Slovenská technická univerzita v Bratislave
Fakulta informatiky a informačných technológií

Umelá Inteligencia

Zadanie č.3

a) Klasifikácia

Akademický rok 2022/2023

Meno: Ján Ágh
Cvičiaci: Ing. Martin Komák, PhD.

Dátum: 27.11.2022
Počet strán: 12

Obsah

1 Stručný opis problematiky a zadania

Základom uvedenej úlohy je dvojdimenzionálny priestor rozmerov 10000x10000, pričom jednotlivé súradnice v horizontálnom aj vertikálnom smere pochádzajú z intervalu od -5000 do 5000. Tento priestor obsahuje celkovo 40020 bodov patriacich do jednej zo štyroch kategórií spomedzi *red (R), green (G), blue (B)* a *purple (P),* ale na začiatku (pred spustením algoritmu) je v ňom rozmiestnených iba prvých 20 bodov, identický počet (5) z každej kategórie, podľa vopred staticky zadefinovaných súradníc.

Primárnym cieľom úlohy je implementácia klasifikátora pre zvyšných 40000 bodov, ktorý bude spĺňať nasledovné kritériá:

- *Jeho funkcionalita bude umiestnená v metóde classify(int X, int Y, int k), kde parametre X a Y predstavujú súradnice nového bodu v dvojdimenziálnom priestore a parameter k určuje počet najbližších susedných bodov, podľa ktorých sa nový bod klasifikuje.*
- *Klasifikátor podľa hodnoty k priradí príslušnú kategóriu novému bodu.*
- *Nový bod je umiestnený do dvojdimenzionálneho priestoru.*
- *Metóda classify(int X, int Y, int k) vráti príslušnú pridelenú kategóriu.*

V spomínanej úlohe sa na klasifikáciu má využiť *k-nearest-neighbors (k-NN) algoritmus* a hodnoty k majú pochádzať z množiny {1, 3, 7, 15}. Súradnice nových bodov sa majú generovať náhodne a dva po sebe nasledujúce body nesmú spadať do rovnakej kategórie.

Podľa zadania sa má vykonať klasifikácia pre všetky rôzne hodnoty k s identickými súradnicami klasifikovaných bodov, návratová hodnota metódy *classify(int X, int Y, int k)* sa má porovnávať s automaticky pridelenou kategóriou bodov s cieľom určenia efektivity navrhnutého klasifikátora a každý vykonaný experiment musí byť aj vizualizovaný.

2 Navrhnuté riešenie

2.1 Použité programové prostriedky

Na riešenie úlohy bol použitý programovací jazyk Python verzie 3.10.8. Riešenie bolo vyvíjané v prostredí Visual Studio Code verzie 1.72.1 a pri jeho vývoji boli využité základné aj pokročilejšie objektovo orientované princípy.

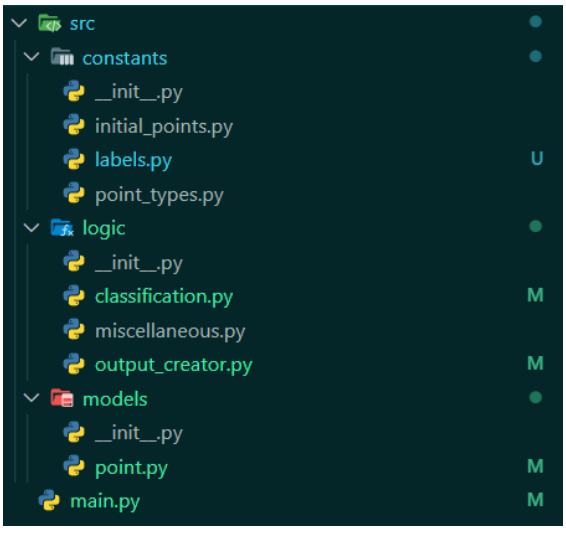
2.2 Organizácia projektu

Na spustenie programu slúži súbor *main*.py a samotná implementácia algoritmu sa nachádza v adresári s názvom "*src*". Tento adresár v sebe obsahuje trojicu ďalších adresárov (modulov) *constants, logic* a *models.*

Adresár *constants* je tvorený tromi súbormi *point_types.py, labels.py* a *initial_points.py*. Prvý menovaný súbor obsahuje enumeráciu jednotlivých kategórií bodov (red, gree, blue a purple), druhý menovaný je naplnený konštantami typu string, ktoré sú využívané na viacerých miestach v programe, a v poslednom súbore sú zadefinované statické súradnice prvých 20 bodov.

V adresári *logic* sú taktiež umiestnené tri súbory, pričom *miscellaneous.py* je domovom viacerých pomocných statických metód využívaných na rozličných miestach, *classification.py* predstavuje jadro programu a obsahuje metódy slúžiace na generovanie bodov a vykonávanie klasifikácie a *output_creator.py* sa stará o grafickú vizualizáciu výsledkov experimentu, či už vrámci grafu, alebo v konzole.

