SLOVENSKÁ TECHNICKÁ UNIVERZITA V BRATISLAVE
FAKULTA INFORMATIKY A INFORMAČNÝCH TECHNOLÓGIÍ

Umelá inteligencia

Zadanie 4

Klasifikácia

Martin Melišek

Cvičenie: Streda 16:00

Cvičiaci: Ing. Ivan Kapustík

2020/2021

Klasifikácia

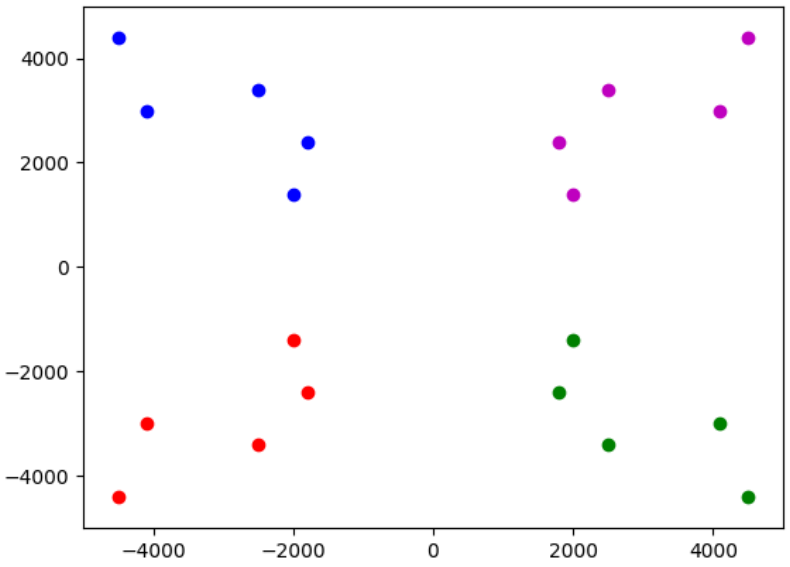
Úvod

Zadaním tohto projektu je implementovať **klasifikátor** pre nové body, ktoré sú postupné pridávané do dvojrozmerného priestoru. Spočiatku priestor obsahuje 20 predefinovaných bodov rozdelených do **štyroch tried**, pričom tieto triedy sú: RED(R), GREEN(G), BLUE(B), PURPLE(P). Pomocou **k-NN** algoritmu budeme klasifikovať všetky ďalšie pridané body. Experiment vykonáme viac krát s rôznymi hodnotami k a na záver, na vizualizáciu a testovanie priestor **vykreslíme**.

Pôvodných 20 bodov má súradnice pevne dané:

No.	RED		GREEN		BLUE		PURPLE	
1.	-4500	-4400	4500	-4400	-4500	4400	4500	4400
2.	-4100	-3000	4100	-3000	-4100	3000	4100	3000
3.	-1800	-2400	1800	-2400	-1800	2400	1800	2400
4.	-2500	-3400	2500	-3400	-2500	3400	2500	3400
5.	-2000	-1400	2000	-1400	-2000	1400	2000	1400

Tabuľka 1: Súradnice počiatočných bodov. Hodnoty sú uvedené ako súradnice X a Y



Obrázok 1: Počiatočné rozdelenie bodov

Testovacie prostredie

Celkovo pridáme spolu **40 000** nových bodov a klasifikovať budeme na základe k susedov. k predstavuje počet najbližších bodov v priestore. V projekte sa zameriame na nasledovné hodnoty k: $1, 3, 7, 15$. Nové body budú každé z inej triedy, s tým, že sa budú cyklicky opakovať. Celkovo 10 000 z každej triedy a dvom bodom vygenerovaným po sebe neprislúcha rovnaká trieda. Súradnice bodov generujeme náhodne, ale s pravdepodobnosťou: Každý bod má na 99% nasledujúce súradnice na základe jeho farby.

	RED	GREEN	BLUE	PURPLE
X	[-5000, 500]	[-500, 5000]	[-5000, 500]	[-500, 5000]
Y	[-5000, 500]	[-5000, 500]	[-500, 5000]	[-500, 5000]

Tabuľka 2: Súradnice nových bodov

S pravdepodobnosťou 1% bude mať nový bod náhodné súradnice z celého priestoru [-5000, 5000].

Implementačné prostredie

Program predstavuje konzolovú aplikáciu, ktorá bola vytvorená v programovacom jazyku **Python v3.8.5 64bit**. Spúšťa sa cez Command Line, PowerShell alebo iný CLI nástroj. Počas celého vykonania programu používame nasledujúce importované moduly.

- timeit
  - Meranie času vykonávania algoritmu
- random
  - Náhodné generovanie celého čísla z rozsahu
- copy
  - Duplikovanie Python objektov
- math
  - Matematické funkcie pre výpočet Euklidovej vzdialenosti
- matplotlib.pyplot
  - Vykreslenie ilustrácie výsledku klasifikátora
- gc
  - Explicitné zavolanie Python garbage collectora po vykreslení grafu
- concurrent.futures
  - Multiprocessing na spúšťanie algoritmu pre rôzne hodnoty k.

