
**VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ**
**BRNO UNIVERSITY OF TECHNOLOGY**

**FAKULTA INFORMAČNÍCH TECHNOLOGIÍ**
**FACULTY OF INFORMATION TECHNOLOGY**

**ÚSTAV INFORMAČNÍCH SYSTÉMŮ**
**DEPARTMENT OF INFORMATION SYSTEMS**


# METODY HLEDÁNÍ K-NEJBLIŽŠÍCH SOUSEDŮ

**K-NEAREST NEIGHBOUR SEARCH METHODS**

**DIPLOMOVÁ PRÁCE**
**MASTER'S THESIS**


**AUTOR PRÁCE** — **Bc. MAREK CIGÁNIK**
**AUTHOR**

**VEDOUCÍ PRÁCE** — **Ing. IVANA BURGETOVÁ, Ph.D.**
**SUPERVISOR**



**BRNO 2023**

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

# Zadání diplomové práce

144246

| | |
|---|---|
| Ústav: | Ústav informačních systémů (UIFS) |
| Student: | **Cigánik Marek, Bc.** |
| Program: | Informační technologie a umělá inteligence |
| Specializace: | Softwarové inženýrství |
| Název: | **Metody hledání k-nejbližších sousedů** |
| Kategorie: | Data mining |
| Akademický rok: | 2022/23 |

Zadání:

1. Seznamte se s problematikou hledání k-nejbližších sousedů a s dostupnými metodami pro řešení tohoto problému.
2. Společně s vedoucí práce vyberte metody, kterými se budete dále zabývat.
3. Najděte vhodné implementace vybraných metod.
4. Po dohodě s vedoucí vyberte vhodné datové sady, na kterých provedete porovnání vybraných metod.
5. Navrhněte a implementujte jednoduchou aplikaci, která umožní porovnání vybraných metod.
6. Vybrané metody porovnejte z hlediska přesnosti, rychlosti a paměťových nároků.
7. Zhodnoťte dosažené výsledky.

Literatura:


- Gallego, Antonio Javier, Juan Ramón Rico-Juan, and Jose J. Valero-Mas. "Efficient k-nearest neighbor search based on clustering and adaptive k values." *Pattern Recognition* 122 (2022): 108356.
- Jiang, Liangxiao, et al. "Survey of improving k-nearest-neighbor for classification." *Fourth international conference on fuzzy systems and knowledge discovery (FSKD 2007)*. Vol. 1. IEEE, 2007.
- Dhanabal, S., and S. J. I. J. C. A. Chandramathi. "A review of various k-nearest neighbor query processing techniques." *International Journal of Computer Applications* 31.7 (2011): 14-22.


Při obhajobě semestrální části projektu je požadováno:

- Body 1 až 4.

Podrobné závazné pokyny pro vypracování práce viz https://www.fit.vut.cz/study/theses/

| | |
|---|---|
| Vedoucí práce: | **Burgetová Ivana, Ing., Ph.D.** |
| Vedoucí ústavu: | Kolář Dušan, doc. Dr. Ing. |
| Datum zadání: | 1.11.2022 |
| Termín pro odevzdání: | 17.5.2023 |
| Datum schválení: | 18.10.2022 |

# Abstrakt

V práci je popsán základní koncept algoritmu K-nejbližších sousedů a jeho vazba s lidským pojetím podobnosti objektů. Jsou rozvedeny pojmy a klíčové myšlenky jako vzdálenostní funkce nebo prokletí dimenzionality. Práce zahrnuje detailní popis metod KD-Strom, Kulovitý strom, Locality-Sensitive Hashing, Strom náhodných projekcí a rodiny algoritmů založené na grafu nejbližších sousedů. Ke každé metodě je poskytnuto vysvětlení ideje s vizualizacemi, pseudokódy a asymptotickými složitostmi. Metody byly podrobeny experimentům a byly měřeny základní i pokročilejší metriky, ze kterých byly vyhodnoceny případy vhodnosti pro jednotlivé metody.



# Abstract

The thesis describes the basic concept of the K-nearest neighbors algorithm and its connection with the human concept of object similarity. Concepts and key ideas such as the distance function or the curse of dimensionality are elaborated. The work includes a detailed description of the methods KD-Tree, Spherical Tree, Locality-Sensitive Hashing, Random Projection Tree and families of algorithms based on the nearest neighbor graph. An explanation of the idea with visualizations, pseudocodes and asymptotic complexities is provided for each method. The methods were subjected to experiments and both basic and more advanced metrics were measured and appropriate use cases for individual methods were evaluated.



# Klíčová slova

K-nejbližších sousedů, KD-Strom, Kulovitý Strom, Strom náhodných projekcí, KNN graf, HNSW, NNDescent, Locality-Sensitive Hashing, KNN, LSH

# Keywords

K-nearest neighbors, KD-Tree, Ball-Tree, RPTree, KNNGraph, HNSW, NNDescent, Locality-Sensitive Hasning, KNN, LSH

# Citace

CIGÁNIK, Marek. *Metody hledání k-nejbližších sousedů.* Brno, 2023. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Ivana Burgetová, Ph.D.

# Metody hledání k-nejbližších sousedů

## Prohlášení

Prohlašuji, že jsem tuto semestrální práci vypracoval samostatně pod vedením paní Ing. Ivany Burgetové Ph.D.. Uvedl jsem všechny literární prameny, publikace a další zdroje, ze kterých jsem čerpal.

.....
Marek Cigánik
16. května 2023


## Poděkování

Děkuji paní Ing. Ivaně Burgetové Ph.D. za odbornou pomoc, vedení a včasné odezvy na mé žádosti. Její postřehy byly vždy trefné. Pomohly mi uvědomit si, co je na práci důležité a na co bych se měl v práci zaměřit. Také děkuji své přítelkyni za neobyčejnou podporu, díky které jsem v diplomové práci nekulhal a podařilo se mi ji dokončit. V neposlední řade blízkému okolí rodiny a přátel za to, že se mnou byli po celou dobu studia od prvního až po poslední ročník.

# Obsah


**1 Úvod** **3**

**2 Algoritmus K-nejbližších sousedů** **4**
2.1 Vztah k datům a datovým typům . . . . . 5
2.1.1 Vzdálenost . . . . . 5
2.1.2 Problémy . . . . . 6
2.2 Hodnocení KNN metod . . . . . 10
2.3 Využití KNN . . . . . 11

**3 Aproximační metody vyhledávání KNN** **13**
3.1 KD-Strom . . . . . 14
3.1.1 Konstrukce, úrovně KD-Stromu . . . . . 14
3.1.2 Vyhledávání . . . . . 16
3.1.3 Složitosti . . . . . 19
3.2 Locality-Sensitive Hashing . . . . . 20
3.2.1 Konstrukce . . . . . 22
3.2.2 Vyhledávání . . . . . 23
3.2.3 Složitosti . . . . . 24
3.2.4 Hardware urychlení . . . . . 25
3.3 Strom náhodných projekcí . . . . . 25
3.3.1 Konstrukce . . . . . 26
3.3.2 Vyhledávání . . . . . 28
3.3.3 Složitosti . . . . . 28
3.4 Kulovitý strom . . . . . 29
3.4.1 Konstrukce . . . . . 30
3.4.2 Vyhledávání . . . . . 31
3.4.3 Složitosti . . . . . 33
3.5 Algoritmy hledání KNN založené na grafu . . . . . 33
3.5.1 Konstrukce . . . . . 34
3.5.2 Vyhledávání . . . . . 36
3.5.3 Složitosti . . . . . 38

**4 Návrh měření a experimentů** **39**
4.1 Implementace . . . . . 39
4.2 Data . . . . . 41
4.2.1 Datová sada o pacientech Covid-19 . . . . . 41
4.2.2 Generované data . . . . . 42
4.3 Knihovny . . . . . 43

4.3.1 KD-Strom . . . 43
4.3.2 Kulovitý strom . . . 44
4.3.3 Locality-Sensitive Hashing . . . 45
4.3.4 Strom náhodných projekcí . . . 46
4.3.5 Algoritmy hledání KNN založené na grafu . . . 46

**5 Experimenty** **49**
5.1 Vlastní parametry aproximačních metod . . . 49
5.1.1 KD-Strom . . . 49
5.1.2 Kulovitý strom . . . 50
5.1.3 Locality-Sensitive Hashing . . . 51
5.1.4 Strom náhodných projekcí . . . 53
5.1.5 Algoritmy hledání KNN založené na grafu . . . 55
5.2 Společné měření . . . 59
5.2.1 Vyhodnocení . . . 63

**6 Závěr** **65**

**Literatura** **66**

**A Zbylé měření vlastních parametrů aproximačních metod** **69**

# Kapitola 1

# Úvod

K-nejbližších sousedů je starý (1951) [6] algoritmus používaný dodnes jak v základní, lineární formě nebo v nějaké z upravených forem. Tyto nové formy vznikly z důvodu nárůstu požadavků na algoritmus z hlediska mohutnosti a dimenzionality prohledávaného prostoru. Cílem práce je přiblížit metody algoritmu K-nejbližších sousedů a empiricky prozkoumat jejich vlastnosti. Je zaměřena na jejich schopnost spolehlivě vyhledat nejbližší sousedy a přínosy nebo ztráty, které z metod vyplývají. Oproti naivnímu algoritmu se snaží hlavně zkrátit vyhledávací časy, se kterými má základní algoritmus problém.

V práci jsem se zaměřil na popis algoritmu, jeho vztah k datům a vysvětlení, proč je tak oblíbený, i když už existují komplexnější algoritmy s některými lepšími klíčovými vlastnostmi. Bude popsáno několik z upravených implementací, či už používanějších tradičních nebo moderních. Každá bude popsána z několika pohledů: základní myšlenka, konstrukce, vyhledávání, asymptotické složitosti a dodatky ke konkrétní implementaci, které stojí za zmínku.

Práce může sloužit při rozhodování výběru algoritmu hledání podobných objektů a jako návod implementace. Algoritmus může sloužit jako klasifikátor, detektor odlehlých hodnot, systém generující doporučení nebo sloužit pro hledání podobných dokumentů [14].

# Kapitola 2

# Algoritmus K-nejbližších sousedů

Algoritmus je oblíbený kvůli jednoduchosti svého konceptu a snadné interpretovatelnosti. Základním úkolem je získat pro vyhledávaný objekt nejpodobnější objekty z množiny dat. Tyto objekty mohou posloužit pro zařazení vyhledávaného objektu do třídy většinovým hlasováním (klasifikaci) [1], pro vytvoření regresní hodnoty z průměru nějakého vlastního atributu objektů [23], nebo mohou samy představovat cíl vyhledávání pro doporučovací systémy, systémy vyhledávající plagiáty a podobně [17].

Podobnost objektů se určuje libovolnou vzdálenostní funkcí. Prostor, ve kterém se vzdálenost počítá, je tvořen atributy objektu. Každý atribut tvoří jednu dimenzi, počet atributů není omezený, jedná se tedy o obecně $D$-dimenzionální prostor, kde $D \in \mathbb{N}$. Naivní algoritmus vypočítá vzdálenosti od vyhledávaného objektu ke všem objektům z datové sady a ty postupně ukládá do vzestupné posloupnosti. Po uvážení všech objektů vznikne konečný seřazený list, kteréhož prvních $k$ objektů je návratovou hodnotou algoritmu.

Pokud je cílem klasifikace, většinovým hlasováním se určí třída. Při měkkém rozhodování se konstruuje množina dvojic $(c, p)$, kde $c$ je třída a $p$ je pravděpodobnost příslušnosti objektu k $c$ třídě $p = \frac{|c_r|}{k}$, kde $c_r = \{x : x \in KNNresult \bigwedge x_c = c\}$. Pokud je cílem regrese, výsledek je aritmetický průměr (nebo medián, geometrický průměr...) $reg = \frac{\sum_{i=1}^{i=k} x_i}{k}$.

Interpretovatelnost je jednoduchá a myšlenkový proces člověku velmi blízký. Už v antickém Řecku [16] můžeme pozorovat snahy o popsání objektů pomocí atributů a na jejich základě vyvozovat příslušnost třídám. Iniciativy se projevovaly v Platónově škole. Je autorem teorie idejí, myšlenky, že mimo naše vnímání existují objekty v jejich čisté formě, ke které se každodenní věci snaží přiblížit. Existuje „modrotiskové schéma“ toho, jak by měl vypadat strom, člověk, kartáček...

Popsáno atributy, kartáček by měl být asi 15 centimetrů dlouhý a mít na jednom konci štětinky. K tomuto závěru lze dospět pozorováním různých kartáčků představujících datovou sadu. Atributy s nízkou informační hodnotou, jako například barva, se záměrně vyloučí, aby neovlivňovaly výsledek (dimenzionální redukce). Teď, když se člověk znalý klíčových atributů kartáčku a jejich hodnot setká s nespecifikovaným objektem, pro jeho zařazení mu stačí porovnat vlastnosti objektu k takzvanému objektu čisté formy.

I když z jiných záměrů, než kvůli kterým jsou implementovány klasifikační algoritmy dnes, Platónova škola dokazuje vztah přirozeného lidského uvažování k metodikám použitých v KNN. Dokonce si uvědomovali chyb, které můžou ze špatné klasifikace vyústit. Definování člověka jako „dvojnožce bez peří“ (neznali zvířata jako klokan) dovolilo Díogenovi ze Sinópé dovléct vypelichaného kohouta se slovy „Podívejte, jakého báječného člověka jsem našel!“. Platón poté musel přidat do definice „... mající široké nehty“ [13].

## 2.1 Vztah k datům a datovým typům

Algoritmus funguje v některých případech lépe než v jiných. Nejvíce ovlivňujícím faktorem je datová struktura objektů v otázce a jejich četnost.