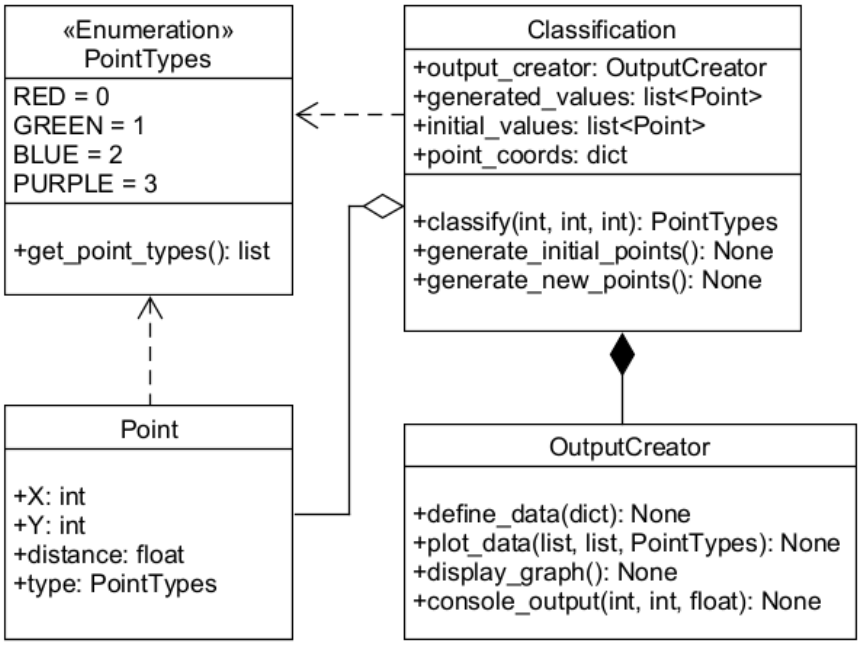
Posledným adresárom je *models* s jediným súborom *point.py* predstavujúcim jeden bod v dvojdimenziálnom priestore s vlastnými súradnicami X a Y, vlastnou kategóriou a vzdialenosťou od bodu, ktorého klasifikácia práve prebieha.



Obr. 2.1 Adresárová štruktúra implementácie

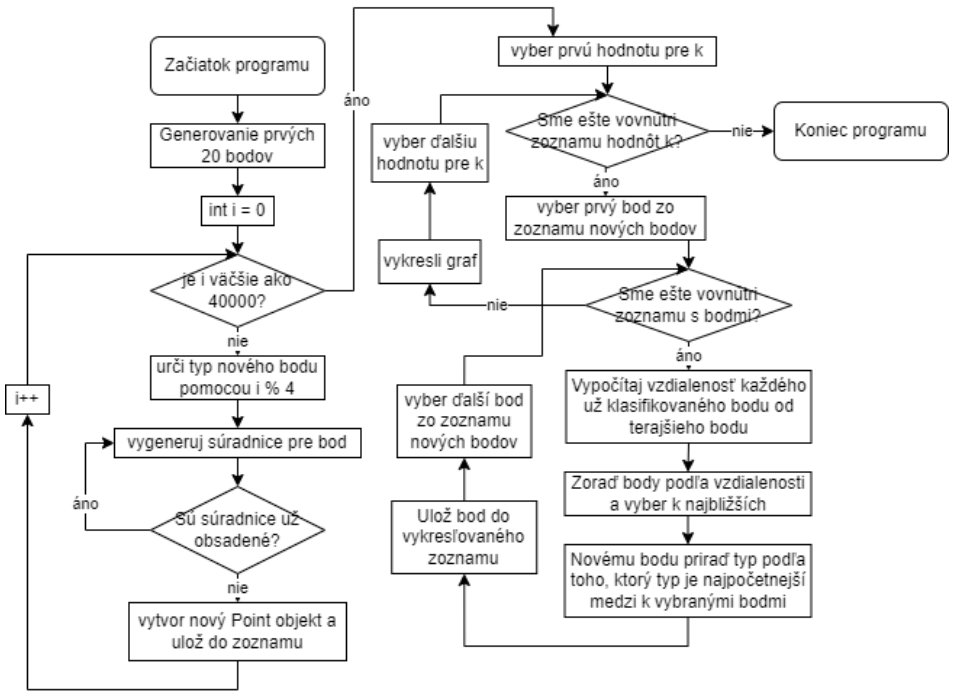
2.3 UML diagram jednotlivých tried

Na obr. 2.2 sa nachádza UML diagram najdôležitejších spolupracujúcich tried algoritmu spolu s jednotlivými úrovňami asociácie, kompozície a agregácie.



Obr. 2.2 UML diagram najdôležitejších tried algoritmu

2.4 Diagram priebehu klasifikácie



Obr. 2.3 Vývojový diagram procesu klasifikácie

2.5 Slovný opis priebehu algoritmu

Ako už bolo spomenuté, vstupným bodom programu je súbor *main.py* obsahujúci objekt triedy *Classification*. V ňom sa postupne volajú metódy *generate_initial_points()* na načítanie staticky zadefinovaných súradníc prvotných 20 bodov a tvorbu objektov typu *Point* pre ne, *generate_new_points()* na vytvorenie požadovaného počtu 40000 náhodných nových bodov a nakoniec *perform_classification()* na vykonanie celého procesu klasifikácie.

```
from src.logic.classification import Classification

classification = Classification()

classification.generate_initial_points()
classification.generate_new_points()

classification.perform_classification()
```