Štruktúra zdrojových súborov

- main.py
  - Hlavný zdrojový súbor, tu sa program spúšťa.
  - Inicializácia potrebných tried a spustenie klasifikácie
- generator.py
  - Generovanie 2D priestoru a náhodných bodov.
- kd_strom.py
  - Reprezentácia priestoru pomocou KD stromu.
- knn.py
  - Algoritmus na nájdenie najbližších susedov.
- klasifikator.py
  - Trieda, v ktorej beží hlavný cyklus klasifikácie
- plot.py
  - Vykreslenie ilustrácie výsledku klasifikátora.
- utils.py
  - Základné triedy, výpočet vzdialenosti, načítanie vstupu a výpis.
- vstup.txt
  - Zoznam počiatočných bodov

Analýza

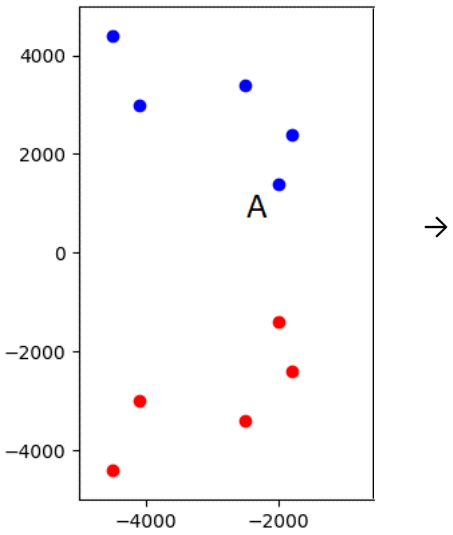
Hlavnou podstatou tejto práce je implementácia funkcie

```
classify(x, y, k)
    returns klasifikovana_trieda
```

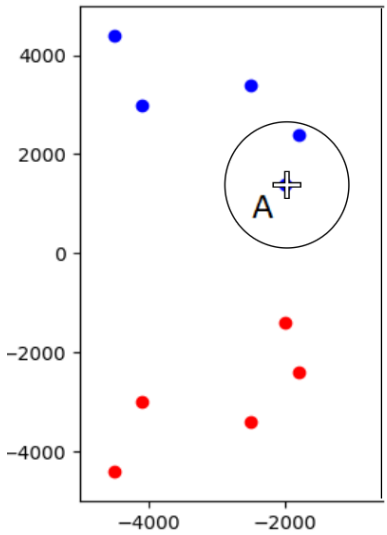
Táto funkcia dostane ako parametre súradnice nového bodu a hodnotu k → počet susedov na základe, ktorých sa má rozhodnúť do akej triedy bod zaradí. Návratová hodnota funkcie je teda trieda, ktorú porovnáme s pôvodnou triedou bodu a ak sa zhodujú považujeme klasifikovanie daného bodu za **úspešné**. Úspešnosť klasifikovania si ukladáme a na konci vykonávania programu vyhodnotíme celkovú úspešnosť a dobu trvania.

Na to aby sme boli schopný klasifikovať nové body potrebujeme si premyslieť **stratégiu vyhľadávania najbližších susedov**. Človek sa dokáže na tento problém pozrieť a okamžite identifikovať body, ktoré sú najbližšie k nášmu bodu, ktorý skúmame:

Uvažujme o nasledovnom priestore. Ak by sme pridávali bod A, vedeli by sme si predstaviť okolo tohto bodu kružnicu, ktorú by sme postupne rozširovali až by sme pokryli k susedov



Obrázok 2: Pridanie nového bodu do priestoru



Obrázok 3: Vykreslenie najbližšieho suseda

Takto by sme boli schopný rýchlo nájsť najbližšie body.

Skúsme sa zamyslieť nad tým, ako by sme to mohli riešiť algoritmicky. Nakoľko nie je možné si ukladať celý priestor bodov a skúmať, či sa v určitom offsete nenachádza ďalší bod, potrebujeme rátať vzdialenosti medzi bodmi. Naivné riešenie pomocou *brute force* skúšania všetkých bodov a počítanie vzdialeností od nového bodu ku ostatným, by nám výsledok síce prinieslo a jeho implementácia je triviálna, ale musíme sa zamyslieť nad časovou efektivitou. Tento spôsob má zložitosť $\boldsymbol{O(n^2)}$, kde n je počet bodov. Vždy po nájdení vzdialeností ich zoradíme $\boldsymbol{O(n \cdot \log n)}$ a následne nájdeme najčastejšie opakujúcu sa triedu medzi prvými k bodmi. Pri datasete 40020 bodov je toto riešenie neuspokojivé a nevyhovujúce.

Na urýchlenie tohto procesu použijeme jednu z techník rozdeľovania priestoru a zavedieme dátovú štruktúru s názvom ***K-D TREE*** (k-d strom).

Reprezentácia údajov

K-D STROM

k-d stromy sú špeciálny typ binárnych stromov, pri ktorých každý vrchol rozdeľuje priestor na dve časti na základe toho, v akej hĺbke sa nachádza[1]. Náš problém je vyjadrený v 2D priestore, takže budeme používať dvoj dimenzionálne stromy.