### 2.1.1 Vzdálenost

Co se výpočtu vzdálenosti týče, ten se liší podle typu atributu, vždy se ale používá již zmíněná vzdálenostní funkce [4]. Jedná se o funkci $d$, která musí nad libovolnými body v prostoru $p$, $q$ a $r$ splňovat podmínky:

- $d(p, q) \geq 0$ nezáporná vzdálenost
- $d(p, q) = D(q, p)$ symetrie
- $d(p, r) + D(r, q) \geq D(p, q)$ trojúhelníková nerovnost
- $d(p, p) = 0$ totožnost

U intervalových proměnných se používá například Euklidovská vzdálenost, Šachovnicová vzdálenost, Minkowského vzdálenost, Čtvercová vzdálenost nebo Manhattanovská vzdálenost.

$$d(x, y) = \sqrt{\sum_{i=1}^{D} (\mid x_i - y_i \mid)^2} \tag{2.1}$$

(a) Výpočet Euklidovské vzdálenosti bodů $x$ a $y$ s $D$ dimenzemi. Výsledkem je délka úsečky mezi body.

$$d(x, y) = \sum_{i=1}^{D} \mid x_i - y_i \mid \tag{2.2}$$

(b) Výpočet Manhattanovské vzdálenosti bodů $x$ a $y$ s $D$ dimenzemi. Výsledkem je součet vzdáleností jednotlivých dimenzí. V literatuře se formuli říká také Vzdálenost městských bloků.

$$d(x, y) = \max_i (\mid x_i - y_i \mid) \tag{2.3}$$

(c) Výpočet Šachovnicové vzdálenosti bodů $x$ a $y$. Výsledkem je vzdálenost v dimenzi $i$ s největším rozdílem. V literatuře se jí říká také Chebychevova vzdálenost.

$$d(x,y) = \sqrt[p]{\sum_{i=1}^{D}(\mid x_i - y_i \mid)^p} \tag{2.4}$$

(d) Výpočet Minkowského vzdálenosti bodů $x$ a $y$. Výpočet obsahuje nastavitelný parametr $p$. Povšimněme si, že když je $p = 1$, vzorec odpovídá Manhattanovské vzdálenosti a když je $p = 2$, vzorec odpovídá Euklidovské vzdálenosti. Dále, pokud by se $p$ limitně blížilo k nekonečnu, výsledek by byla vzdálenost dimenzí s největším rozdílem - Šachovnicová vzdálenost. Minkowského vzdálenost je tedy generalizací a Manhattanovská, Euklidovská a Šachovnicová vzdálenost jsou specializace. Pokud $p < 1$, nejedná se o vzdálenostní metriku. Pro příklad, pokud by $p < 1$, tak pro vektory $s = (0,0), q = (1,1)$ je vzdálenost $d(s,q) = 2^{1/p}$. Třetí vektor $r = (0,1)$ je od obou vzdálený 1. Protože ale $d(s,r) + D(r,q) < D(s,q)$, trojúhelníková nerovnost neplatí.

U nominálních proměnných se vzdálenost počítá shodami a neshodami. Shody i neshody mohou být dvojího typu podle shodné/neshodné hodnoty.

| | x = 1 | x = 0 |
|---|---|---|
| y = 1 | q | r |
| y = 0 | s | t |

Tabulka 2.1: Možnosti podobnosti vektorů. $x$ a $y$ jsou substitucí pro složky vektoru, složek je $p = q+r+s+t$. $q$ je pozitivní shoda, $t$ je negativní shoda, $r$ a $s$ jsou neshody s opačnými hodnotami.

$$jac(u,v) = \frac{q+t}{q+r+s+t} \tag{2.5}$$

(a) Jaccardova vzdálenost. Je to poměr mezi shodami a počtem proměnných.

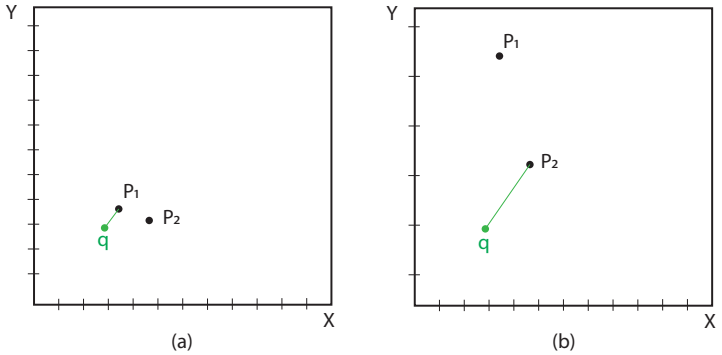
$$ham(u,v) = r+s \tag{2.6}$$

(b) Hammingova vzdálenost. Výsledkem je součet neshod. Interpretuje se také jako počet akcí, které se musí uskutečnit, aby byly vektory stejné.

Pokud jsou nominální hodnoty ordinální, vzdálenost mezi neshodami je násobena počtem možných hodnot mezi nimi. V ordinálním atributu $velikost \in \{maly, stredni, velky\}$ by platilo, že $d(maly, stredni) = 1$ a $d(maly, velky) = 2$.

### 2.1.2 Problémy

Algoritmy používající vzdálenostní funkci čelí mnoha problémům,. Jeden z řešitelných je měřítko jednoho z uvažovaných atributů vůči druhému. Vzdálenostní funkce v neupravené verzi vnímá všechny dimenze jako rovnocenné. Úpravou měřítka se zvýhodňuje/znevýhodňuje výpovědní síla atributu vůči ostatním 2.1.

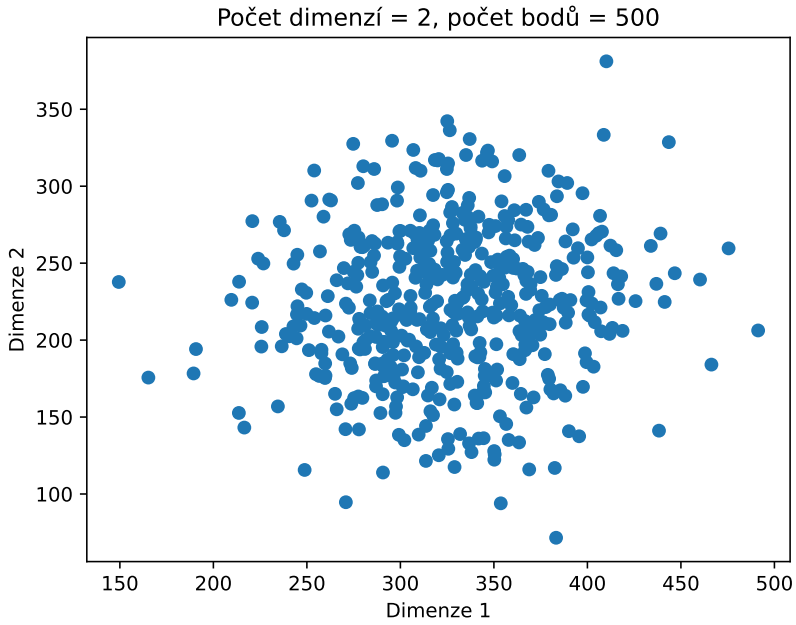



Obrázek 2.1: Změna měřítka atributu $Y$ ovlivňující výsledek algoritmu. V grafu **(a)** je vyhodnocen jako nejbližší soused pro $q$ bod $P_1$. V grafu **(b)** po úpravě a znevýhodnění výpovědní síly atributu $Y$ je vyhodnocen jako nejbližší soused bod $P_2$.

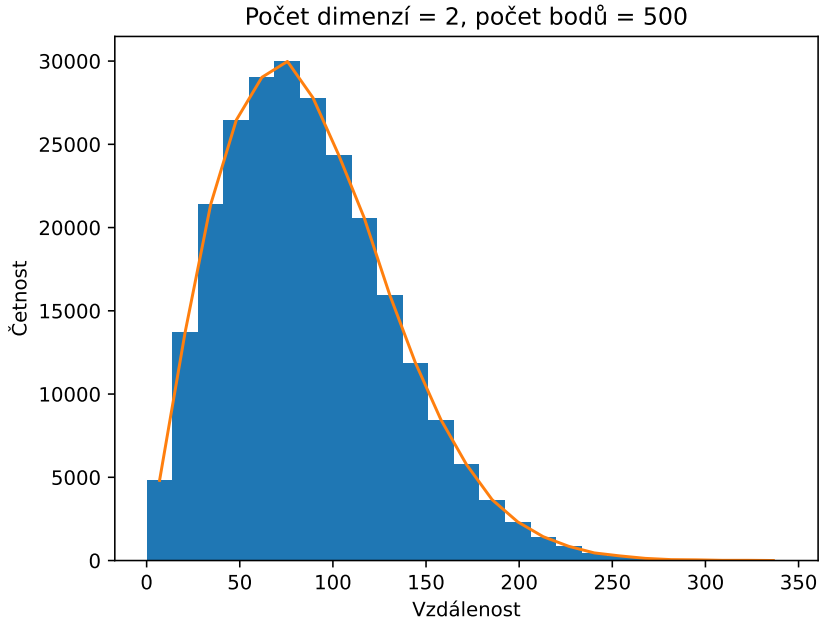
Měřítka, pokud se jedná o fyzikální veličiny, nemusí být nutně stejné jednotky. Není problém, když jeden atribut délky je v metrech a druhý v milimetrech. Pro příklad, objekt hada pro určení jeho druhu může mít atributy délka těla v metrech a délka zubů v milimetrech. Důležité je zachovat výpovědní hodnotu, přizpůsobit se oboru hodnot, které může atribut reálně nabývat (výška člověka obvykle nepřesáhne $2,5$m) a pomocí experimentů uzpůsobit měřítka tak, aby algoritmus dosahoval nejlepší výsledky.

Co se počtu atributů (dimenzí) týče, nastává problém zvaný prokletí dimenzionality [27]. Souvisí se vzdáleností objektů mezi sebou. V nízko dimenzionálních prostorách se vzdáleností funkce chová očekávaně – dokáže detekovat relativní sousednost. Čím víc dimenzí prostor nabývá, tím víc „příležitostí“ pro větší rozdíl nebo neshodu mezi sebou objekty mají. Příbuznosti objektů se stírají a ztrácí se sousednost. V extrémních vysoko dimenzionálních prostorách nejbližší dva objekty mají podobnou vzdálenost jako nejvzdálenější, přičemž interpretace této hodnoty se nedá vyvodit jako nízká, blízká, sousední. U algoritmů využívajících vzdálenostní funkce se obecně doporučuje používat nízký počet dimenzí i kdyby každá dimenze z vysoko dimenzionálního prostoru měla velkou výpovědní hodnotu. Optimální počet dimenzí není definován, liší se podle datových sad a algoritmů.

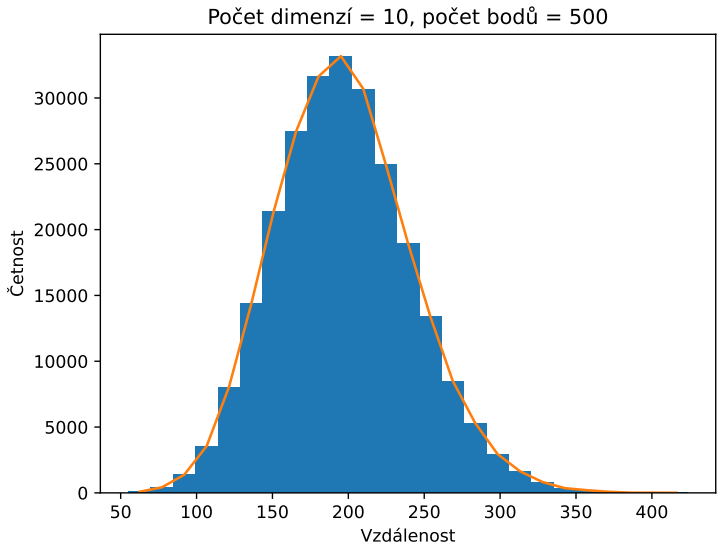
Uvažujme datovou sadu na obrázku 2.2, ve které je počet dimenzí $D$ proměnlivý, data jsou normálního rozložení $(\mu, \sigma^2)$ v každé dimenzi s parametry $\mu = 100 - 500$ a $\sigma = 40 - 50$. Zaznamenejme vzdálenost od každého bodu ke každému. V nízko dimenzionálních prostorách, kde existuje sousednost, se dá vypozorovat, že existuje značné množství párů, které jsou bezprostředně blízko 2.3. S postupným zvyšováním dimenzí roste průměrná vzdálenost libovolných dvou bodů, odchylka zůstává stejná. Klesá ale počet bodů, které k sobě mají bezprostředně blízko 2.4. V extrémních případech už sousednost prakticky neexistuje 2.5.



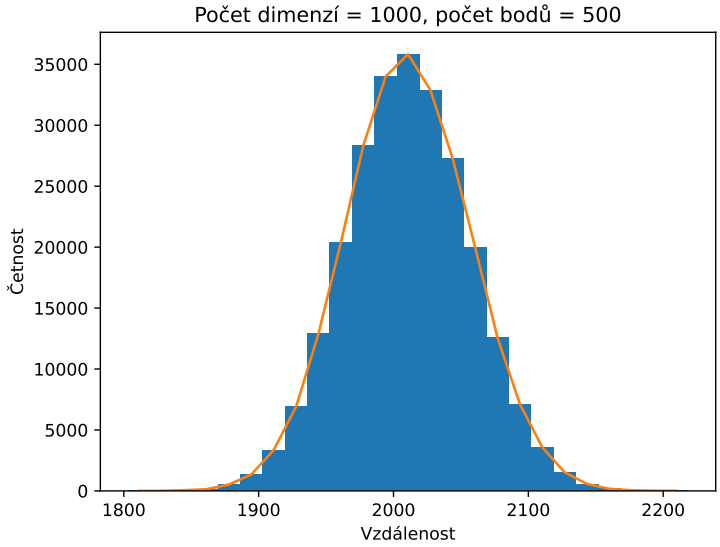

Obrázek 2.2: Datová sada s počtem dimenzí $D = 2$ generované normálním rozložením. Všimněme si, že pro mnoho bodů existuje nějaký soused v bezprostřední blízkosti.