Obr. 2.4 Obsah súboru main.py

Trieda *Classification* si vo svojom konštruktore inicializuje prázdne zoznamy *initial_values* na uchovávanie objektov prvotných bodov a *generated_values*, ktorá má identickú úlohu, ale uchováva objekty náhodne generovaných bodov. Tieto zoznamy sú postupne naplnené metódami *generate_initial_points()* a *generate_new_points()* (obr. 2.5) a ich obsah je nemenný počas celého procesu klasifikácie, keďže podľa zadania je potrebné vykonať klasifikáciu s rôznymi hodnotami k pre identickú množinu bodov.

```
def generate_initial_points(self) → None:
    for point_type, collection in all_initial_points():
        for X, Y in collection:
            X += 5000
            Y += 5000

            self.map[Y][X] = True
            self.initial_values.append(Point(X, Y, point_type))

def generate_new_points(self) → None:
    for index in range(0, 40000):
        point_type = PointTypes.get_point_types()[index % 4]

        X, Y = self.generate_coords(point_type)

        self.map[Y][X] = True
        self.generated_values.append(Point(X, Y, point_type))
```

Obr. 2.5 Metódy na generovanie bodov triedy Classification

Zaujímavosťou je premenná *point_type* metódy *generate_new_points()*. Počas jej inicializácie je zavolaná statická metóda z enumeračnej triedy *PointTypes*, ktorá vráti zoznam všetkých dostupných kategórií bodov. Následne sa z tohto zoznamu vyberie kategória pre nový bod podľa výpočtu *index % 4* (index uchováva poradie novovytvoreného bodu z intervalu od 0 po 39999), čím je zabezpečená skutočnosť, že žiadne dve bezprostredne po sebe nasledujúce body nebudú patriť do rovnakej kategórie.

Za zmienku stojí aj pomocná metóda *generate_coords()* triedy *Classification*, ktorá je tiež využívaná vrámci metódy *generate_new_points()*. Ako to už z jej názvu vyplýva, jej primárnou úlohou je získanie platných súradníc pre nový bod, pričom za platné sa považujú doposiaľ neobsadené súradnice. Na získanie súradníc podľa kategórie bodu (každá kategória má 99%-nú šancu, že sa jej vygenerujú súradnice z vopred stanoveného rozsahu zadefinovaného v zadaní úlohy) slúži metóda *get_random_coords()* statickej triedy *Misc*. Generovanie súradníc prebieha dovtedy, kým sa nenájdu ešte neobsadené súradnice (na kontrolu slúži dvojdimenzionálny zoznam *map*).

```
def generate_coords(self, point_type: PointTypes) → tuple[int, int]:
    X, Y = 0, 0

    while True:
        X, Y = Misc.get_random_coords(point_type)

        if not self.map[Y][X]:
            break
    return (X, Y)
```

Obr. 2.6 Metóda generate_coords() triedy Classification

Po vytvorení všetkých potrebných objektov typu *Point* nasleduje realizácia samotnej klasifikácie. Poslúži na to metóda *perform_classification()* triedy *Classification*, ktorej jedinou úlohou je iterácia cez zoznam platných hodnôt k (zoznam *num_of_neighbors*) a vykonanie úplnej klasifikácie všetkých bodov pre každú z týchto hodnôt. Metóda taktiež určí čas každej klasifikácie, uvedený v sekundách, a vypočíta úspešnosť procesu klasifikácie v percentách.

Najdôležitejšou súčasťou algoritmu je bezpochyby metóda *classify(int X, int Y, int k)* triedy *Classification* (obr. 2.7), slúžiaca na určenie kategórie daného bodu a volaná pre každý z vygenerovaných bodov pre všetky platné hodnoty k.

```
def classify(self, x: int, y: int, neighbors: int) → PointTypes:
    for point in self.current_values:
        point.distance = sqrt((point.X - x) ** 2 + (point.Y - y) ** 2)

    self.current_values.sort(key = lambda point : point.distance)

    return Misc.get_point_type(self.current_values[: neighbors])
```

Obr. 2.7 Metóda classify(int X, int Y, int k) triedy Classification

Princíp metódy je veľmi jednoduchý. Jej primárnou úlohou je vypočítanie vzdialenosti nového klasifikovaného bodu od už všetkých klasifikovaných, uložených v zozname *current_values*. Na výpočet slúži vzorec Euklidovskej vzdialenosti, ktorý určí dĺžku najkratšej priamočiarej cesty medzi dvoma bodmi. Po vypočítaní všetkých vzdialeností sa body v celom zozname *current_values* zoradia podľa nich a zvolí sa k najbližších bodov, podľa ktorých sa následne

určí kategória nového bodu. Aj proces voľby kategórie je priamočiary proces: zistí sa početnosť jednotlivých kategórií medzi zvolenými najbližšími bodmi a nový bod nadobudne najpočetnejšiu kategóriu. Túto skutočnosť realizuje metóda *get_point_type()* triedy *Misc*.

```python
@staticmethod
def get_point_type(nearest_points: list[Point]) -> PointTypes:
    nearest_points_count = [0 for _ in range(4)]

    for value in nearest_points:
        nearest_points_count[value.type.value] += 1

    return PointTypes.get_point_types()[nearest_points_count.index(max(nearest_points_count))]
```