Trieda reprezentujúca strom sa volá `KdStrom` a obsahuje základné funkcie ako pridanie `insert(new_point)` a hľadanie `search(point)`. Základná myšlienka vkladania bodu do stromu je rovnaká ako pri binárnych stromoch, rekurzívne sa vnárame a následne vložíme nový bod tam kam patrí. Je možné implementovať strom, tak že samotné body budú len v listoch a vrcholy, ktoré nie sú listy si budú niesť iba hodnotu, na základe ktorej rozdeľujú priestor[2]. V našom projekte budú body **všetky body vrcholy stromu**.

```
class Point(list):
    def __init__(self, bod, farba):
        self.left = None
        self.right = None
        self.farba = farba
        list.__init__(self, [bod[0], bod[1]])
```

Obrázok 4: Reprezentácia vrcholu v strome

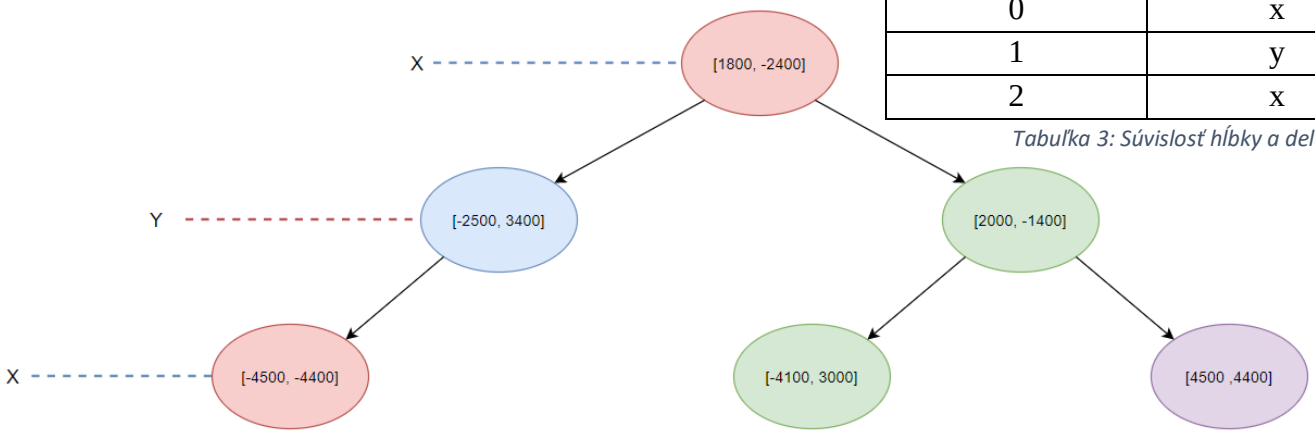
Klasicky potrebujeme pointer na *ľavý* a *pravý* podstrom. Farbu, teda triedu do akej bod patrí a jeho súradnice, ktoré sú reprezentované ako elementy objektu list a preto aj trieda `Point` dedí z triedy `List` a môžeme ju považovať ako pole s *custom* atribútmi.

Ak začíname od prvého vloženého bodu, (koreňa – root) v hĺbke **0**, tak delíme body v **ľavom** podstrome na vrcholy s **menšou** hodnotou súradnice X a v **pravom** podstrome na vrcholy s **väčšou** súradnicou X. Vnorením sa do jedného z podstromov zvýšime hĺbku o 1 a teda zmeníme deliacu dimenziu na Y. Vykonáme operáciu $dim = hlbka \,\%\, 2$ a jednoduchým porovnaním $new\_point[dim] < root[dim]$, postupujeme ďalej.

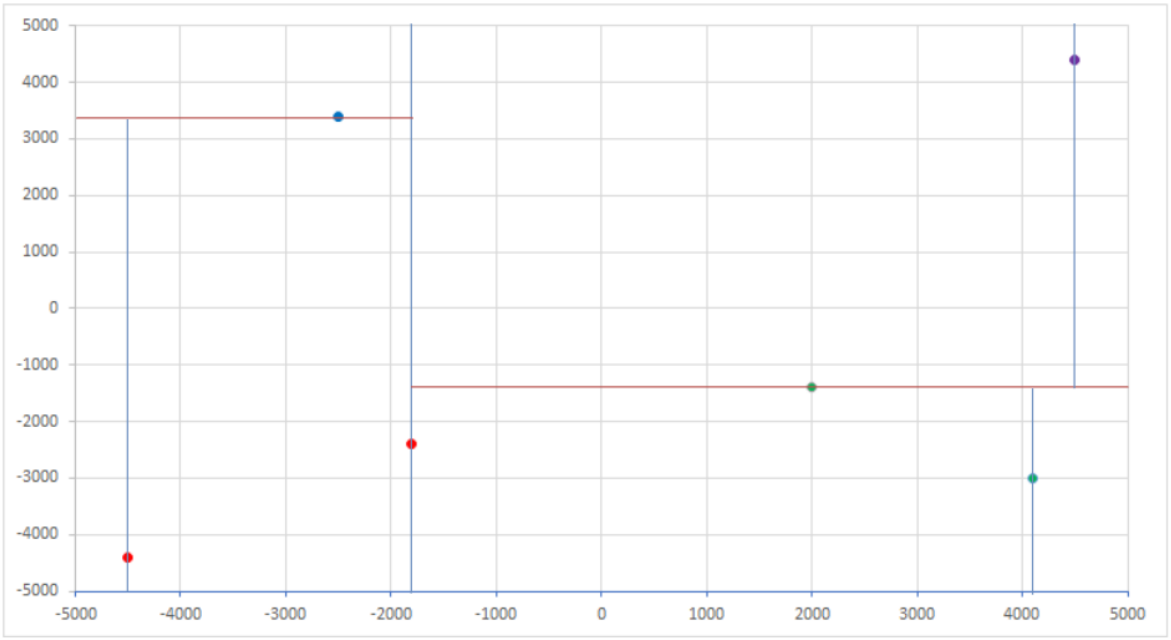
Uvažujme o zjednodušenej časti nášho stromu vytvoreného z niektorých počiatočných bodov.