Obrázek 2.3: Vzdálenosti mezi body v prostoru $D = 2$ generované normálním rozložením 2.1.2. Počet dvojic, které mají mezi sebou vzdálenost blízkou 0 je skoro 5000. Na této datové sadě by algoritmus KNN fungoval excelentně, protože se spoléhá na to, že takovéto dvojice existují.




Obrázek 2.4: Vzdálenosti mezi body v prostoru $D = 10$ generované normálním rozložením 2.1.2. Už s 10 dimenzemi je problém najít dva podobné body. Průměrná vzdálenost je 200 a neexistuje jediná dvojice, která by mezi sebou měla vzdálenost menší než 50.




Obrázek 2.5: Vzdálenosti mezi body v prostoru $D = 1000$ generované normálním rozložením 2.1.2. V extrémních případech sousednost prakticky neexistuje (žádný bod není jinému blízký). Může vyvstat argument, že pořád existují body, které k sobě mají blíž a jsou si podobnější, než jiné. Je potřeba brát ohled na to, že pro to, aby byly objekty stejné, musí mít mezi sebou vzdálenost 0 a podobné objekty se vzdálenostně kolem nuly soustředí. Zde nejpodobnější dvojice jsou vzdálené přibližně 1900 a nejvzdálenější 2150. I ty nejbližší body jsou podobně vzdálené, jako ty nejvzdálenější. Žádní sousedé neexistují, pouze velmi vzdálené body, i když některé mírně vzdálenější, než jiné.

Jako příklad je možné si představit praktickou analogii s doporučovacím systémem pro filmy. S malým množstvím dimenzí (žánr, rok premiéry, hlavní role...) by systém, který používá KNN, pracoval dobře. Čím vice dimenzí by algoritmus uvažoval (délka filmu, počet vedlejších postav, počet exteriérových scén...), tím by se zvětšoval rozdíl mezi podobnými filmy, protože v konečném důsledku se objeví dimenze, v kterých se budou lišit. Ve stejnou chvíli se objeví podobnosti mezi dříve odlišnými tituly a v konečném důsledku budou mít podobný poměr shod ku neshodám/rozdílům jako původně podobné filmy.

Podobně, jako velký počet dimenzí, i velký počet dat je Achillovou patou KNN algoritmů. Při použití KNN uživateli záleží na čase, který algoritmus potřebuje pro **(a)** vytvoření modelu, **(b)** vyhledávání.

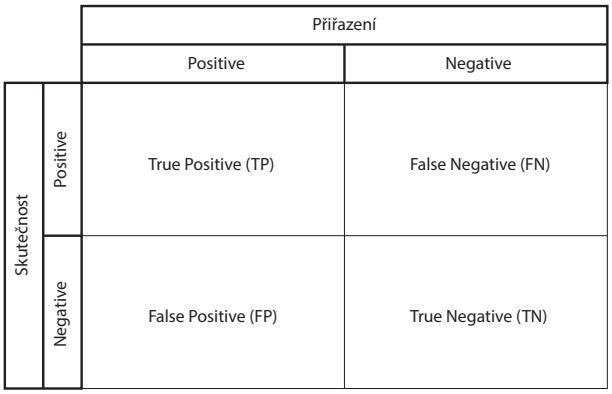
**Rychlost vytvoření modelu je důležitá pro systémy, u kterých se klade důraz na poměr online/offline času a hrozí výpadky**. Opětovné vytváření modelu, pokud je pomalé, je v dané situaci problémem. Lineární verze algoritmu KNN je v tomto ohledu dobrou volbou. Vytvoření modelu znamená pouhé načtení dat do paměti. Každopádně, i to je ovlivněno velikostí datové sady a rostoucím počtem dat se inicializace zpomaluje. Tento efekt je více znatelný u jiných verzí KNN, které požadují vytvoření nějaké datové struktury jako stromy nebo grafy.

Časová složitost vyhledávání je lineární, jelikož KNN algoritmus v základní podobě musí uvažovat všechny data z datové sady. Existují implementace, které snižují vyhledávací čas za cenu větší paměťové složitosti a/nebo časové složitosti tvorby modelu.

Velikost dat v datové sadě pro vyhledávání je jeden z klíčových faktorů, který je potřebné zohlednit při výběru algoritmu. U KNN platí, že čím je větší datová sada, tím je horší časová a prostorová složitost. U velké datové sady stojí za zvážení použít jiný algoritmus.

## 2.2 Hodnocení KNN metod

V oblasti strojového učení existuje mnoho metrik určujících správnost modelů, každá se svou interpretací chybovosti. Ty základní vycházejí z matice záměn (chybové matice) 2.6. Základní metriky se jmenují accuracy, precission a recall.




Obrázek 2.6: Chybová matice. V kontextu KNN se používá jako prostředek určení správnosti modelu. Zobrazuje přiřazení sousedů. True Positive (TP) je počet správně přiřazených, False Positive (FP) je počet nesprávně přiřazených, False Negative (FN) je počet nesprávně nepřiřazených a True Negative (TN) je počet správně nepřiřazených sousedů. Platí, že $TP + FP = k$, protože model KNN vždy přiřazuje (Positive) pouze a právě $k$ sousedů. Také platí, že $TP + FN = k$, čili $FP = FN$, ke každému nesprávně přiřazenému (FP) sousedovi přiléhá jeden nesprávně nepřiřazený (FN).

Při testování aproximačních KNN metod je důležitá odpověď na otázku „Jaké procento získaných nejbližších sousedů jsou skuteční nejbližší sousedé?“. Jinými slovy, poměr mezi správně získanými sousedy oproti $k$. K získání odpovědi pomůže metrika recall, která se počítá $recall = \frac{TP}{TP+FN}$. Protože $TP$ jsou získaní skuteční nejbližší sousedé a $TP+FN = k$. Protože ale $Precision = \frac{TP}{TP+FP}$ a $FP = FN$, precision a recall budou mít stejný výsledek.

Odvozenou a užitečnou metrikou je počet vyhledávání, které metoda zvládne za dosažení určitého recallu, takzvaný Recall-Queries per second trade-off. Aproximační metody jsou často nastavitelné různými vlastními parametry, které upravují poměr rychlosti vyhledávání k recallu. Čím vyšší recall, tím menší počet vyhledávání za sekundu a naopak. Tato metrika neuvažuje poměr rychlosti konstrukce modelu k recallu.

Pokud je účelem použití metriky sledování závislosti recallu od zvoleného $k$, používá se Recall@K nebo-li Recall at rank. Metrika se používá u seřaditelných výsledků, což nejbližší sousedé jsou. V seřazených výsledcích se vyskytují jak relevantní výsledky (skuteční sousedé), tak nerelevantní. $K$ v Recall@K referuje k počtu prvků vrácených systémem, v případě KNN shodné s $k$ ve vyhledávání. Obecně, Recall@K uvažuje $m$ relevantních prvků v datové sadě. Měří kolik prvků z $m$ je v $K, K \geq m$ vrácených prvků zastoupeno. V případě KNN je $m = K = k$, stejný počet relevantních prvků jako vyhledávaných. Zvyšováním $K$ by se měl recall zvyšovat také. Pokud $K = N$, recall je stoprocentní, protože jsou navráceny všechny prvky, tudíž i každý z $m$ požadovaných prvků. Prakticky využití této metriky u KNN znamená sledování, zda-li počet vyhledávaných sousedů ovlivňuje recall.

## 2.3 Využití KNN

Z vlastností KNN vyplývá několik faktorů, které je třeba před implementací uvážit.

Výsledky z KNN jsou přesné, ať už jde o klasifikaci, regresi nebo doporučovací úlohu. Algoritmus uvažuje nejbližší okolí vyhledávaného bodu, což u kvalitní datové sady vede ke kvalitním návratovým hodnotám. U použití KNN se víc věnuje pozornost datové sadě samotné než implementaci, protože ta má na dobré sadě zaručené dobré výsledky.

Implementace KNN je ve většině případů poměrně jednoduchá, lehce interpretovatelná a odladitelná. Tím konkuruje složitějším algoritmům, které kladou důraz na jiné aspekty jako rychlost vyhledávání nebo prostorová složitost.

Využití najde u menších datových vzorků s nízkým počtem dimenzí a v situacích, kdy není požadována vysoká frekvence vyhledávání kvůli delegaci výpočetní zátěže na akt vyhledávání samotný.

Algoritmus je dobré použít, pokud [1]:

- Se klade důraz na kvalitu výsledku a je zaručena kvalita datové sady.
- Je možné, že za běhu bude potřebná úprava datové sady. Přidání, odebrání dat nebo celé třídy v případě klasifikace.
- Je nízký počet dat v datové sadě.
- Data mají nízký počet dimenzí.
- Je požadována interpretovatelnost implementace.
- Není předpokládaná vysoká frekvence vyhledávání.

# Kapitola 3

# Aproximační metody vyhledávání KNN

KNN lineární algoritmus po obdržení bodu pro vyhledání sousedů prochází všechny data z datové sady, spočítá délky a navrací uspořádaný seznam $k$ nejbližších sousedů. Toto řešení vyžaduje nízký čas inicializace a má nejmenší prostorovou zátěž, jaké se dá bez komprimace docílit. Jeho hlavním problémem je ale časová složitost vyhledávání $\mathcal{O}(n)$. Redukce časové složitosti je hlavní motivací upravených verzí KNN. Obecně ale u upravených verzí platí, že zlepšení časové složitosti vyhledávání vede ke zhoršení prostorové složitosti a (nebo) zhoršení časové složitosti konstrukce. Interpretovatelnost zůstává zachována, protože algoritmům zůstává stejné jádro, šablona:

1. **Konstrukce**.
2. Obdržení vyhledávaného bodu.
3. **Procházení prostoru a hledání nejbližších sousedů**.
4. Navrácení nejbližších sousedů.

Esenciální rozdíl mezi lineární a libovolnou upravenou verzí je v konstrukci a prohledávání prostoru. Zbylé aspekty algoritmu zůstávají nedotčeny. Z úpravy **Konstrukce** plyne zhoršení prostorové složitosti a časové složitosti konstrukce. Z úpravy **Procházení prostoru a hledání nejbližších sousedů** plyne zlepšení časové složitosti vyhledávání.

V práci jsou představeny metody KD-Strom, Locality-Sensitive Hashing, Strom náhodných projekcí, Kulovitý strom a grafové algoritmy.

KD-Strom je jednoduchý na implementaci, používá se v nízko dimenzionálních prostorách (oblast počítačové grafiky). Krok v KD-Stromu je jednoduchý na výpočet, vhodný pro procesory grafické karty. Jeho upravené verze, Strom náhodných projekcí a Kulovitý strom, jsou méně náchylné na počet dimenzí. Locality-Sensitive Hashing využívá hashovací funkci k redukci počtu kandidátů pro nejbližší sousedy. Grafové algoritmy jsou rodina algoritmů, která pro získání výsledku nepoužívá dělení prostoru, ale průchod grafem, kde uzly $q, r \in N$ jsou prohledávané data a hrany $(q, r)$ (většinou) značí vztah pro $r$ „býti mezi $k$ nejbližšími sousedy bodu $q$“. Rodina je rozšiřována a je předmětem nejnovějších výzkumů algoritmů KNN.

## 3.1 KD-Strom

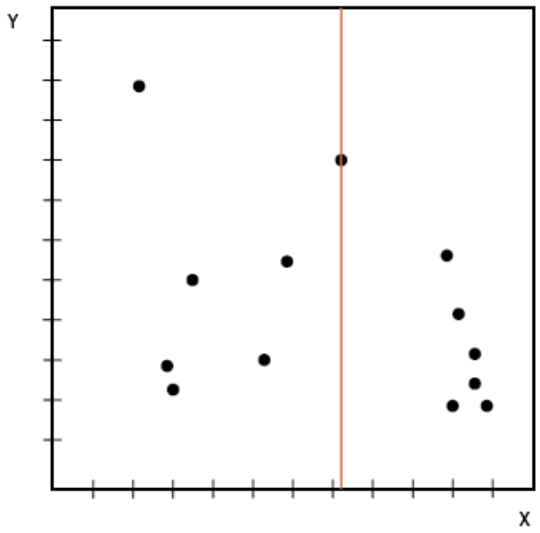
KD-Strom [15] je datová struktura uzpůsobená k vyhledávání nad vícero parametry. Vychází z předpokladu, že řazená posloupnost je časově méně náročná na vyhledávání. Pokud jsou data $K$-dimenzionální, naivní řadící algoritmus musí vybrat, podle které dimenze posloupnost seřadí, protože posloupnost je možno vytvořit pouze nad jedním atributem, ne kombinovaně. Data seřazená podle jedné dimenze mohou přinést zrychlení ve vyhledávání. Nastávají ale případy, kdy seřazení nepřinese žádný nebo malý účinek a to když všechna data mají stejnou hodnotu nebo skoro stejnou hodnotu v dané seřazené dimenzi.

Seřazení podle jednoho atributu a vyhledání nejbližšího souseda v takto seřazené posloupnosti ale stále nepřináší žádnou jistotu, že si tím algoritmus v hledání nejbližšího souseda pomůže. V struktuře KD-Strom dochází k zohlednění vícero kritérií. KD-Strom je binární vyhledávací strom, který obsahuje uzel (či už vnitřní nebo listový) pro každý záznam z datové sady. Řadí data do posloupnosti ve všech $K$ dimenzích (odtud název $K$ dimenzionální) střídavě. Každá úroveň stromu řadí podle jednoho kritéria.

Vnitřní uzel $u$ rozděluje prostor na 2 subprostory (má 2 potomky) hyperrovinou, která:

- prochází $u$
- je kolmá na osu dimenze, podle které strom řadí v úrovni, ve které se $u$ nachází
- Je rovnoběžná s osami ostatních dimenzí

Taková hyperrovina je pouze myšlená. V praxi se jedná o rozdělení dat na data s menší hodnotou a data s větší hodnotou v dané dimenzi 3.1.




Obrázek 3.1: Rozdělení prostoru podle dimenze zobrazené na ose $X$.