Obr. 2.8 Metóda get_point_type() triedy Misc

Akonáhle sa ukončí proces klasifikácie pre jednu z hodnôt k, nastane roztriedenie vytvorených bodov a zhotovenie grafu. Uvedené procesy sa vykonávajú v metóde *generate_graph_data()* triedy *Classification*, kde sa zoznamy klasifikovaných bodov upravia do tvaru zobraziteľného grafom (roztriedia sa do osobitných zoznamov podľa kategórie), vytvorí sa výpis času a efektivity v konzole a pomocou metód triedy *OutputCreator* prebehne zhotovenie a zobrazenie grafu.

```python
def generate_graph_data(self, neighbor: int, time: float) -> None:
    for point in self.current_values:
        X, Y = point.X, point.Y

        X -= 5000
        Y -= 5000

        match point.type:
            case PointTypes.RED:
                self.point_coords[labels.RED_X].append(X)
                self.point_coords[labels.RED_Y].append(Y)
            case PointTypes.GREEN:
                self.point_coords[labels.GREEN_X].append(X)
                self.point_coords[labels.GREEN_Y].append(Y)
            case PointTypes.BLUE:
                self.point_coords[labels.BLUE_X].append(X)
                self.point_coords[labels.BLUE_Y].append(Y)
            case _:
                self.point_coords[labels.PURPLE_X].append(X)
                self.point_coords[labels.PURPLE_Y].append(Y)

    self.output_creator.console_output(
        self.guessed_num_of_points,
        self.total_num_of_points,
        time
    )
    self.output_creator.initialize_graph_settings(neighbor)
    self.output_creator.define_data(self.point_coords)
    self.output_creator.display_graph()
```

Obr. 2.9 Metóda generate_graph_data() triedy Classification

3 Testovanie algoritmu

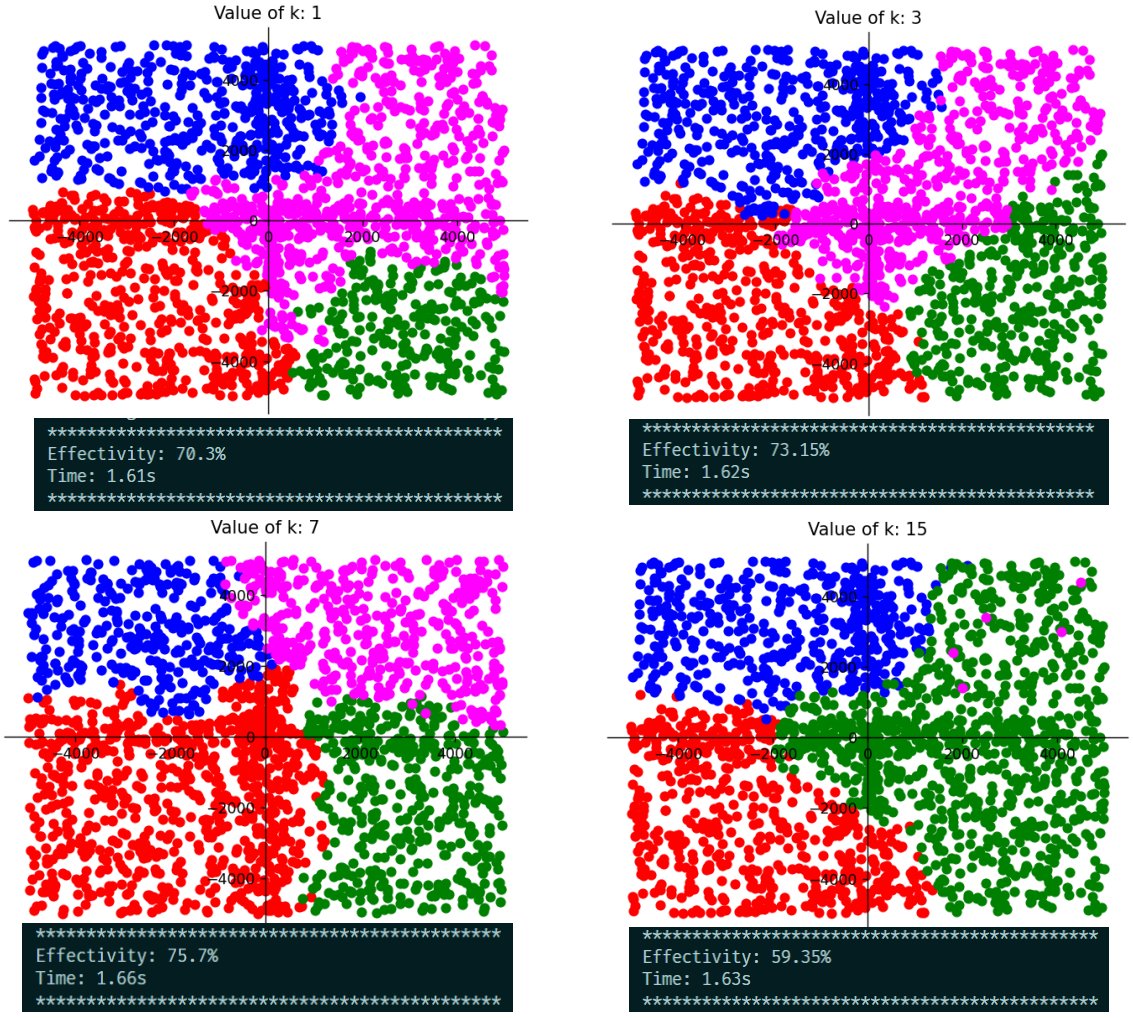
Počas testovania sa kládol dôraz hlavne na rýchlosť vykonávania algoritmu a percentuálnu úspešnosť vykonaných klasifikácií.

3.1 Spôsob testovania

Testovanie prebiehalo okrem konečného počtu 40000 bodov aj s menšími množstvami, menovite s 2000, 5000 a 10000 bodmi. Taktiež bolo vyskúšaných niekoľko rôznych spôsobov klasifikácie nových bodov s cieľom zvýšenia efektivity algoritmu.