Hĺbka	Deliaca dimenzia
0	x
1	y
2	x

Tabuľka 3: Súvislosť hĺbky a delenia



Obrázok 5: Vizualizácia k-d stromu
Inšpirované [3]



Obrázok 6: Vizualizácia v 2D priestore, inšpirované [3]

Koreň, teda prvý pridaný bod tohto stromu je červený $[-1800,2400]$, ktorý nám priestor rozdelí na dve polovičky, body so súradnicou **X** menšou ako -1800, sú od neho vľavo a body s väčšou vpravo. Celkovo si môžeme všimnúť na obr. 5, že sú od neho vľavo dva body jeden červený $[-4500,-4400]$ a jeden modrý $[-2500,3400]$, čo vychádza aj na vizualizácii na obr. 6. Podobne ak si zoberieme zelený bod $[2000,-1400]$, ktorý priestor rozdeľuje na základe osy **Y**, pod sebou má bod $[4100,-3000]$, čo je jeho ľavé dieťa a nad sebou fialový [4500,4400], čo je jeho pravé dieťa.

Takýmto postupným rozdeľovaním priestoru dokážeme vyhľadávať oveľa **rýchlejšie** ako keby sme museli kontrolovať všetky vzdialenosti. Pokiaľ by sa nám podarilo tento strom udržať vyvážený mohli by sme v ňom hľadať v logaritmickom čase. Bohužiaľ bežné rotácie pri samovyvažovacích napr. AVL stromoch nemôžeme použiť inak by sa porušila podmienka delenia priestoru. Vyvažovanie stromu môžeme simulovať efektívnym vkladaním podľa mediánov bodov danej rozdeľovacej dimenzie[3]. Keďže nepoznáme aké body sa budú generovať, (jedná sa o náhodné body) **nemáme možnosť zachovať strom vyvážený** a eventuálne určite začne byť krivý. Toto nám trocha uškodí pri hľadaní susedov, ale celková zložitosť (teoreticky $O(h)$, kde h je hĺbka stromu) ostane určite lepšia ako keby to bol iba zoznam.

Opis použitých algoritmov

Inicializácia

Na to aby sme mohli spustiť všetky klasifikátory naraz vygenerujeme v triede `Generator` náhodné body. Už pri generovaní týchto bodov zostavíme prvý k-d strom, ktorý nám umožní efektívne kontrolovať, či sme nevygenerovali duplicitný bod, ktorý už bol vytvorený predtým. Do triedy `Klasifikator` pošleme ako parameter zoznam $40\,000 + 20$

bodov a hodnotu k. Vytvoríme počiatočný priestor prvotných bodov ako nový k-d strom s veľkosťou 20 a spustíme hlavný cyklus klasifikácie.

Klasifikácia

Súradnice nových bodov posielame do funkcie `classify(..)`, v ktorej nájdeme k najbližších bodov, z ktorých vypočítame **modus** (najčastejšie opakujúcu sa triedu) a vrátime klasifikovanú triedu. Rozšírime k-d strom, tak že pridáme tento bod a nastavíme mu triedu, takú akú vrátila funkcia `classify(..)`. Ak sa trieda vrátená a pôvodná zhodujú zvýšime úspešnosť.

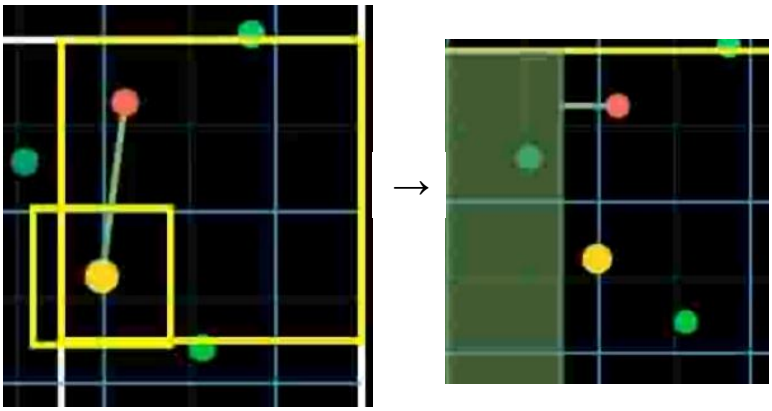
Hľadanie *k* najbližších susedov

Funkcia `KdStrom.najdi_susedov(point, k)` inicializuje k-NN algoritmus,
`returns susedia : List`
ktorý sa nachádza vo vlastnej triede `KNN`. Tento algoritmus začína v koreni k-d stromu a vnára sa **rekurzívne**, rovnako ako keby do stromu vkladal nový bod – na základe deliacej dimenzie sa posúva do ľavého/pravého podstromu. Nakoľko všetky vrcholy sú aj body v strome, potrebujeme skontrolovať vzdialenosť medzi bodom v aktuálne navštívenom vrchole a bodom, ku ktorému hľadáme susedov. Skutočnosť, že výsledkom hľadania má byť k susedov, si vyžaduje udržiavať určitú dátovú štruktúru, ktorá združuje zatiaľ všetkých k najbližších nájdených bodov. Nachádzajú sa v nej inštancie triedy `Vzdialenost`, ktorá má atribúty `Point` a `distance` k nemu. Možností akú štruktúru zvoliť je niekoľko:

- Jednoduché **pole**, ktoré zoraďujeme po každom pridaní nového prvku s tým, že odstránime najvzdialenejší pomocou funkcie `list.pop()`. Tento zoznam si celý čas udržiavame usporiadaný, čiže optimalizované algoritmy Pythonu `list.sort()`, to zvládnu v lineárnom čase.
- Max-Heap – binárna **halda**, v ktorej k-ty najvzdialenejší bod je udržiavaný v koreni. Nové vzdialenosti porovnávame s najhoršou a ak sme našli lepšiu, vykonáme operáciu `heap.pushpop`(Vzdialenost(bod,distance)), ktorá odstráni koreň, čiže najvzdialenejšieho suseda a pridá nového. Nakoľko pridávame nový bod, musíme zachovať haldovú vlastnosť a vykonať `heapify()`[4].

Empirická skúsenosť ukázala, že v tejto implementácií nezáleží na tomto výbere a preto pre prehľadnosť použijeme **pole**.

Akonáhle sa dostaneme k prvému listu stromu, dostali sme sa k v vrcholu, kam by sme mali **vložiť** nový bod. Namiesto vloženia, sa začneme **vynárať** z rekurzie smerom nahor a pri každom bode v spätnom chode kontrolujeme, či by nemohli existovať vrcholy v danej časti stromu, ktoré sú **bližšie než doteraz nájdené**. Predsa len skutočne sa môže stať, že k novému bodu sa bližšie nachádza bod z iného „obdĺžnika“(časti nadroviny priestoru), ktorý sme ešte **nenavštívili**. Teraz nepočítame Euklidovu vzdialenosť, ale vypočítame **vzdialenosť deliacich osí** nového bodu a prezeraného bodu. Ak je táto vzdialenosť menšia než k-ty najbližší suseda musíme navštíviť aj tento podstorm. Naopak ak nie je, celú túto vetvu stromu môžeme odignorovať, čím vylepšíme časovú zložitosť.



Obrázok 7: Ilustrácia hľadania NN pre k=1[3][5]

V prvotnom vnorení sme ako najbližšieho suseda určili žltý bod.

Následne musíme skontrolovať aj ľavú vetvu tohto podstromu, nakoľko vzdialenosť k X-ovej deliacej osy je nižšia než vzdialenosť k žltému bodu. Týmto procesom sa dopracujeme až k najbližšiemu susedovi vo vedľajšom obdĺžniku

```
def search(self, root, point, hlbka):
    if root is None:
        return
    self.check_distance(root, point)
    dim = hlbka % 2
    if point[dim] < root[dim]:
        self.search(root.left, point, hlbka + 1)
        if math.fabs(root[dim] - point[dim]) < self.susedia[-1].distance:
            self.search(root.right, point, hlbka + 1)
    else:
        self.search(root.right, point, hlbka + 1)
        if math.fabs(root[dim] - point[dim]) < self.susedia[-1].distance:
            self.search(root.left, point, hlbka + 1)
```

Obrázok 8: Kód funkcie hľadania k-NN

- `root` – vrchol práve prezeraného podstromu
- `point` – nový bod, ku ktorému hľadáme k-NN
  (k nearest neighbours → k najbližších susedov)
- `hlbka` – aktuálna hĺbka v strome, značí poradie splitovacej dimenzie.

Najhorší doteraz nájdený *k*-ty suseda sa nachádza na poslednom indexe poľa susedia.
V procedúre `check_distance(root, point)` kontrolujeme nasledovné záležitosti:

- Ak sme ešte nenašli *k* susedov, automaticky navštívený vrchol pridáme do poľa.
- Ak je vzdialenosť menšia než najhorší doteraz nájdený suseda, do poľa appendneme prezeraný vrchol, usporiadame ho podľa vzdialeností vzostupne a odstránime posledný bod v zozname.

Proces opakujeme až pokým, algoritmus nedokončí kontrolu pre koreň celého stromu, kedy sa hľadanie ukončí a vrátime zoznam *k* najbližších susedov.
Po klasifikovaní všetkých náhodných bodov klasifikátor vráti úspešnosť, čas trvania a samotný vybudovaný strom do funkcie na ilustráciu a výpis riešenia.

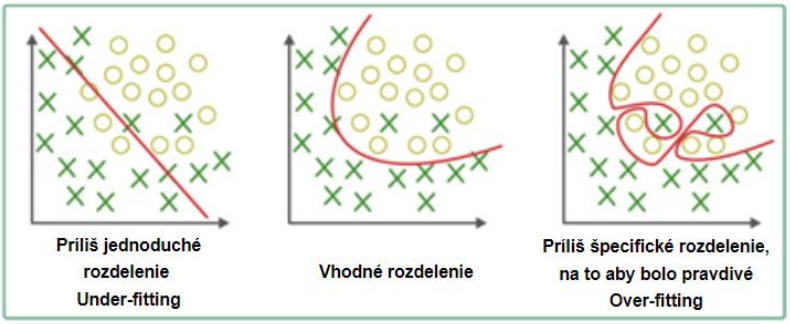
Vykreslenie priestoru

Na vykreslenie bodov v grafe sme použili importovaný Python modul *matplotlib.pyplot*. Do funkcie `utils.plot_2d(..)` sme poslali všetky dôležité dáta a pomocou in-order prehľadávania skonvertovali strom na zoznam. Jednotlivé body sme na základe ich triedy vykreslili cez funkciu `plt.plot(x, y, farba)`.

