenie, kde som pre každý bod vypočítal priemernú vzdialenosť od ostatných bodov a vybral ten s najmenšou a prehlásil ho za medoid. Toto spôsobilo zväčšenie ako časových tak aj pamäťových nárokov, avšak algoritmus už pracuje korektne.

V rámci riešenia optimalizácie som aj skúšal program spustiť cez pypy interpreter, čo výrazne zrýchlilo riešenie, kde program dokázal pri aglomeratívnom zhlukovaní pri 20000 bodov zbehnúť cca za pol hodiny, avšak nepodarilo sa mi zabezpečiť vykreslenie grafov, keďže sa mi nechel nainštalovať knižnica matplotlyb. Po viacerých neúspešných pokusoch som sa vrátil k pôvodnému interpreterovi na ktorom boli aj spustené všetky testy.

Z implementovaných algoritmov bolo pre mňa jednoznačným víťazom divízne zhlukovanie, ktoré vedelo veľmi presne identifikovať zhluky a prišiel mi ako jediný reálne použiteľný kedže aglomeratívne zhlukovanie trvá strašne dlho a K-means nedokáže vôbec tak dobre rozdeliť body do zhlukov.

V zipku prikladám okrem zdrojového kódu aj gify jednotlivých algoritmov.