3.1.1 Testovanie algoritmu s 2000 bodmi

S takýmto nízkym počtom bodov sa neobjavujú žiadne zásadné problémy týkajúce sa času vykonávania algoritmu. Klasifikácia prebehne v priemere za 1.6 sekúnd a jej úspešnosť sa pre k z intervalu {1, 3, 7} pohybuje v rozmedzí od 70% do 80%, pričom pre k = 15, prirodzene, klesá.



Obr. 3.1 Výsledky klasifikácie pre 2000 bodov

Podľa údajov zaznamenaných na obr. 3.1 aj v tabuľke 3.1 je evidentné, že najefektívnejší výstup dostaneme v prípade k = 7, pričom čas vykonávania v každom prípade zostáva skoro identický. Naopak, pre k = 15 sa vo väčšine prípadov vyskytnú zkreslené výsledky spôsobený príliš veľkým počtom najbližších bodov na porovnanie.

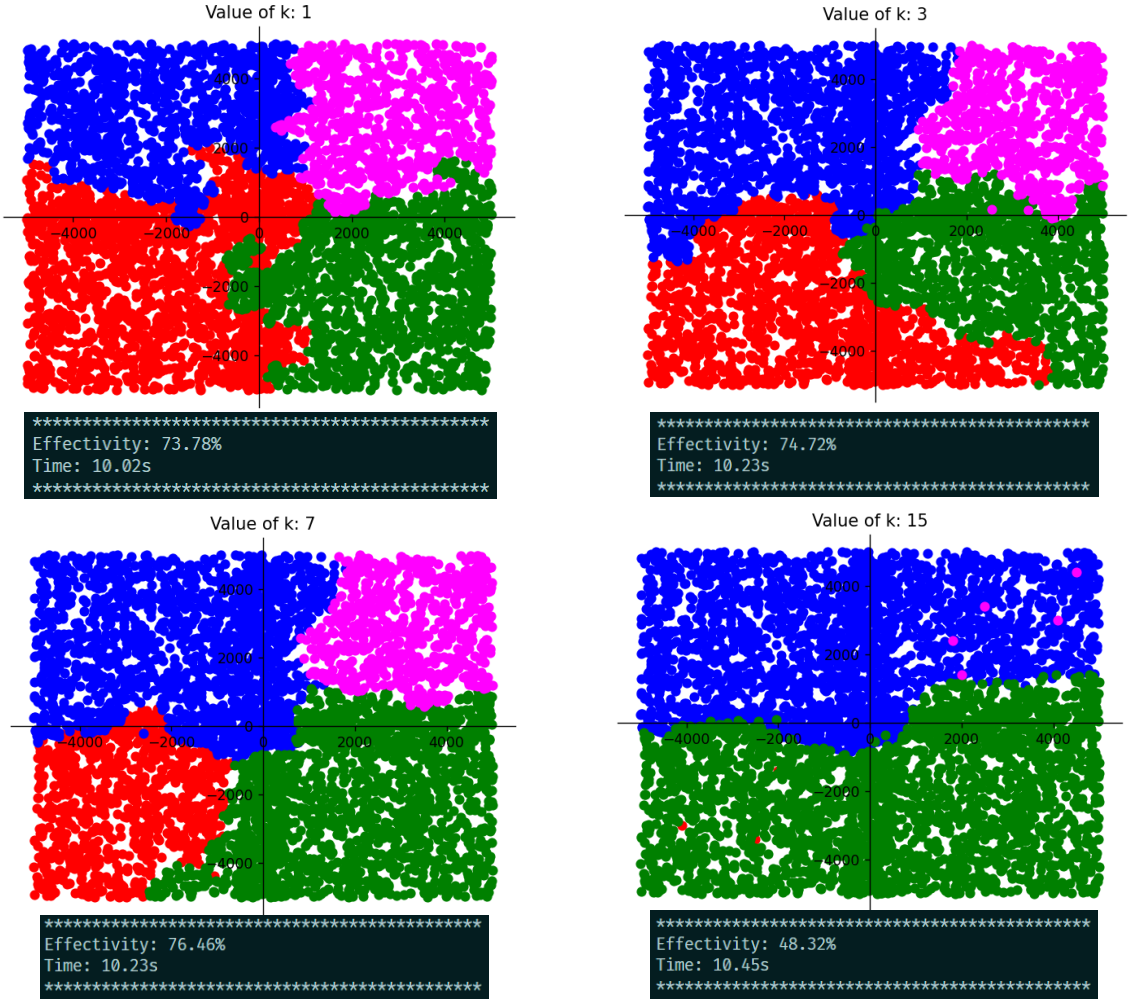
20 behov s 2000 bodmi

k	čas	efektivita
1	1.59s	72.15%
3	1.62s	74.61%
7	1.63s	75.27%
15	1.69s	54.89%

Tab. 3.1 Priemer 20 behov pre klasifikáciu s 2000 bodmi

3.1.2 Testovanie algoritmu s 5000 bodmi

Uvedený počet bodov tiež nespôsobuje príliš veľké problémy ohľadom času vykonávania, ale začína sa objavovať mierna nelinearita v porovnaní ku klasifikácii s 2000 bodmi. Miera úspešnosti sa taktiež skoro vôbec nelíši od predchádzajúceho testu.