Testovanie

Analýza

Na to aby model strojového učenia bol účinný a dal sa považovať za dobrý musí zvládnuť **generalizovať** akékoľvek nové vstupy, čím v budúcnosti dokáže vytvoriť predikcie pre dáta, ktoré ešte predtým nikdy nezanalyzoval[6]. Takýto návrh budeme potrebovať pri rozhodovaní sa o vhodnom zvolení hodnoty k v našom algoritme. Pozrime sa na vizuálnom príklade ako sa šum v dátach môže prejavovať:



Obrázok 9: Generalizácia hraníc a šum v dátach[6]

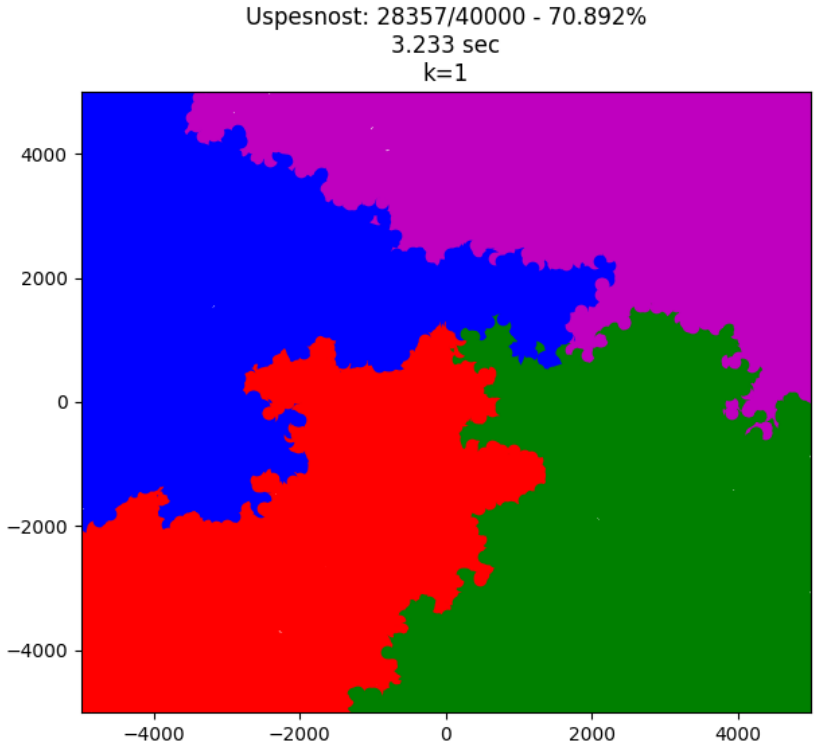
Na obr. 9, môžeme pozorovať algoritmus, ktorý má vytvoriť krivku, ktorá oddelí krúžky od krížikov. Úplne vľavo je krivka priveľmi jednoduchá a nedokáže korektne zachytiť štruktúru dát. Stredný obrázok by bola lepším ukazovateľom daného stavu. Napravo, aj keď to môže vyzerať, že algoritmus vyprodukoval krivku, ktorej sa podarilo korektne odseparovať dáta, tento výsledok je priveľmi konkrétny a „šitý na mieru“ len na tento jeden dataset.
V reálnych dátach nemusí správne vyhodnotiť situáciu, ak by bol prítomný šum[6] (niektoré symboly sú v nesprávnom klastri).

V našom príklade budeme pozorovať tieto úkazy pri rôznych hodnotách k.

Vyhodnotenie experimentu

Klasifikátor sme testovali so 4 hodnotami k → 1, 3, 7, 15. Do pôvodného datasetu 20 predefinovaných bodov sme dogenerovali ďalších 40 000 bodov. Nevyplnené miesta vo vizualizácií predstavujú priestor, ktorý sme nezaplnili počas vkladania bodov do stromu. Na výsledných figúrach je toto voľné miesto ponechané aj z dôvodu, ktorý nám jednoznačne potvrdí, že vygenerované body boli rovnaké pre všetky hodnoty k. Na ploche -5000, 5000 vykresľujeme spolu 40020 bodov a ofarbujeme na základe tried.