### 3.1.1 Konstrukce, úrovně KD-Stromu

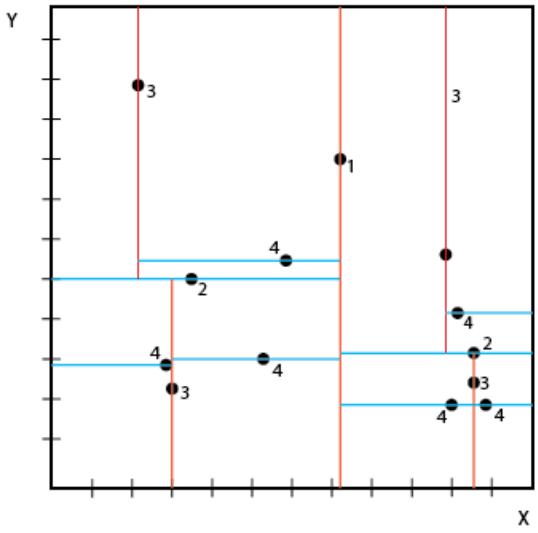
Každá úroveň ve stromu se řadí podle jedné dimenze. Po $K$ úrovních se dimenze typicky opakují. Uvažujme 3 dimenzionální data reprezentované vektorem $(x, y, z)$. Pokud první 3 úrovně budou rozdělovat prostor v pořadí $xyz$, následující úrovně budou tuto posloupnost opakovat. K samotné struktuře KD-Stromu je tedy potřebné uchovávat ještě jednu cyklickou

strukturu uchovávající data o výběru dimenze pro řazení (nebo v uzlu uchovat informaci o výběru dimenze). Každým zanořením se výběr v cyklické struktuře posune dopředným směrem, vynořením se výběr posune směrem opačným.

Výběr pořadí se nemusí řídit žádnou heuristikou, je to ovšem doporučený postup. Nejpoužívanější je výběr podle rozptylu $\delta^2$ datové sady v dané dimenzi a prostoru [11]. Existuje $K!$ možností uspořádání výběru, kombinační strom se ale jednoduše ořezává, protože je potřebné spočítat pro každou úroveň pouze rozptyly nepoužitých dimenzí. V první úrovni je to $K$ dimenzí, vybere se dimenze s největším rozptylem dat a použije se jako rozdělovací dimenze též úrovně. V druhé úrovni už jenom $K-1$ dimenzí, ve třetí $K-2$ až do 1. Poté jsou vybrány všechny dimenze a v dalších úrovních se jejich pořadí opakuje. Dohromady je potřebné vypočítat pouze $\sum_{x=1}^{K} x$ rozptylů, ne $K!$. Po výběru posloupnosti $K$ dimenzí se zkonstruuje již zmíněná cyklická struktura (nebo informaci o dimenzi nese uzel).

Je možné vytvořit acyklický výběr striktně podle rozptylu pro každou úroveň, což znamená výpočet rozptylu každé dimenze pro každou úroveň ($K \log(n)$ výpočtů). Takový přístup je poměrně zbytečný, protože nejvíce záleží na rozdělení v prvních úrovních.

Binární vyhledávací strom má dobré vyhledávací časy a vlastnosti, když je vyvážený. Nevyvážený strom v extrémním případě nepřináší žádné zlepšení a časová složitost vyhledávání se rovná lineárnímu algoritmu. Pro zajištění hloubkové vyváženosti záleží na výběru uzlu rozdělujícím zbytek prostoru na dva podstromy. Používá se technika, kdy vybraný uzel je medián v dimenzi datové sady, kterou rozděluje. Jeho podstromy se budou lišit počtem uzlů maximálně o 1. Dá se říct, že u klasického binárního stromu se medián vybírá z jedné dimenze (klasický binární vyhledávací strom je vlastně KD-Strom, kde $K = 1$). U KD-Stromu se medián volí v dimenzi zvolené podle pomocné struktury a úrovně zanoření. Takovým výběrem levý i pravý podstrom obsahují stejný počet uzlů $\pm 1$, přičemž levý podstrom obsahuje data s (obvykle) menší hodnotou v rozdělovací dimenzi a pravý podstrom obsahuje data s větší hodnotou. V praxi, pokud datová sada obsahuje velké množství dat, se používá medián z náhodně zvoleného reprezentativního vzorku, což vede ke skoro vyváženému stromu (dostatečná aproximace). Grafické zobrazení KD-Stromu nad daty má podobu střídajících se rozseknutí prostoru 3.2.



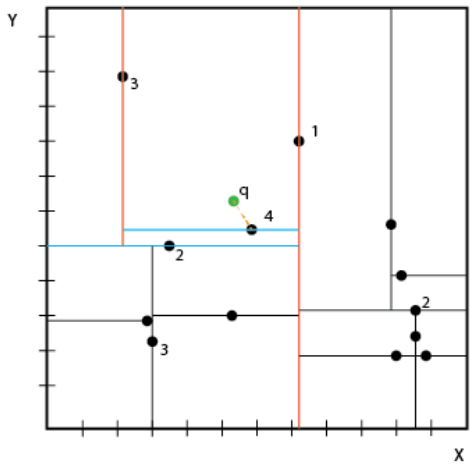

Obrázek 3.2: 2-dimenzionální data a výběr mediánů v jednotlivých úrovních stromu. Číslem je značená hloubka stromu, ve které došlo k danému rozdělení. Všimněme si střídání výběru dimenze pro rozdělení prostoru. Hyperroviny rozdělují pouze už rozdělený prostor z vyšší úrovně stromu, ne celý datový prostor.

Vyváženost v KD-Stromu se nedá jednoduše uchovat vyvažujícími algoritmy jako u obyčejného jedno dimenzionálního vyhledávacího stromu. Rotace není možná, protože každá úroveň pracuje s jinou dimenzí. Rotací by se vybrané mediánové uzly ocitly na nesprávných úrovních, což by vedlo k ještě vyšší míře nevyváženosti. Je tedy možné zkonstruovat vyvážený strom, vkládáním dat ale strom časem degraduje. Vkládání funguje stejně jako u obyčejného stromu, vytvořením nového listového uzlu.

Řešení nevyváženosti je rekonstrukce stromu. Ta může být implementována buď při každém novém vložení, nebo po několika požadavcích na vložení, které se dočasně uloží ve vyrovnávací paměti.

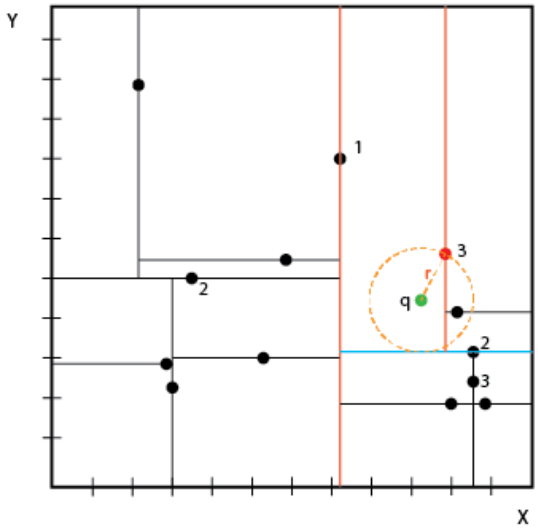
### 3.1.2 Vyhledávání

Uvažujme data z obrázku 3.3 a vytvořený strom. Dále uvažujme bod $q$, kterému se hledá nejbližší soused.




Obrázek 3.3: Bod $q$ je dotazovací bod, žlutá čára vede k nejbližšímu sousedovi. Algoritmus procházel strom, úrovně jsou značeny číslem. V každé úrovni měl na výběr z dvou potomků, výběr je značen barvou. Pokud by nejbližším sousedem nebyl listový uzel jako na obrázku ($q$ by bylo blíže k uzlu označeného jedničkou), algoritmus to zjistí při porovnání ve vynořování z rekurze.

Nyní uvažme, že se algoritmu nepodaří nalézt nejbližšího souseda 3.4.




Obrázek 3.4: Příklad, kdy pro bod $q$ není nalezen nejbližší soused.

Všimněme si nalezeného souseda 3.4 a kružnici se středem v $q$ o poloměru $r = d(q, x)$, V obrázku je vidět bod, který je blíže ke $q$, jelikož se nachází v kružnici. Algoritmus tedy nejbližšího souseda nenašel. Chyba nastala v okamžiku, kdy se prostor rozdělil podle červeného uzlu $x$ a uvážil se pouze levý subprostor. **To, že vyhledávaný bod $q$ má menší hodnotu v jedné dimenzi než červený uzel $x$ nutně neznamená, že nejbližší soused bude v levém subprostoru a naopak**. KD-Strom je tedy v základní podobě bez úprav aproximační metoda.

Populární je implementace zjištění této chyby pomocí backtrackingu, kdy se v každé úrovni vynoření zkontroluje, zda hodnota $d(q, best)$ je menší než vzdálenost $q$ od rozdělovací hyperroviny aktuálního uzlu. Pokud je vzdálenost menší, žádný bližší bod se za rovinou nacházet nemůže. Pokud je ale větší, část kružnice překračuje rozhodovací hranici a je šance, že v této úseči se nachází bližší bod. V tom případě vyhledávání pokračuje v druhé, předem nezvolené části podstromu. Tato technika aproximační metodu transformuje na metodu se 100% úspěšností vyhledání nejbližšího souseda za cenu snížení rychlosti dotazování. V čím vyšší úrovni stromu nastane chyba, tím větší je časová penalta. Chyba nemusí nastat pouze jedna za jedno vyhledávání. V extrémním případě, kdy chyba nastane u každého rozdělení prostoru, má technika časovou složitost lineárního algoritmu, jelikož projde celý strom. Frekvence chyby narůstá u KD-Stromu se zvyšujícím se počtem dimenzí. Zde jsou možnosti:

- Může se použít aproximační verze KD-Stromu.
- **Použít jinou, vhodnější techniku na vyhledávání ve vysoko dimenzionálních datech.**
- Použít kombinaci aproximačního přístupu a backtrackingu, kde se backtracking použije pouze na nejnižší úrovně stromu [12].

Zvyšujícím počtem dimenzí se také zvyšuje průměrná vzdálenost mezi body 2.5. Vzdálenost libovolného bodu od hyperroviny libovolné dimenze je ale nezávislá od celkového počtu dimenzí. V příkladě 2.3 na problémy vzdálenostních funkcí si představme hyperrovinu, která by přes středovou hodnotu rozdělovala data z normálního rozložení se směrodatnou odchylkou $\sigma = 40 - 50$. Vzdálenost libovolného bodu od ní by by byla přibližně v rozmezí $< 0, \frac{500-150}{2} >$ (v dalších rozděleních prostoru pro tuto dimenzi by se horní hranice intervalu pokaždé zmenšovala o polovinu, protože by rozdělovala už rozdělený prostor). Nyní uvažme příklad, kdy je prostor tvořen 1000 podobnými dimenzemi 2.5, kdy průměrná vzdálenost mezi body je přibližně 2000 a nejmenší přibližně 1900. Vzdálenost $d(q, best)$ bude tedy v nejlepším případě 1900, ale vzdálenost bodu $best$ od libovolné hyperroviny bude pořád v rozmezí $< 0, \frac{500-150}{2} >$ pro první úroveň rozdělení a ještě menší pro ostatní, protože se horní hranice půlí. Tudíž vzdálenost $dist(q, best)$ bude vždycky větší, než vzdálenost $q$ od hyperroviny. To má za následek, že při backtrackingu 3.1.2 se pokaždé navštíví každý podstrom, tedy se navštíví všechny uzly; prohledá se celý strom.

### Proces vyhledávání

1. Rekurzivně se prochází stromem od počátečního uzlu, volí se potomek podle porovnání v dimenzi dané úrovně stromu.
2. Dojde až k listovému uzlu, který bude považovat za nejbližšího souseda (pokud už ale nějaký uzel označený je, nejdříve dojde k porovnání vzdálenostní funkcí). Pokud listový uzel obsahuje vícero dat, lineárně se prohledají.
3. Při návratech z rekurze kontroluje, zda je aktuální uzel blíž, než označený nejbližší soused. Pokud ano, nejbližším sousedem se stává aktuální uzel.
4. Zjistí se, jestli se může nacházet potencionální nejbližší soused na druhé straně rozdělovací hyperroviny aktuálního uzlu. Pokud ano, vyhledávání pokračuje v druhém podstromu, tj. krok č. 2. Pokud ne, algoritmus pokračuje ve vynořování.
5. Vynořením z kořenového uzlu je ukončeno vyhledávání.

**Algorithm 1** Pseudokód vyhledávání v KD-Stromu

```
function KDTreeSearch(Node n, Node q, Node best, int level)
    dimension ← getDimensionForLevel(level)
    if isLeaf(n) and best is Null then
        return n
    end if
    if isLeaf(n) and d(n, q) < d(best, q) then
        return n
    end if
    if n[dimension] < q[dimension] then
        searchSide ← n.leftChild
        otherSide ← n.rightChild
    else
        searchSide ← n.rightChild
        otherSide ← n.leftChild
    end if
    best ←KDTreeSearch(searchSide, q, best, level + 1)
    if inceptsHyperplane(q, best, n) then
        best2 ←KDTreeSearch(secondSide, q, best, level + 1)
        best ← d(best, q) < d(best2, q) ? best : best2
    end if
    return best
end function
```

### 3.1.3 Složitosti

V neseřazené posloupnosti, jako jsou typicky data v datové sadě, se medián dá garantovaně najít s časovou složitostí $\mathcal{O}(n \log n)$ za použití heap nebo merge sortu[1]. Výběr mediánu se ale musí opakovat pro každé rozdělení prostoru. Vyvážený strom má hloubku $\log(n)$. Dohromady $\mathcal{O}(n \log(n) \log(n)) = \mathcal{O}(n \log^2 n)$.

Pokud se konstruuje strom s výběrem pořadí dimenzí v úrovních podle rozptylu, je potřebné spočítat pro každou úroveň rozptyl nepoužitých dimenzí. V první úrovni je to $K$ dimenzí, vybere se dimenze s největším rozptylem dat a použije se. V druhé úrovni už jenom $K - 1$ dimenzí až do $K = 1$. Dohromady $\sum_{x=1}^{K} x$.