Obr. 3.2 Výsledky klasifikácie pre 5000 bodov

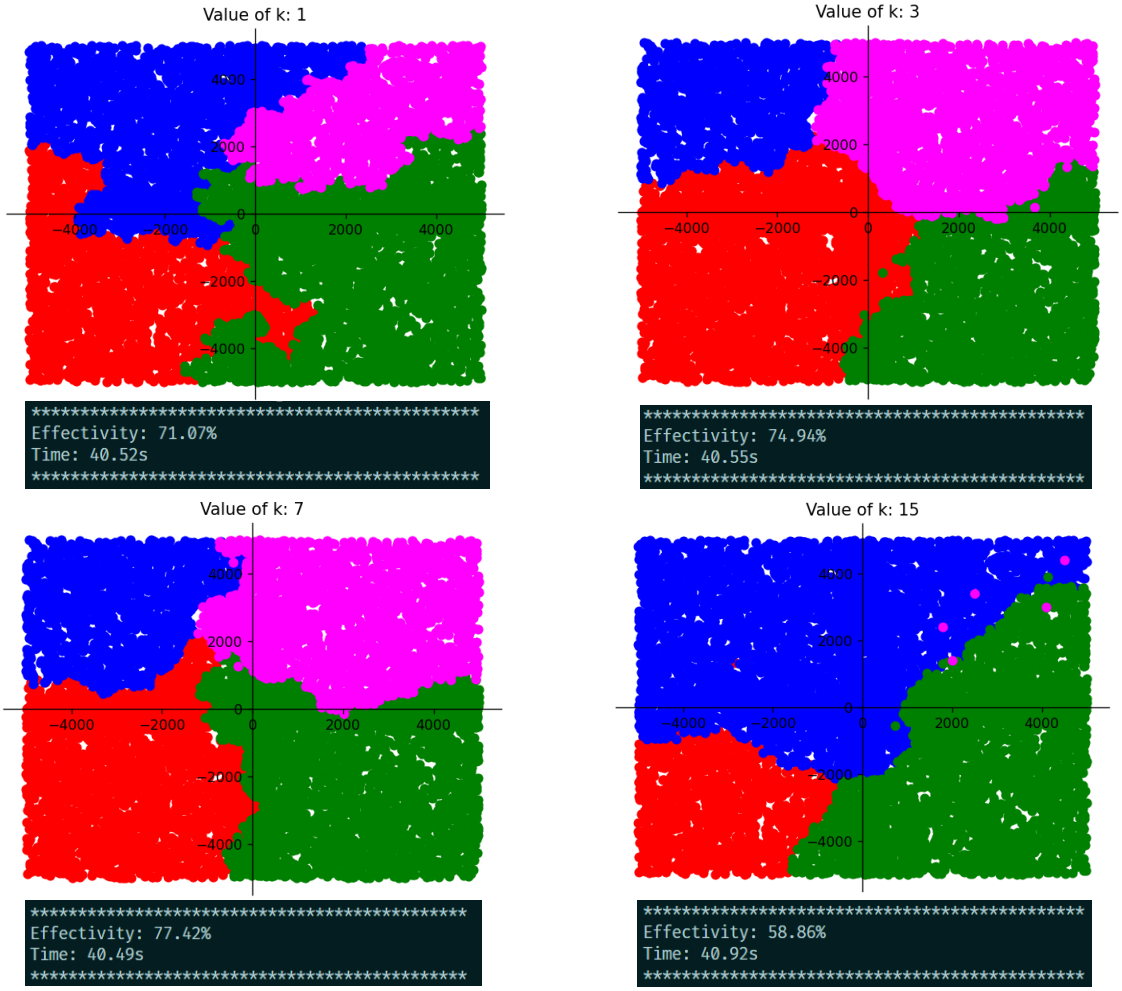
Aj v tomto prípade sa najefektívnejším ukázala byť možnosť s k = 7. Začínajú sa taktiež objavovať mierne rozdiely v časoch vykonávania medzi jednotlivými hodnotami k, pričom efektivita zostáva na podobnej úrovni. Spomenutý trend je možné vidieť aj na údajoch zaznamenaných v tabuľke 3.2.

20 behov s 5000 bodmi		
k	čas	efektivita
1	10.17s	73.63%
3	10.29s	74.87%
7	10.33s	77.30%
15	10.45s	52.11%

Tab. 3.2 Priemer 20 behov pre klasifikáciu s 5000 bodmi

3.1.2 Testovanie algoritmu s 10000 bodmi

Pri takomto počte bodov sa začína výraznejšie prejavovať problém s časom vykonávania (počet bodov oproti prechádzajúcemu testu je dvojnásobný, ale čas vykonávania sa až štvornásobil). Naopak, miera úspešnosti sa s pribúdajúcimi bodmi mierne zvyšuje.