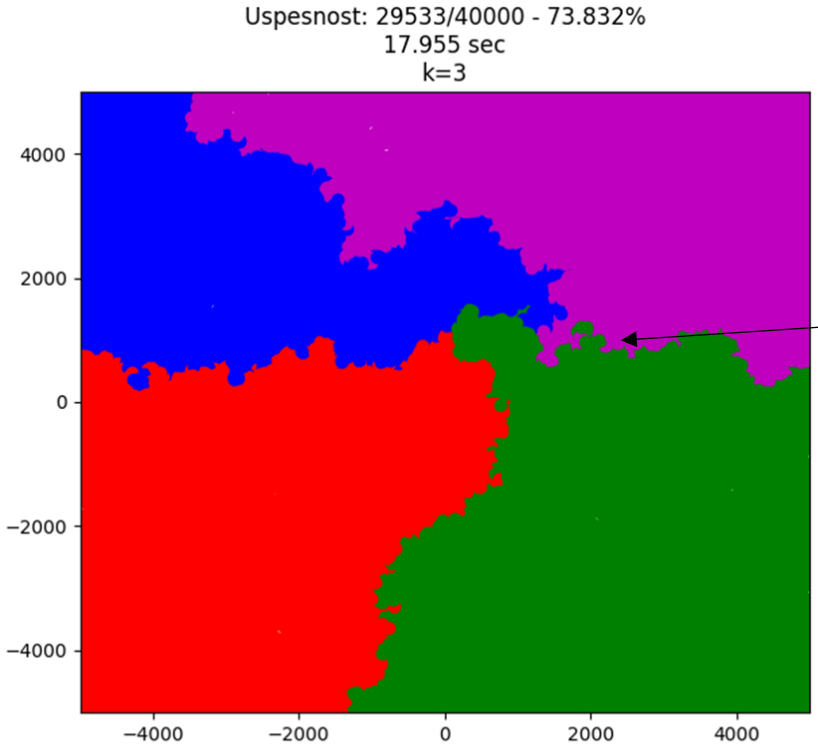
K=1



Obrázok 10: Výsledok pre k=1

Na vygenerovanom bodovom grafe pozorujeme jasný úkaz prílišnej konkretizácie hraníc jednotlivých tried. Na viacerých miestach body zasahujú jeden do druhého a celkovo hranice sa snažia byť **príliš detailné**. Tento fakt môže postupne viesť k nereálnym výsledkom. Úspešnosť je samozrejme celkom dobrá a doba vykonávania najlepšia zo všetkých, pretože vždy hľadáme iba jedného najbližšieho suseda. Nemusíme robiť operácie sort a ani veľa krát počítať zbytočne vzdialenosti, či kontrolovať veľké množstvo vetiev.

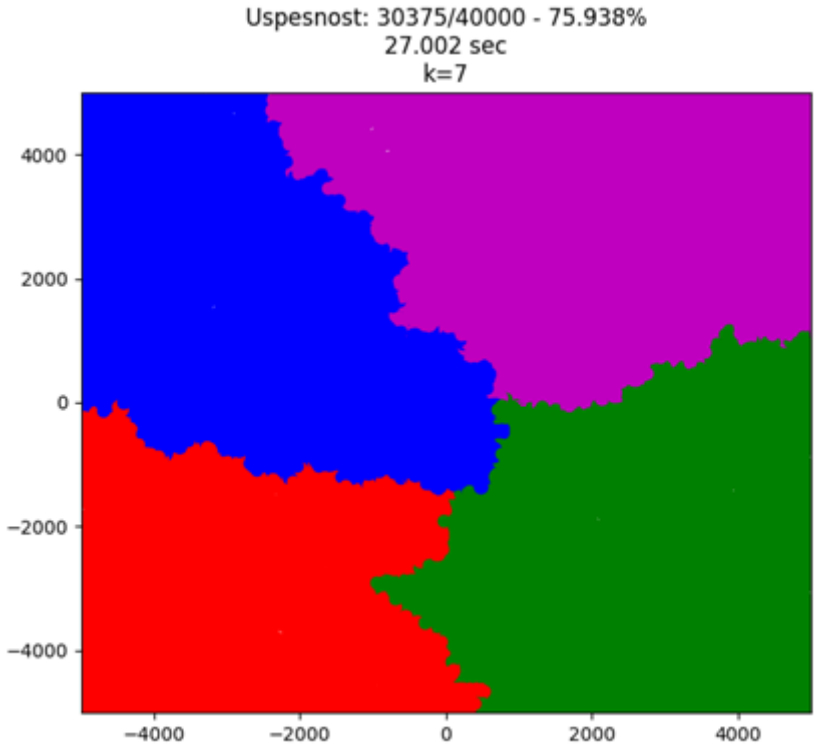
K=3



Obrázok 11: Výsledok pre k=3

Hodnota k=3 začína vylepšovať stav hraníc a aj zvyšovať percentuálnu úspešnosť. Stále môžeme pozorovať prílišnú detailnosť. Doba vykonávania sa mierne zvýšila, ale zvýšenie počtu hľadaných susedov sa ukázalo byť vhodným začiatkom.