Pomocný výpočet průměru $\mu$ má složitost $\mathcal{O}(n)$ a výpočet rozptylu $\sigma^2$ má také složitost $\mathcal{O}(n)$. Dohromady $2\mathcal{O}(n)$ pro každý jeden výpočet rozptylu. Je potřeba spočítat $\sum_{x=1}^{K} x$ rozptylů, což se dá brát jako konstanta (záleží na tom, jak chování škáluje s $n$, ne $K$). Výpočet rozptylů tedy zabere $(\sum_{x=1}^{K} x)\, 2\mathcal{O}(n) \approx \mathcal{O}(n)$.

Prostorová složitost vůči lineárnímu algoritmu nijak nenarůstá – $\mathcal{O}(n)$. Struktura navíc obsahuje pouze ukazatele na potomky. Ještě je zde pomocná struktura výběru dimenze pro úrovně stromu (za případu, že heuristicky volíme pořadí dimenzí), ta je ale zanedbatelně malá.

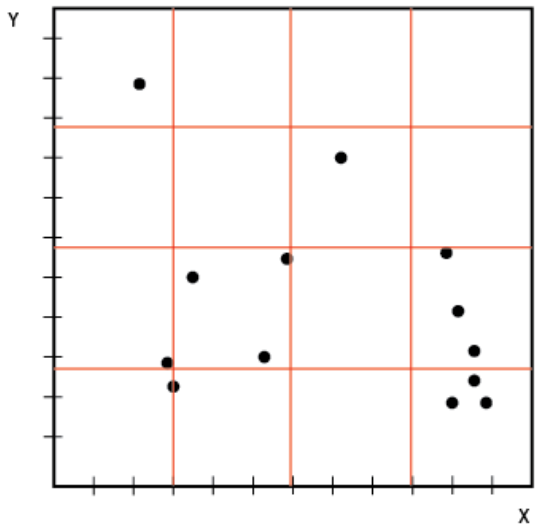
Časová složitost vyhledávání ve vyváženém stromu je $\mathcal{O}(\log n)$. Ořezávání větví stromu vede ke zlepšení vůči lineárnímu algoritmu. Kvůli prohledávání dříve zahozených větví se ale navštíví více, než jedna cesta stromem, čímž časová složitost narůstá, reálně se pohy-

[1] Existují algoritmy, které najdou medián v čase $\mathcal{O}(n)$ jako quickselect [8], ten ale v nejhorším případě může mít složitost $\mathcal{O}(n^2)$. Volí se stabilnější algoritmus.

buje mezi $\mathcal{O}(\log n)$ a $\mathcal{O}(n)$, tudíž žádným zlepšením vůči lineárnímu algoritmu (navštíví se všechny větve). U vysoko dimenzionálních dat se složitost přibližuje lineárnímu algoritmu. Algoritmus se pro takový druh dat nedoporučuje.

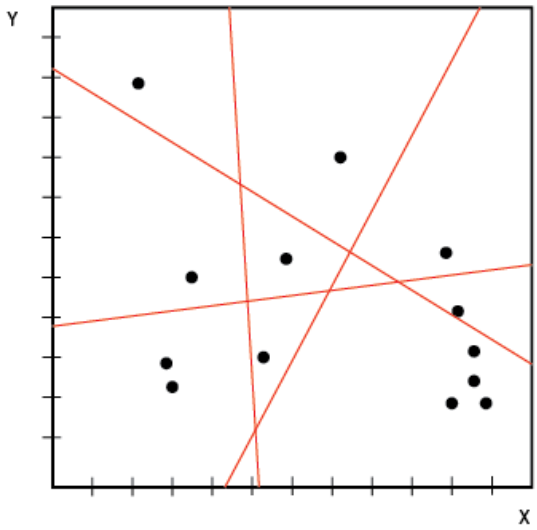
## 3.2 Locality-Sensitive Hashing

Local-Sensitive Hashing [22] nepoužívá žádné pokročilé datové struktury na ukládání a prohledávání. Idea vychází z rozdělení prostoru na regiony nebo-li zmenšení prostoru, který je potřebný prohledat. Odvození začíná u rozdělení prostoru mřížkou pomocí rovnoběžných hyperrovin v každé dimenzi.




Obrázek 3.5: Rozdělení prostoru na regiony pomocí $m = 3$ hyperrovin pro každou dimenzi.

Takovéto rozdělení z prostoru 3.5 pomocí $m$ hyperrovin pro každou z $D$ dimenzí zkonstruuje mřížku o $(m+1)^D$ regionech. Pokud se vychází z předpokladu, že není známé rozložení dat v žádné dimenzi, každý takový region by odhadem obsahoval $\frac{n}{(m+1)^D}$ dat. Pro dotazovaný bod $q$ by byl zjištěn region a následně by byl použit lineární algoritmus pro nalezení nejbližšího souseda. Locality-Sensitive Hashing tuto metodu rozdělení prostoru zdokonalil použitím nerovnoběžných, náhodně vygenerovaných hyperrovin 3.6.




Obrázek 3.6: Rozdělení prostoru na regiony pomocí náhodně vygenerovaných hyperrovin.

U rozdělení prostoru pomocí nerovnoběžných hyperrovin je počet regionů $\frac{m*(m+1)}{2}+1$, přičemž vlastnost, že regiony obsahují pravděpodobnostně stejný počet dat je zachována. Početně už ne $\frac{n}{(m+1)^D}$, ale $\frac{n}{\frac{m*(m+1)}{2}+1}$, tedy počet regionů ani počet dat v nich už není závislý od počtu dimenzí. Počet je závislý pouze od počtu hyperrovin, což je volitelný parametr algoritmu.

U obou případů, klasického rovnoběžného rozdělení i náhodného generování se algoritmus potýká s problémem, že nejbližší soused nemusí být přítomen v prohledávaném regionu. Může se nacházet v regionu sousedním. Zde se nabízejí 2 možnosti:

- Spočítaná vzdálenost mezi $q$ a potencionálním nejbližším sousedem z prohledávaného regionu se použije jako poloměr pomyslné kružnice, ve které se bližší bod (pokud existuje) – i když za hranicí regionu – bude nacházet. Kružnice se může, ale nemusí protínat s hranicí regionu. Pokud se neprotíná, je možné prohlásit, že kandidátní nejbližší soused je skutečný a ukončit vyhledávání. Pokud se protíná, je možné prohledat sousední region. Pak se ale už nejedná o aproximační metodu, jelikož algoritmus by vždy našel skutečného nejbližšího souseda. Zde se nabízí použít nějaký rozhodovací mechanismus, který pomůže určit, zda se vyhledávání oplatí. Hraje roli i výsledek, který se od algoritmu požaduje. Pokud je požadovaná vysoká přesnost a ne pouze aproximovaný výsledek pro dotazované $q$, tato metoda prohledávání sousedních regionů může být preferována. Dále však nebude nijak rozváděna, je zde uvedena pouze jako zajímavost.

- Druhou a v praxi preferovanou možností je použít vícero rozdělení hyperrovinami $l$. Vyhledávání se provede nad všemi prostorovými rozděleními a nejlepší výsledek algoritmus vrací jako výsledek hledání. Parametr $l$ by bylo vhodné zvolit podle pravděpodobnosti, že se NN nachází ve stejném regionu jako $q$. Tuto pravděpodobnost ale nelze obecně spočítat, protože záleží na počtu hyperrovin (granularity prostoru), pravděpodobnostním rozdělení dat a jejich hustotě (řídký datový soubor x hustý datový soubor). To ale nepředstavuje velký problém, protože s vyšším $l$ pravděpodobnost na-

lezení NN rychle stoupá. Je nutno opět brát zřetel na to, že cílem nemusí být skutečný NN, ale pouze aproximace. Aproximovaný NN tedy může stačit.

Pro ukázku uvažujme různé pravděpodobnosti, že při náhodném rozdělení prostoru podle vygenerovaných hyperrovin jsou $q$ a skutečný nejbližší soused ve stejném regionu: $p_1 = 0.1$, $p_2 = 0.3$, $p_3 = 0.5$, $p_4 = 0.7$, a $p_5 = 0.9$.

Dále rozdělme prostor $m$ náhodnými hyperrovinami ne jednou, ale $l$-krát. Algoritmu stačí pouze u jednoho z $l$ rozdělení najít skutečného nejbližšího souseda, protože se výsledky z $l$ hledání porovnají a vybere se ten nejlepší.

| | l=1 | l=2 | l=3 | l=4 | l=5 | l=6 | l=7 | l=8 |
|---|---|---|---|---|---|---|---|---|
| $P_1$ | 0.1 | 0.19 | 0.271 | 0.3439 | 0.40951 | 0.46855 | 0.5217 | 0.56954 |
| $P_2$ | 0.3 | 0.51 | 0.657 | 0.7599 | 0.83193 | 0.88235 | 0.91764 | 0.94239 |
| $P_3$ | 0.5 | 0.75 | 0.875 | 0.9375 | 0.96875 | 0.98438 | 0.99219 | 0.99609 |
| $P_4$ | 0.7 | 0.91 | 0.973 | 0.9919 | 0.99757 | 0.99928 | 0.99978 | 0.99993 |
| $P_5$ | 0.9 | 0.99 | 0.999 | 0.9999 | 0.99999 | $\doteq 1$ | $\doteq 1$ | $\doteq 1$ |

Tabulka 3.1: Každým dalším rozdělením prostoru $l+1$ narůstá pravděpodobnost, že alespoň v jednom z nich bude nejbližší soused nalezen. Jedná se o binomické rozdělení $Bi(n, p)$, kde šance úspěchu $p$ je jedno z $p_1$–$p_5$, počet pokusů $n$ je roven parametru $l$ a pravděpodobnost se počítá pro alespoň jeden úspěch $P(X >= 1)$.

Výsledkem je pravděpodobnost nalezení skutečného, neaproximovaného nejbližšího souseda. Pomocí experimentů se dá před reálným použitím zjistit pravděpodobnost $p$. Zvolením povolené tolerance chybovosti (error rate) pro aproximační metodu se za použití binomického rozdělení dá určit parametr $l$, které se pro datovou sadu použije. Z tabulky 3.1 je zřejmé, že se zpravidla nebude jednat o vysoký parametr.

### 3.2.1 Konstrukce

Před vyhledáváním $q$ v datové sadě je nejprve nutno rozdělit prostor na regiony a každému regionu inicializovat pole. To je posléze naplněno daty náležícími danému regionu. Přidávání, úprava nebo mazání prvků datové sady je jednoduché, jelikož algoritmus nepoužívá žádné speciální datové struktury pro uložení dat kromě pole.

Příslušnost bodu k regionu se určuje podle pozice bodu k jednotlivým hyperrovinám. Každá hyperrovina rozděluje prostor na 2 subprostory. Jeden z nich bude značen 0, druhý 1 (nebo 1 a 2, $A$ a $B\ldots$). Vypočítáním vztahu bodu ke všem rovinám vznikne řetězec bitů „$aaaa$“, kde $a \in \{0, 1\}$ a $i$-tý znak značí vztah k $i$-té hyperrovině. Pro $m$ hyperrovin existuje $2^m$ možných řetězců délky $m$. Každému regionu náleží jeden řetězec, existují řetězce bez regionů protože počet regionů $\frac{m*(m+1)}{2}+1$ je vždy menší nebo roven počtu možných řetězců $m^2$, pro libovolné $m \in \mathbb{N}$.

Je vytvořena jedna hashovací tabulka, pro kterou jsou řetězce klíčem a návratovými hodnotami jsou pole dat tvořící daný region. Z této tabulky je odvozen název algoritmu. **Podle lokality bodu (locality-sensitive) je vytvořen hash (hashing), který v tabulce odkazuje na pole regionu**. Tato datová struktura pouze slouží pro adresování polí s daty. Tabulka je vytvořena pro každé rozdělení prostoru $l_i$, tedy parametr $l$ určuje mimo jiné i počet hashovacích tabulek. Když jsou v každé tabulce přiřazeny všechny data, inicializační práce jsou dokončené a je možné provádět vyhledávání nejbližšího souseda.

**Algorithm 2** Pseudokód Inicializace Locality-Sensitive Hashing

```
function constructLSH(DataSet S, int l)
    dimesnsions ← getNumberOfDimensions(S)
    setups ← setup[l]                                   ▷ declare l setups
    for all iinl do
        hyperplanes ← generateHyperplanes(dimensions, M)
        hashTable ← createHashTable(hyperplanes)
        for all datainS do
            hash ← createHash(hylerplanes, data)
            push(hashTabe[hash], data)
        end for
        setups[i].hashTable ← hashTable
        setups[i].hyperplanes ← hyperplanes
    end for
    return setups
end function
```

### 3.2.2 Vyhledávání

Samotné vyhledávání probíhá následovně:

1. Pro $q$ se zjistí vztah k jednotlivým hyperrovinám.
2. Vyskládá se řetězec, který se použije jako klíč do hashovací tabulky, obsahující pole dat jednotlivých regionů.
3. Vrácené pole se lineárně prohledá a nalezne se aproximovaný nejbližší soused.
4. Provede se u každého rozdělení $l_i$.
5. Nejlepší výsledek (nejbližší aproximovaný soused) je výsledkem celého algoritmu.

**Algorithm 3** Pseudokód Vyhledávání Locality-Sensitive Hashing

```
function searchLSH(Node q, Setup setups[])
    results ← Node[length(setups)]
    resultDistances ← int[length(setups)]
    for all i, setupinsetups do
        hashTable ← setup.hashTable
        hyperplanes ← setup.hyperplanes
        qHash ← createHash(hyperplanes, q)
        region ← hashTable[qHash]
        nearest ← region[0]                    ▷ caution: region may contain 0 points
        nearestDist ← d(nearest, q)
        for all datainregion do                                  ▷ linear algorithm
            dist ← d(data, q)
            if dist < nearestDist then
                nearest ← data
                nearestDist ← dist
            end if
        end for
        results[i] ← nearest
        resultDistances[i] ← nearestDist
    end for
    bestDist ← min(resultDistances)
    return results[indexOf(bestDist)]
end function
```

### 3.2.3 Složitosti

V složitosti algoritmu sehrává roli několik parametrů:

- Počet bodů v datové sadě $n$
- Počet dimenzí $D$
- Počet hyperrovin $m$
- Počet opakování rozdělení prostoru $l$

#### Konstrukce

Prostorová složitost roste s použitím $l$ tabulek, každá tabulka obsahuje celý datový soubor. $l$ je ale pouze multiplikační faktor proto prostorová složitost zůstává lineární (i když $l$ stojí za povšimnutí) $\mathcal{O}(ln) \approx \mathcal{O}(n)$.