Obr. 3.3 Výsledky klasifikácie pre 10000 bodov

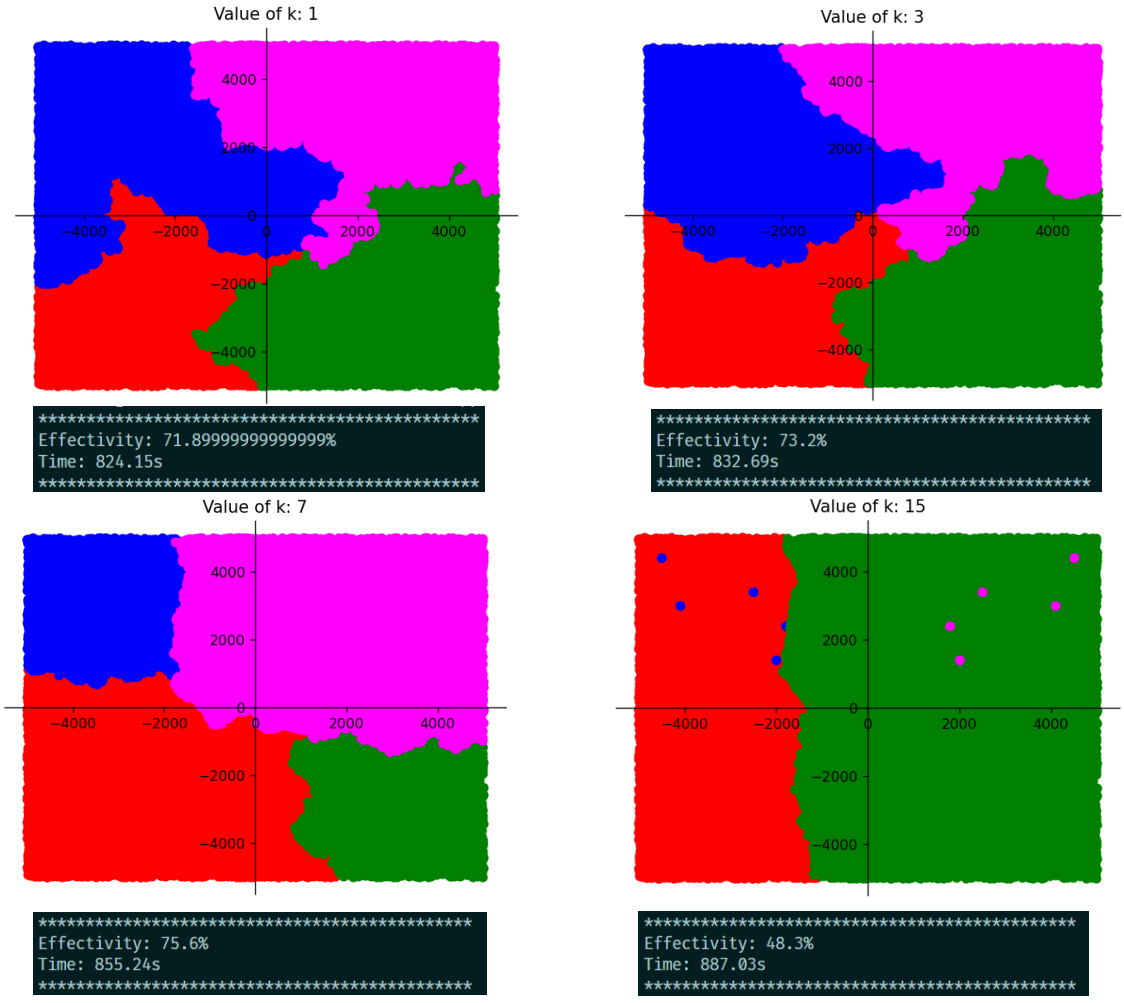
Ako už bolo spomenuté, efektivita klasifikácie pre jednotlivé hodnoty k ukazuje stúpajúcu tendenciu v spojení so zvyšujúcim sa počtom bodov. Opäť sa ukázala nadradenosť k = 7 oproti iným hodnotám. Kvôli dlhšiemu času vykonávania a len malému zlepšeniu efektivity sa však klasifikácia vyššieho počtu bodov nemusí oplatiť.

20 behov s 10000 bodmi		
k	čas	efektivita
1	40.26s	73.58%
3	40.25s	75.12%
7	40.49s	78.51%
15	40.84s	58.37%

Tab. 3.3 Priemer 20 behov pre klasifikáciu s 10000 bodmi

3.1.3 Testovanie algoritmu so 40000 bodmi

Tento test je základom celej úlohy. Časové nároky vykonania testu dosahujú desiatky minút s minimálnym zlepšením efektivity, kvôli čomu sa nepovažuje za dobrú voľbu. Test bol vykonaný štyrikrát, raz pre každú hodnotu k.

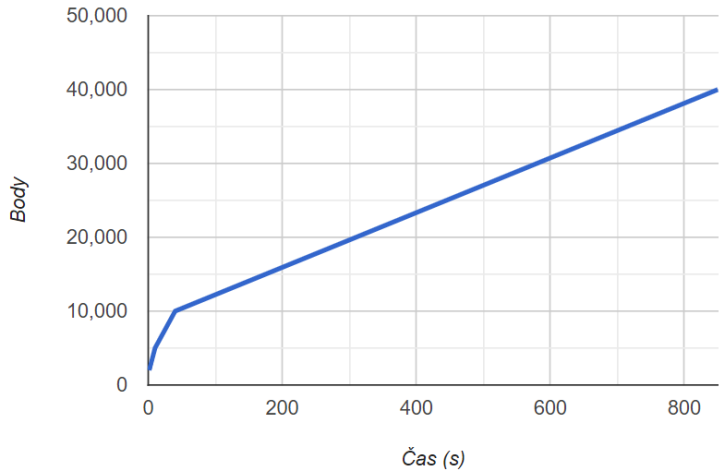


Obr. 3.4 Výsledky klasifikácie pre 40000 bodov

3.2 Zhodnotenie testovania

Počas testovania s rozličnými počtami bodov bolo zistených niekoľko zaujímavých a dôležitých skutočností. Zistilo sa, že aj relatívne nízky počet bodov (do 5000) dokáže poskytnúť reprezentatívne výsledky a kvôli mnohonásobne kratšiemu času vykonávania sa môže považovať za vhodnejšie riešenie oproti vysokému počtu bodov. Naopak, efektivita riešenia vo veľkej miere nezáleží od počtu analyzovaných bodov, ale skôr od hodnoty k.