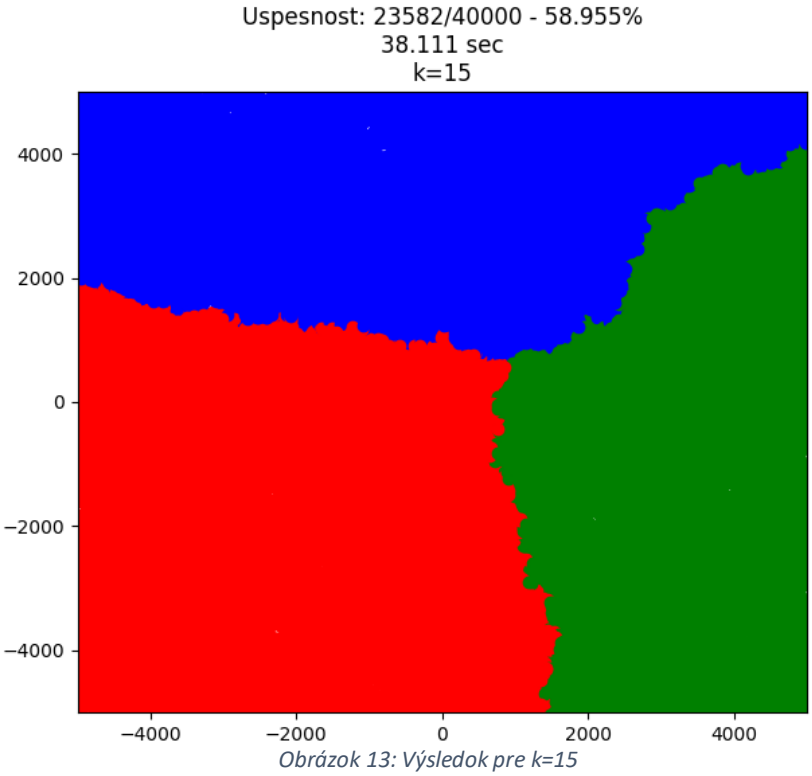
K=7



Obrázok 12: Výsledok pre k=7

Neprekvapivo jednoznačne **najlepší výsledok** sme dostali, keď sme si zvolili stredný počet nájdených susedov a klasifikáciu na základe 7 najbližších bodov. Hranice sa zdajú byť omnoho lepšie a robustnejšie ako pri menších hodnotách k. Úspešnosť je pri k=7 najvyššia zo všetkých testov. Doba trvania je priemerná, ale ešte stále veľmi uspokojivá.

K=15



Obrázok 13: Výsledok pre k=15

Pri vysokej hodnote k, došlo k všeobecne nesprávnemu vyhodnoteniu problému, kde fialová trieda **úplne zanikla**. Za následok to má skutočnosť, že sme si vybrali veľký počet susedov, ktorý nekorešpondovali k správnemu klasifikovaniu tried a tým pádom kompletne narušili hranice. Všimnime si, že hranice a sú príliš jednoduché vymedzujú triedy veľmi lenivo.

Pri analýze sme si spomínali šum, tento model vyhodnotil aj skutočný „signál“ ako šum a odstránil ho. Celkovo nedokázal pochopiť komplexnosť problému a preto je výsledok horší.

Záver

V závere tejto práce môžeme potvrdiť, že sa nám podarilo splniť ciele kladené na začiatku a to naimplementovať **funkčný klasifikátor** bodov v dvojrozmernom priestore. Prvotná implementácia, ktorá slepo rátala všetky vzdialenosti k novo pridanému bodu sa vykonávala približne **25 minút** a preto sme zvolili adekvátnu alternatívnu, ktorá znížila časovú náročnosť približne **50x**. Využili sme pokročilé dátové štruktúry ako ***k* dimenzionálne stromy** a algoritmus na hľadanie najbližších susedných bodov v priestore. Výsledkom bolo riešenie a vykreslenie, ktoré sa podarilo vykonať v **rozumnom čase**.

Na overenie sme využili vizualizáciu bodov v priestore a tým potvrdili teóriu o analýze **modelov strojového učenia**. Hodnoty *k* sa ukázali byť korešpondujúce s tým, čo sme očakávali a všetky okrem k=15 mali úspešnosť klasifikácie nad 70%. Spočiatku sa zdal byť počet bodov priveľký, ale podrobným štúdiom danej problematiky sa nám podarilo vysporiadať aj s väčším datasetom.

Celkový výsledok odzrkadľuje čas venovaný tomuto projektu a opäť raz, môžeme využiť nadobudnuté znalosti z umelej inteligencie a strojového učenia v ďalšom pokračovaní podobných predmetov, či v praxi. Poďakovanie na záver semestra patrí tvorcom pôvodnej osnovy, sylabov a všetkých zadaní na predmete Umelá Inteligencia. Dá sa tvrdiť, že som si počas semestra z nich odniesol najviac, čo sa dalo a cesta k zložitejším riešeniam je otvorená.

Martin Melišek 13.12.2020

Zdroje:

[1] - Bentley, J. L. (1975). "Multidimensional binary search trees used for associative searching". Communications of the ACM. 18 (9): 509–517. doi:10.1145/361002.361007.
[2] - Berg, Mark de; Cheong, Otfried; Kreveld, Marc van; Overmars, Mark (2008). "Orthogonal Range Searching". Computational Geometry. pp. 95–120. doi:10.1007/978-3-540-77974-2_5. ISBN 978-3-540-77973-5.
[3] - Wikipedia contributors. "K-d tree." Wikipedia, The Free Encyclopedia. Wikipedia, The Free Encyclopedia, 2 Nov. 2020. Web. 11 Dec. 2020.
[4] - Singh, Sachin Kumar. Max Heap on k-NN https://stackoverflow.com/a/63470510/12348001. 2018.
[5] - Das, Gopal. k-d Tree and Nearest Neighbor Search https://gopalcdas.com/2017/05/24/construction-of-k-d-tree-and-using-it-for-nearest-neighbour-search. May. 2017.
[6] - Sai Nikhilesh Kasturi, Underfitting and overfitting in machine learning and how to deal with it. https://towardsdatascience.com/underfitting-and-overfitting-in-machine-learning-and-how-to-deal-with-it-6fe4a8a49dbf. 2019.

Kreslenie diagramov - draw.io