Konstrukce sestává ze dvou kroků – generování hyperrovin a vytvoření hashovacích tabulek. Generování hyperrovin je v podstatě generování $D + 1$ náhodných čísel pro každé z $l$ rozdělení prostoru, složitost je $\mathcal{O}(l(D + 1))$.

Přiřazení bodů hashovacím tabulkám znamená vypočítání hashe pro jednotlivé body. Nalezení hashe pro $n_i$ zahrnuje výpočet vztahu ke každé hyperrovině $m_j$, výpočet vztahu je skalární součin vektorů $n_i$ a $m_j$ (bod v datové sadě se dá vyjádřit jako vektor hodnot v jednotlivých dimenzích). Vektory jsou $D$-rozměrné, tedy vyžadují $D$ operací. Časová

složitost hash funkce je $\mathcal{O}(Dm)$. Vypočítání hashe a přiřazení do tabulek pro všechny body ve všech variantách $l$ má tedy notaci $\mathcal{O}(Dmln)$.

$$\mathcal{O}(l(D+1)+l(Dmn)) = \mathcal{O}(l(D+1+Dmn)) \tag{3.1}$$

Časová složitost generování hyperrovin s časovou složitostí vytvoření hash tabulek spolu tvoří časovou složitost konstrukce. $D$, $m$, $l$ a 1 jsou konstanty, složitost je tedy lineární $\mathcal{O}(n)$.

#### Vyhledávání

Nejdříve je nutné najít pomocí hash funkce správnou tabulku k prohledání, poté lineárně prohledat daný prostor.

$$\mathcal{O}(l(Dm + \frac{D*n}{\frac{m*(m+1)}{2}+1})) = lDm + \frac{l*D*n}{\frac{m*(m+1)}{2}+1} \tag{3.2}$$

Nalezení hashe pro $q$, stejně jako u ostatních bodů datové sady, zahrnuje výpočet vztahu ke každé hyperrovině $m_i$ Časová složitost hash funkce je $\mathcal{O}(Dm)$. Průměrný počet prvků v regionu, který bude lineárně prohledáván je $\frac{n}{\frac{m*(m+1)}{2}+1}$. Zjištění vzdálenosti mezi $q$ a prvkem v regionu je skalární součin s $D$ operacemi. Celý výpočet je prováděn $l$-krát.

$$lD\log n + \frac{l*D*n}{\frac{\log n*(\log n+1)}{2}+1} \tag{3.3}$$

Časová složitost, pokud zvolíme počet hyperrovin $m = \log n$.

$D$ a $l$ budou vyloučeny 3.3, protože jsou konstanty. Důležité je to, jak roste komplexita ve vztahu ku $n$. Z pravé strany součtu se stává konstanta, jelikož dělenec i dělitel jsou řádu $n$. Zůstane tedy $\mathcal{O}(\log n)$.

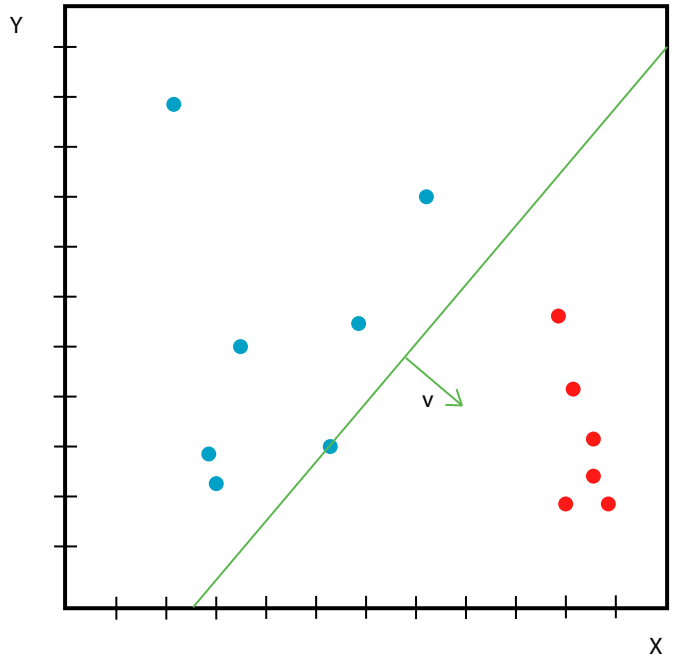
### 3.2.4 Hardware urychlení

Zde se nabízí myšlenka vyhledávat v každé z $l$ tabulek zvlášť. Každá nese vlastní data a jedná se pouze o čtení, nehrozí proto žádný z problémů, které se s paralelními algoritmy obvykle pojí. V každém $l_i$ nastavení z $l$ se vyhledá aproximovaný nejbližší soused a výsledky se agregují pomocí operace $min()$. Stejný princip se dá uplatnit i ve vnitřním cyklu vyhledávacího algoritmu, kdy se mezi výpočetní jednotky rozdělí data z regionu.

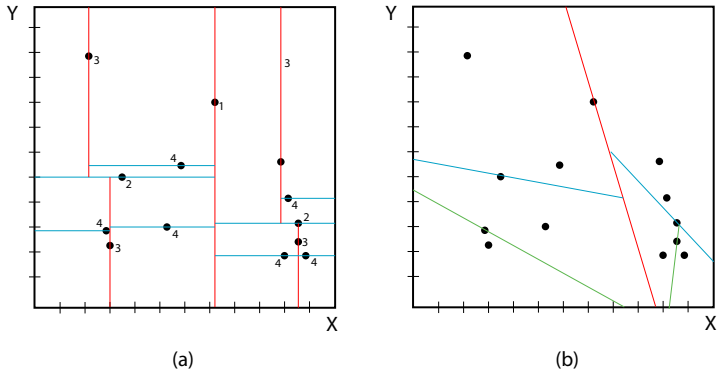
## 3.3 Strom náhodných projekcí

Mezi KD-Stromem a LSH vznikla metoda, která kombinuje tyto dva přístupy. Z KD-Stromu přebrala stromovou strukturu, z LSH techniku projekce bodů na rozdělující hyperrovinu 3.8. Strom náhodných projekcí je binární vyhledávací strom, který buduje levý a pravý podstrom podle toho, na které straně rozdělující hyperroviny body leží. Strom je vyvážený (v základní podobě konstrukčního algoritmu), protože hyperrovina prochází mediánem množiny bodů

$S$, nad kterou je strom konstruován. Zvolí se náhodný jednotkový vektor $v$ 3.7, vůči kterému se spočítá skalární součin všech bodů z datové množiny. Z výsledků se vybere medián, který slouží jako rozhodovací hranice pro patřičnost bodů levému nebo pravému potomkovi.




Obrázek 3.7: Rozdělení prostoru pomocí náhodného vektoru $v$.




Obrázek 3.8: Rozdíl v konstrukci KD-Stromu a stromu náhodných projekcí. Při backtrackingu a hledání možného lepšího souseda metoda netrpí stejným problémem s prokletím dimenzionality jako KD-Strom 3.1.2.

### 3.3.1 Konstrukce

Používají se dva přístupy tvorby stromu [7]. Oba mají své teoretické opodstatnění a přizpůsobují se vnitřním dimenzím[2].

[2]Minimální počet proměnných, kterými se dá popsat prostor. Zkoumaný prostor popsaný $D$ dimenzemi může být dobře popsán méně dimenzemi. $D$ není nutně nejmenší počet dimenzí, které jsou pro data po-

Základ konstrukce stromu je stejný 4, liší se pouze v rozdělení množiny bodů do podstromů.

**Algorithm 4** Pseudokód konstrukce Stromu náhodných projekcí

**function** BUILDTREE(Set S)
  **if** $|S| \leq leafSize$ **then**
    **return** LEAF(S)
  **end if**
  $rule \leftarrow$ CHOOSERULE($S$)
  $left \leftarrow$ BUILDTREE($\{x \in S : rule(x) = true\}$)
  $right \leftarrow$ BUILDTREE($\{x \in S : rule(x) = false\}$)
  **return** $(rule, left, right)$
**end function**

Pravidla pro rozdělení jsou RPTree-Max 5 a RPTree-Mean 6.

**Algorithm 5** Pseudokód pravidla rozdělení RPTree-Max

**function** CHOOSERULE(Set S)
  vyber náhodný jednotkový vektor $v \in \mathbb{R}^D$
  vyber náhodný bod $x \in S$, nechť $y \in S$ je nejvzdálenější bod k bodu $x$
  vyber $\sigma$ náhodně mezi $< 1, -1 > \cdot 6\frac{\text{DIST}(x,y)}{\sqrt{D}}$
  $rule(x) \leftarrow x \cdot\ v \leq (median(\{z \cdot\ v : z \in S\}) + \sigma)$
  **return** $(rule)$
**end function**

**Algorithm 6** Pseudokód pravidla rozdělení RPTree-Mean

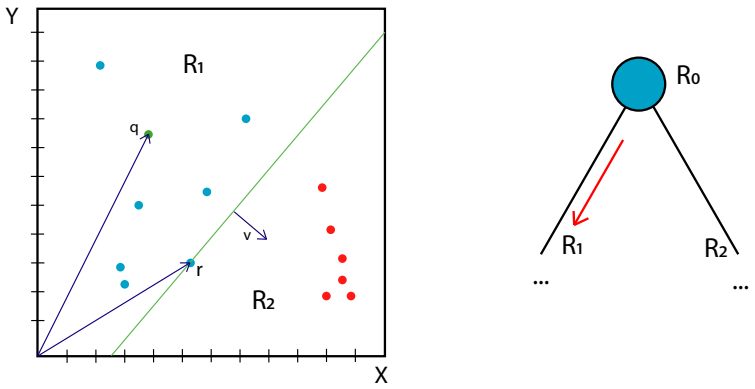
**function** CHOOSERULE(Set S)
  **if** $\Delta^2(S) \leq c \cdot\ {\Delta_A}^2(S)$ **then**
    vyber náhodný jednotkový vektor $v \in \mathbb{R}^D$
    $rule(x) \leftarrow x \cdot\ v \leq median(\{z \cdot\ v : z \in S\})$
  **else**
    $rule(x) \leftarrow \text{DIST}(x, mean(S)) \leq median(\text{DIST}(z, mean(S)) : z \in S)$
  **end if**
  **return** $(rule)$
**end function**

V kódu RPTree-Mean 6 $\Delta(S)$ značí vzdálenost mezi dvěma nejvzdálenějšími body v množině - průměr kružnice opisující data. ${\Delta_A}^2(S)$ značí průměrnou vzdálenost mezi body v množině $S$ a $c$ je konstanta. RPTree-Mean občasně dovoluje jiný druh rozdělení prostoru a to na základě vzdálenosti od průměrného bodu v $S$.

třebné. Počet dimenzí je tedy větší, než vnitřní dimenzionalita dat, což je základ pro transformační techniku dimenzionální redukce. Hodnota vnitřní dimenzionality se používají třeba jako spodní hranice počtu dimenzí pro tuto transformaci.

### 3.3.2 Vyhledávání

Vyhledávání je rekurzivní algoritmus, stejně jako v jiných stromových vyhledávacích strukturách. Každý uzel reprezentuje rozdělovací hyperrovinu (pomocí jednotkového vektoru a bodu, kterým hyperrovina prochází), jeho potomci jsou uzly, které uvažují prostor nad/-pod rozdělovací hyperrovinou [7]. Listové uzly obsahují množinu dat $S$, kde $|S| \leq leafSize$, což je nastavitelný konstrukční parametr. Když se algoritmus rekurzí dostane do listového uzlu, provede naivní vyhledávání nad touto množinou. Při obdržení vyhledávaného bodu se v každé úrovni stromu vypočte skalární součin mezi vyhledávaným bodem a vektorem uzlu 3.9. Pokud je tento výsledek menší nebo roven skalárnímu součinu vektoru a bodu uzlu, vyhledávání pokračuje v podstromu reprezentujícím prostor pod rozdělovací hyperrovinou. Pokud je vyšší, algoritmus pokračuje druhým podstromem.




Obrázek 3.9: Ukázka jednoho kroku sestoupení v Stromu náhodných projekcí s vyhledávaným bodem $q$ a rozdělovací hyperrovinou $(v, r)$. Skalární součin $q \cdot v \leq r \cdot v$, tudíž vyhledávání pokračuje v regionu $R_1$.

Množina dat v listovém uzlu ale nemusí obsahovat skutečné nejbližší sousedy, proto je Strom náhodných projekcí aproximační metoda. Algoritmus pro lepší aproximaci nevyužívá backtrackingu jako KD-Strom. Místo toho se vytváří takzvaný les Stromů náhodných projekcí s parametrem $m$, který značí počet stromů v lese. Jelikož každý strom je konstruován náhodnými procesy, výsledky z jednotlivých stromů se budou lišit a listový uzel, který algoritmus průchodem navštíví, se bude složením prvků lišit od ostatních. Konečná množina pro vyhledávání je $\bigcup_{i=1}^{m} S_i$, sjednocení množin dat listových uzlů všech stromů v lese.

**Stejně jako u LSH, spíš, než korekční mechanismus se používá škálování metody do šířky**. Dovoluje vytvoření jednoduché datové konstrukce, která ani nemusí být tak dobrá v aproximaci, jako nabízet jednoduše spočítatelný výsledek. Takové konstrukce se škálováním do šířky dají využít v kombinaci s vícero výpočetními jednotkami a dovolují urychlení vyhledávání pomocí paralelismu.