Kvôli ukladania bodov do jednoduchého zoznamu sa čas vykonávania exponenciálne zväčšuje s narastajúcim počtom bodov. Táto skutočnosť je zobrazená na obr. 3.5.



Obr. 3.5 Graf závislosti počtu klasifikovaných bodov od času

Dôležitým poznatkom je aj zistenie, že pri každom teste, hlavne pri 40000 bodoch, je hodnota 15 pre k priveľmi vysoká a výsledky sú nepresné. Existuje tu veľká šanca, že kvôli vysokému počtu zvolených najbližších bodov budú v prevahe vzdialenejšie body inej kategórie a z tohto dôvodu sa nový bod nesprávne klasifikuje. Najvhodnejšie hodnoty pre k sú 3 a 7, kde nastane optimálne porovnanie a vo väčšine prípadov sa bod správne klasifikuje. Problémy nastávajú ibe v strede dvojdimenzionálneho priestoru (rozsahy súradnicových ôs približne od -1000 do 1000), kde sa môžu jednotlivé farebné kategórie prekrývať a tým znižovať efektivitu.

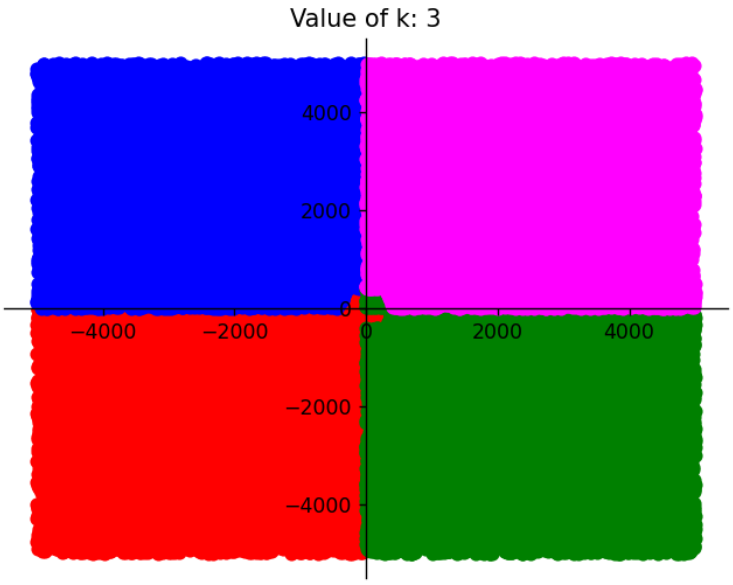
Z pohľadu časových nárokov sa za najoptimálnejšiu hodnotu pre k môže považovať 1, pretože je najrýchlejšie (z dôvodu najmenšieho počtu porovnaní) a výsledky sú aj vedľa hodnôt 3 a 7 prijateľné. Medzi jednotlivými hodnotami k však ani pri vysokom počte bodov nie sú obrovské zmeny v čase vykonávania (medzi k = 1 a k = 15 pri 10000 bodoch maximálne pol sekundy, pri 40000 bodoch približne 45-50 sekúnd).

3.3 Možnosti rozšírenia a optimalizácie riešenia

Jedným z možných rozšírení programu by mohlo byť umožnenie používateľovi manuálne zadávať hodnotu generovaných bodov a hodnoty k (momentálne je počet bodov konštantne stanovených na 40000 a zoznam hodnôt k obsahuje iba množinu {1, 3, 7, 15}).

Čas potrebný na vykonanie klasifikácie pri vyšších počtoch bodov by sa dokázal rapídne znížiť použitím vhodnejšej dátovej štruktúry na ukladanie už klasifikovaných bodov. Momentálne sú body ukladané do jednoduchého lineárneho zoznamu (poľa), kvôli čomu musí algoritmus pri pridaní nového bodu vypočítať jeho vzdialenosť od všetkých zvyšných bodov. Ak by ale namiesto zoznamu boli implementované k-D stromy alebo Ball-Tree stromy, nebolo by potrebné počítať vzdialenosť všetkých bodov, čo by mohlo výrazne zrýchliť algoritmus.

Bolo tiež zistené, že efektivita riešenia by sa dala zvýšiť jednoduchou zmenou v algoritme; namiesto porovnávania kategórií s k najbližšími bodmi by boli nové body porovnávané iba s prvými 20 staticky zadefinovanými bodmi (obr. 3.6). Výsledky po časovej stránke aj stránke efektivity sú povzbudivé: algoritmus vykoná klasifikáciu za oveľa kratší čas (12.7 sekúnd oproti cca. 850 pri 40000 bodoch) a s vyššou efektivitou (83.46%). Treba však podotknúť, že uvedené informácie platia iba v prípade, ak je prvých 20 bodov rozmiestnených rovnomerne, ako v zadaní. Pri iných aplikáciách by tento spôsob pravdepodobne nefungoval.



Obr. 3.6 Výsledok algoritmu s pozmeneným algoritmom