### 3.3.3 Složitosti

Strom náhodných projekcí je podobná datová struktura KD-Stromu, proto jsou asymptotické složitosti stejné.

### Konstrukce

Konstrukce stromu má stejnou časovou asymptotickou složitost jako KD-Strom, tedy $\mathcal{O}(n)$. Procedura konstrukce je totiž až na pravidla rozdělení 3.3.1 množiny na subprostory stejná. Jak u KD-Stromu, tak v Stromu náhodných projekcí se vyhledá medián v každé úrovni podstromu. Tentokrát ne však medián hodnoty nějaké jedné dimenze, ale $median(\{z \cdot v : z \in S\})$ nebo $median(\text{DIST}(z, mean(S)) : z \in S)$ pro RPTree-Mean, kde $S$ je množina dat prostoru a $v$ je náhodný jednotkový vektor. Posléze se data podle mediánu a použitého rozdělovacího pravidla přiřadí subprostorům.

Prostorová složitost, znovu stejně jako u KD-Stromu, nenarůstá oproti složitosti naivní metody. Les obsahuje konstantu $m$, počet stromů, která se vylučuje $\mathcal{O}(mn) \approx \mathcal{O}(n)$.

### Vyhledávání

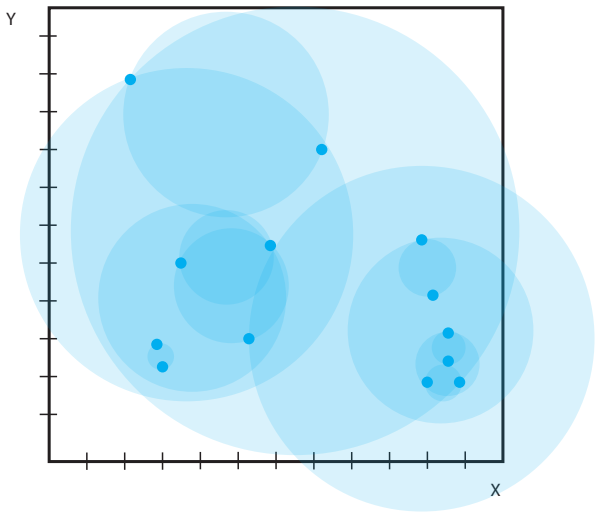
Vyhledávání má časovou složitost $\mathcal{O}(m \log n) \approx \mathcal{O}(\log n)$, kde $m$ je zmíněná konstanta počtu stromů v lese. S počtem dat tedy škáluje logaritmicky. Je to z důvodu, stejně jako u jiných stromových vyhledávacích struktur, že algoritmus při každé redukci prostoru přestane uvažovat polovinu datových bodů.

## 3.4 Kulovitý strom

Podobně jako KD-Strom nebo Strom náhodných projekcí, Kulovitý strom je také binární vyhledávací stromová struktura [25]. Mají v podstatě stejný přístup k vyhledávání NN, jediný rozdíl v základní myšlence je opět relaxace omezení, kdy v KD-Stromu v každém rozdělení dat uvažujeme pouze jednu dimenzi. Střídavě všechny, ale pouze jednu v jednom rozdělení prostoru. Pokud vyhledávání probíhá v nízko dimenzionálním prostoru, toto omezení není žádný problém. Ten nastává, když vyhledávání probíhá ve vysoce dimenzionálním prostoru, kde KD-Strom ztrácí efektivitu a jeho vyhledávací čas se blíží k lineárnímu algoritmu, protože je náchylný na prokletí dimenzionality 2.1.2. Je nutné podotknout, že Kulovitý strom také trpí na prokletí dimenzionality, jako všechny algoritmy používající vzdálenostní funkce, je ale vůči problému robustnější metodou.

Kulovitý strom při dělení prostoru uvažuje všechny dimenze. Shluk podobných dat modeluje pomocí hyperkoulí 3.10, což jsou obecně koule v $D$-dimenzionálním prostoru, kde $D \in \mathbb{N}$. V modelu každá taková hyperkoule obsahuje dva potomky, pokud je splněna podmínka pro rozdělení prostoru. Podmínka může mít formu maximální hloubky stromu, minimálního počtu datových vzorků pro rozdělení nebo dokud není v každém listovém uzlu jeden nebo dva body.

Každá úroveň Kulovitého stromu je významově stejná - formuje množinu dat v určité vzdálenosti od nějakého středového bodu. Každý uzel nese informaci o středovém bodu a poloměru hyperkoule ohraničující uzlový prostor.




Obrázek 3.10: Vizuální reprezentace Kulovitého stromu nad daty. Povšimněme si, že na rozdíl od jiných algoritmů a struktur pro dělení prostoru se subprostory mohou prolínat. Každý subprostor zachycuje část z každé dimenze (v každé dimenzi dochází k redukci prostoru). Kořenový uzel stromu reprezentovaný největším kruhem, v literatuře značený $O$, se často z vizuální reprezentace vynechává. Na obrázku se prostor se dělil pokud uzel měl víc než 2 potomky.

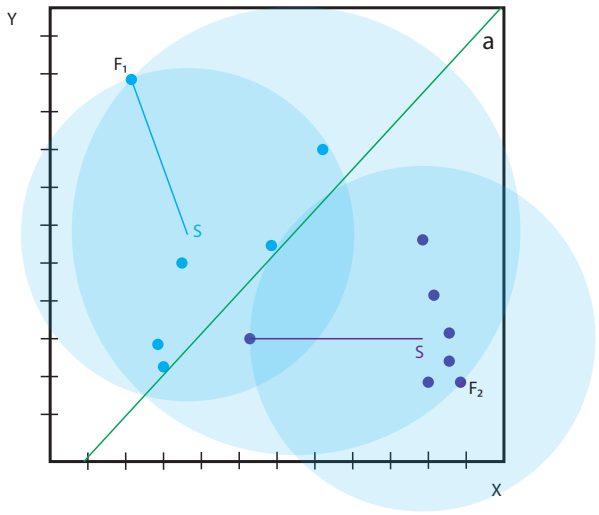
### 3.4.1 Konstrukce

Existuje vícero možností jak konstruovat datovou strukturu pro algoritmus Kulovitého stromu. Většina z nich má přístup shora–dolů, kde se nejdříve konstruují nejvyšší úrovně stromu a konstrukce sestupuje k nižším úrovním. Obecně, v každé úrovni se najde rozdělovací hyperrovina, která určí příslušnost bodů nově vznikajícím potomkům podle jejich relativní polohy k ní. Hyperrovina se konstruuje následovně [9]:

1. Vybere se náhodný bod z uvažovaného prostoru.
2. Najde se k němu nejvzdálenější bod $F_1$.
3. K bodu $F_1$ se najde nejvzdálenější bod $F_2$. Tímto jednoduchým postupem se získají dva velice vzdálené body.
4. Osa $a$ úsečky $F_1F_2$ je rozdělovací hyperrovina.

Po získání rozdělující hyperroviny se vypočítá:

- Příslušnost bodů potomkům a výsledné dvě podmnožiny bodů.
- Pro každou podmnožinu geometrický střed, centroid.
- Pro každou podmnožinu vzdálenost od centroidu k nejvzdálenějšímu bodu z podmnožiny, poloměr nové hyperkoule.




Obrázek 3.11: Konstrukce potomků. $F_1$ a $F_2$ jsou body získané postupem hledání vzdálených bodů 3. $a$ je osa úsečky $F_1F_2$, podle které se body rozdělí mezi potomky. $S$ jsou značeny centroidy podmnožin, úsečkou je znázorněn poloměr hyperkoulí, vzdálenost od centroidů k nejvzdálenějším bodům podmnožiny.

Konstrukce 3.11 se rekurzivně opakuje pro oba potomky. Rekurze končí, když je splněna podmínka zanoření 3.4. Představený algoritmus nevede k vyváženému stromu, což je problém, pokud se vyžaduje stabilní vyhledávací složitost. Vyvážený strom se dá konstruovat tak, že se promítnou všechny body na úsečku $F_1F_2$ a jako rozdělovací hyperrovina se nezvolí osa úsečky $a$, ale kolmice procházející mediánem promítnutých bodů.

Dále se v konstrukci neuvažuje rozložení bodů. Tento problém se dá adresovat pomocí analýzy hlavních komponent [9]. Zjistí se vektor $v$, ve kterém je v datech největší variabilita a na něj se všechna promítnou. Vybere se medián a body se přiřadí do levého nebo pravého podstromu podle pozice jejich promítnutí od mediánu. Tímto konstrukčním postupem se vytvoří vyvážený strom s ohledem na rozložení bodů a vyváženost stromu.

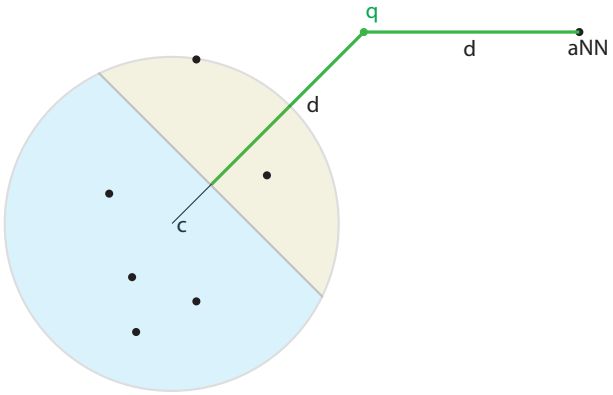
**Algorithm 7** Pseudokód rozdělení množiny pomocí analýzy hlavních komponent

**function** SPLIT(X)
  $v_1 \leftarrow \arg\max \{\frac{v^T X^T v}{v^T v}\}$ ▷ PCA eigenvector
  $T \leftarrow v_1.X$ ▷ Project the points on the vector
  $T_c \leftarrow \text{MEDIAN}(T)$
  $X_l \leftarrow \{x_i | x_i \in X, t_i < t_c\}$ ▷ $t_i$ is $x_i$ projected
  $X_r \leftarrow \{x_i | x_i \in X, t_i \geq t_c\}$
  **return** $X_l, X_r$
**end function**

### 3.4.2 Vyhledávání

Data v Kulovitém stromu se uchovávají v listových uzlech. Vnitřní uzly pouze navigují algoritmus k těmto uzlům [25]. K tomu používají první ze dvou informací vnitřního uzlu, centroid. Při sestupu do nižší úrovně se vypočítá vzdálenost od vyhledávaného bodu k centroidům potomků a vyhledávání pokračuje potomkem, který má bližší centroid. Po dosažení

listového uzlu se vybere nejbližší z $n$ dat ($|n| \leq leafsize$), které uzel obsahuje a prohlásí se aproximovaným NN. Kulovitý strom je metoda prohledávání do hloubky, která při vynořování navštěvuje dříve nenavštíveného potomka pokud algoritmus usoudí, že se v něm může nacházet bližší objekt. K tomu slouží druhá uchovávaná hodnota v uzlu – poloměr hyperkoule. Pokud vzdálenost od vyhledávaného bodu k povrchu hyperkoule druhého potomka je menší, než vzdálenost od vyhledávaného bodu k aproximovanému NN, algoritmus sestoupí do druhého potomka 3.12. V podstromu totiž (mimo jiné možné blízké body) leží alespoň jeden bod, který je na povrchu hyperkoule.




Obrázek 3.12: Znázornění stavu, kdy se algoritmus rozhodne sestoupit do druhého, dříve nenavštíveného podstromu. Bod $aNN$ je aproximovaný NN, $q$ je vyhledávaný bod, $c$ je centroid druhého potomka. Pokud existuje znázorněná výseč, v hyperkouli se mohou nacházet bližší body. Existence výseče se vypočítá jako $d(c,q) - radius < d(q,aNN)$, kde $radius$ značí poloměr hyperkoule druhého potomka. Použití této podmínky pro sestup zaručuje nalezení skutečného NN.

**Algorithm 8** Pseudokód Vyhledávání Kulovitým stromem [25]

```
function searchBTree(BTree t, Point q, Point best)
    if D(t.centroid, q) − t.radius ≥ D(q, best) then
        return best
    end if
    if isLeaf(t) then
        for all Pointpint.points do
            if D(p, q) < D(best, q) then
                best ← p
            end if
        end for
    else
        close ←the child subtree of t closest to q
        far ←the child subtree of t furthest to q
        searchBTree(close, q, best)
        searchBTree(far, q, best)
    end if
    return best
end function
```

### 3.4.3 Složitosti

Prostorová složitost v Kulovitém stromu nijak nenarůstá, protože nedochází k replikaci dat. Data jsou uložena v listech binárního stromu, který má prostorovou složitost $\mathcal{O}(n)$.

Časová složitost rozdělení dat potomkům je i po úpravě a použití analýzy hlavních komponent stejná jako v neupravené verzi. Pokud je počet dat $n$ a počet dimenzí $D$, výpočet rozdělovací hyperroviny pomocí analýzy hlavních komponent je $O(D^2n+D^3)$ [3]. V závislosti na datech je časová složitost $\mathcal{O}(n)$, protože $D$ je konstanta. Takové rozdělení proběhne v $log(n)$ úrovních, čili časová složitost konstrukce je $\mathcal{O}(n \log n)$.

Časová složitost vyhledávání ve vyváženém stromu je $\mathcal{O}(\log n)$. Hlavní myšlenkou algoritmu, stejně jako u KD-Stromu a Stromu náhodných projekcí, je ořezávání větví, které představují části prohledávaného prostoru. Stejně ale jako u jiných algoritmů používajících vzdálenostní funkci, i zde nastává zhoršení výsledků zvyšováním počtu dimenzí. Kulovitý strom je v porovnání s KD-Stromem na počet dimenzí méně náchylný. V nejhorším případě může algoritmus navštívit všechny větve a časová složitost by byla $\mathcal{O}(n)$, stejná jako v lineární verzi.

## 3.5 Algoritmy hledání KNN založené na grafu

Doposud představené algoritmy používaly ke zkrácení času vyhledávání nějaký způsob dělení prostoru. Ať už rekurzivní dělení vedoucí k stromové struktuře definující části prostoru, nebo rozdělení hyperrovinami u LSH. Menší uvažovaný prostor znamená méně dat k prohledání. Tento přístup je jeden z dvou hlavních, přičemž druhý je založen na grafu z uvažovaných bodů. Grafové algoritmy naznačují superioritu v přesnosti výsledků (recall) [2].

Jedná se o grafy formující síť nejbližších sousedů, nebo jeho aproximaci. Každý bod je spojen orientovanou váženou hranou (váha hrany mezi libovolnými body $r$ a $q$ je $d(q, r)$) s jeho $M$ nejbližšími sousedy. Směrovaná proto, že vztah nejbližší sousednosti není symetrický, tudíž bod $q$ může být nejbližším sousedem bodu $p$, naopak tomu ale tak být nemusí.

Při vyhledávání se zvolí startovací bod, od kterého se postupně procházením grafu algoritmus dostává k okolí vyhledávaného bodu. Průchod grafem je rekurzivní volba z bodů, kterými je startovací bod spojen, podle jejich vzdálenosti od vyhledávaného bodu. Ze spojených bodů je vybrán ten, který je k vyhledávanému bodu nejblíž a stává se novým startovacím bodem. Proces je v podstatě hill-climbing algoritmus, který hledá globální minimum vzdáleností funkce. Okolí bodu, které hill-climbing volí k prohledání, jsou nejbližší sousedé bodu v rekurzi.

Existuje řada grafových algoritmů pro KNN, všechny ale sdílí stejnou myšlenku a pouze upravují dílčí části. Třeba použijí nesměrované hrany, heuristicky vytvoří aproximovaný graf nejbližších sousedů nebo použijí jiný než hladový algoritmus pro průchod grafem. V základní podobě řešení problému vypadá následovně:

1. V offline fázi se nad daty vytvoří graf nejbližších sousedů.
2. Obdrží bod, pro který se v grafu budou vyhledávat nejbližší sousedé.
3. Náhodně se zvolí startovací bod.
4. Rekurzivně se z jeho nejbližších sousedů volí ten nejbližší k bodu hledání, který se stává novým startovacím bodem.

5. Vyhledávání končí, když žádný z kandidátních bodů přímo nepřinese výsledek lepší, než už získaný (lokální minimum).

### 3.5.1 Konstrukce

V offline fázi 1 se vytvoří graf pomocí lineárního algoritmu. Ten zaručí správnost grafu, je ale časově velmi náročný, $\mathcal{O}(n^2)$. Pro účely vyhledávání nad grafem ale není potřeba mít přesný graf nejbližších sousedů, algoritmus to nijak nevyžaduje, je to ale pro něj dobrý základ. Ve skutečnosti přesný a pouze graf nejbližších sousedů není preferován a je záměrně upravován pro získání lepších výsledků. Jedním z příkladů takové úpravy je přidání odkazů na dlouhé vzdálenosti. Jedná se o úpravu, kdy náhodným bodům jsou přidělovány hrany ke vzdáleným bodům, což může vést ke zrychlení v prvních fázích vyhledávání. Nejjednodušší možností takové konstrukce je ke každému bodu přidělit hranu k náhodným bodům, přičemž každý bod bude mít stejnou pravděpodobnost přidělení - rovnoměrná pravděpodobnost. Každý bod bude mít hrany k nejbližším sousedům a vzdáleným bodům. Nejdříve algoritmus zkusí použít hrany propojující vzdálené body, aby se dostal do bližšího okolí rychle, posléze pokračuje pouze hranami nejbližších sousedů. Hrany na dlouhé vzdálenosti jsou základní myšlenkou Hierarchical Navigable Small World (HNSW) grafů. HNSW patří mezi oblíbené moderní grafy pro vyhledávání nejbližších sousedů.

Další vylepšení ve fázi vytvoření grafu je odstranění některých hran z grafu s neorientovanými hranami. Odstranění hran řeší problémy pouze u neorientovaného grafu, který se obdrží tak, že se sestrojí orientovaný graf a posléze se jeho hrany zamění za neorientované. Smysl neorientovaného grafu je ten, že některé grafové algoritmy vyžadují buď neorientovaný graf nebo graf symetrický a také ten, že použitím i opačných hran se může zvýšit přesnost vyhledání, tudíž mohou být užitečné. Vlastnost „býti nejbližším sousedem“ je nesymetrická. Odstraněním hran se vlastnost modelována hranami stává symetrickou. Orientovaná verze grafu měla jistotu, že z každého bodu bude vést přesně $k$ orientovaných hran. Odstraněním orientace přibudou bodu $q$ hrany $(q, r)$, kde $q$ bylo $i$-tým ($i \leq k$) nejbližším sousedem $r$. To vede k vzniku takzvaných HUBů. Jejich výskyt se pojí s vysoko dimenzionálními daty. Radovanovic [26] označuje vlastnost datového setu „hubness“; vysoká četnost výskytu nějakého konkrétního bodu jako jednoho z nejbližších sousedů pro mnoha ostatních bodů, značená $k$-výskyt. S nárůstem dimenzí se $k$-výskyt a počet HUBů zvyšuje, což autor pokládá za příčinu prokletí dimenzionality a důvodu, proč algoritmus nejbližších sousedů obecně nefunguje ve vysoko dimenzionálních prostorách. Když algoritmus v průchodu grafem narazí na HUB, přidá tím veliké množství kandidátů pro pokračování průchodu, kterým musí být vypočtena vzdálenost k vyhledávanému bodu a pak seřazeny.

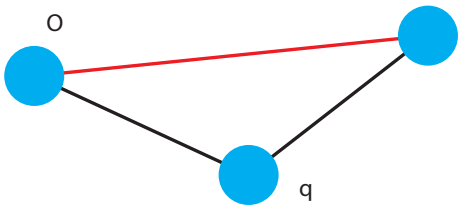
Po konstrukci je možno:

- Použít horní hranici pro $M$ v neorientovaném grafu. Když nějaký bod přesáhne počtem hran horní hranici, je považován za HUB a z grafu odstraněn. Kohei Ozaki [21] použil tuto metodu, kdy z orientovaného grafu nejbližších sousedů s $M = 3$ vytvořil graf neorientovaný a následně odstranil uzly s $M \geq 30$ hranami. Odstranění přineslo lepší výsledky, čímž empiricky potvrzuje škodlivý vliv HUBů na algoritmus.

- Použít horní hranici pro $M$ v neorientovaném grafu, tentokrát zachovat bod a pouze top $M$ (nejkratších) hran a odstranit ostatní.

- Odstranit nejdelší hrany v trojúhelníku 3.13. Trojúhelníkem v grafu se myslí nejmenší možný cyklický podgraf. To se zajistí jednoduchým algoritmem. Pro každý bod (referovaný jako originální):

1. Získat jeho nejbližšího souseda $q$ z výčtu jeho nejbližších sousedů, který už je dostupný.
2. Zanechat hranu mezi originálním bodem a tímto sousedem, označit bod jako „zachovaný“.
3. Pro každého jiného souseda originálního bodu:
   (a) Pokud je soused blíže k originálnímu bodu, než ke $q$, označit tento bod jako „zachovaný“ také.
   (b) Pokud je soused blíže ke kterémukoliv „zachovanému“ bodu, než k bodu originálnímu, odstranit hranu mezi originálním bodem a sousedem.




Obrázek 3.13: Trojúhelník v grafu. $o$ je originální bod, $q$ je jeho nejbližší soused a třetí bod je jeden ze společných sousedů. Dlouhá červená hrana je redundantní, protože algoritmus dokáže projít graf přes všechny tři uzly použitím krátkých hran. Použije při tom sice o 1 krok průchodu navíc, odstraněním dlouhých hran se ale snižuje počet kandidátů pro postup algoritmu. Odstranění má pozitivní vliv na časy procházení. Tento proces uchová hrany mezi nejbližšími sousedy, postupně ale odstraňuje dlouhé redundantní hrany. Hrany na dlouhé vzdálenosti 3.5.1 ale ovlivněny nebudou, jelikož zpravidla netvoří trojúhelníky v grafu.

Vytváření grafu nejbližších sousedů je výpočetně náročný úkon. Pro adresování tohoto problému se používá aproximace grafu nejbližších sousedů. Jedna z možných konstrukčních taktik je využití předpokladu „sousedé mých sousedů jsou pravděpodobně taky mí sousedé“. Inkrementální taktika, která začíná s grafem, který je lehký na konstrukci. Buď se může použít úplně náhodný graf, kde každý bod je spojen s náhodnými $M$ dalšími body, nebo se pro tvorbu výchozího grafu použije Strom náhodných projekcí.

V obou případech hrají roli sousedé sousedů každého bodu $q$, kteří jsou potencionální kandidáti býti lepším sousedem pro $q$ než jeho originální sousedi v nedokonalém, rychle vytvořeném grafu.

1. Vytvoření náhodného grafu nebo využití Stromu náhodných projekcí pro rychlé vytvoření slabé aproximace grafu nejbližších sousedů.
2. Pro každý bod:
   (a) Změřit vzdálenosti od bodu ke všem sousedům jeho sousedů. Pokud je nějaký z nich blíž, než aktuální sousedi, aktualizovat graf. Sousedé sousedů jsou v podstatě chápáni jako vlastní. Spolu s reálnými vlastními sousedy vytvoří jeden seznam, ze kterého se po seřazení ponechá pouze $k$ nejbližších.
   (b) Pokud nastala změna v grafu, vrať se k bodu 2, jinak skonči konstrukci grafu.

Autoři *NNDescent* algoritmu ukazují, že tento konstrukční algoritmus je velmi efektivní. V jejich ukázce z náhodného grafu měří přesnost správného přiřazení sousedů, výsledky jsou shrnuty v tabulkách 3.2, 3.3.

| **Iterace** | 0. | 1. | 2. | 3. | 4. | 5. |
|---|---|---|---|---|---|---|
| **Přesnost** | 1.12 % | 36.6 % | 95.76 % | 99.28 % | 99.48 % | 99.6 % |

Tabulka 3.2: Přesnosti grafu při inicializaci náhodným grafem. Uvažováním „sousedé mých sousedů jsou pravděpodobně taky mí sousedé“ a postupnou úpravou grafu se pokaždé vycházelo z lepší aproximace, než předtím. Přesnost 1.12 % jsou náhodně správně přiřazení sousedé.

| **Iterace** | 0. | 1. | 2. |
|---|---|---|---|
| **Přesnost** | 70.32 % | 98.56 % | 99.92 % |

Tabulka 3.3: Přesnosti grafu při inicializaci použitím náhodných projekčních stromů. Autoři dosáhli ještě lepších výsledků s méně iteracemi.

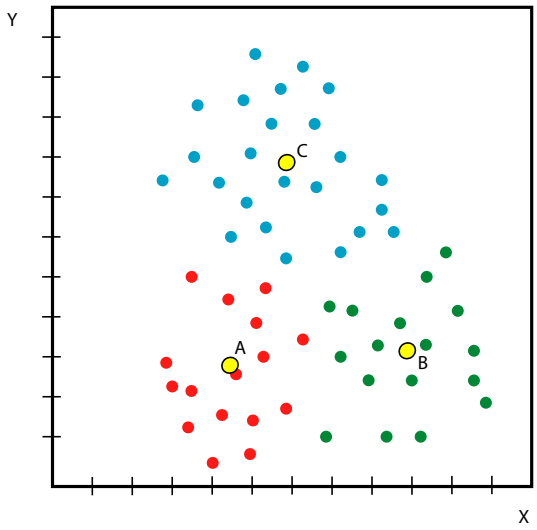
**Algorithm 9** Konstrukce KNN grafu

```
function constructKNNGraph(DataSet S, int M, metric)
    graph ← initEmptyGraph(metric)
    for all data in S do
        graph.addNode(data)
    end for
    for all node in graph.getNodes() do
        neighbours ← naiveKNN(graph.getNodes() − node, node, M, metric)
        node.createAssociations(neighbours)
    end for
    return graph
end function
```

Povšimněme si, že vzdálenostní funkce `metric` je konstrukční parametr 9, ne vyhledávací. Je tomu tak pro to, že konstrukce využívá vyhledávací algoritmus KNN (v pseudokódu naivní) a pro to už musí být poskytnuta. Při vyhledávání se pak vzdálenostní funkce už neuvádí a používá se ta, pomocí které byl graf zkonstruován.

### 3.5.2 Vyhledávání

Ve fázi volby bodu pro start vyhledávání základní algoritmus volí tento bod náhodně. Dříve nebo později, procházením grafu se algoritmus dostane k výsledku 10, zde se ale nabízí použít nějaký způsob lepší inicializace vyhledávání. Využívá se shlukování; nejlépe algoritmus, který vytváří pevný počet shluků jako například k-means 3.14. Cílem není totiž identifikace shluků, ale vytvoření regionů s reprezentací (centroid), které poslouží jako kandidáti pro startovní bod. To, že k-means identifikuje shluky je pouze příjemné vylepšení, není to požadavkem ani nutností. Navíc, ve vysoko dimenzionálních datech se ztrácí sousednost 2.1.2. V takovém případě by shlukovací algoritmy, které detekují počet shluků samy, s vysokou pravděpodobností detekovaly jeden veliký shluk reprezentující celý datový vzorek.




Obrázek 3.14: Metoda shlukování k-means na rovnoměrně rozložená data. Výhodou je, že i u dat, kde nejsou identifikovatelné přirozené shluky vytvoří pevný počet shluků, v tomto případě $k = 3$. Centroidy z k-means jsou tedy buď rovnoměrně rozložené pokud data neformují shluky, nebo opisují středy, pokud data shluky formují. V obou případech jsou dobrými kandidáty pro seed vyhledávání.

**Centroidy z k-means se použijí jako seznam kandidátních seedů**. Seznam se lineárně prozkoumá a vybere se nejbližší centroid k vyhledávanému bodu. Délka seznamu záleží od výběru parametru $k$ u shlukovacího algoritmu. Sluků nemusí být málo. Empiricky se určí ideální počet, což může být v řádu stovek nebo tisíců shluků (záleží na počtu dat).

Selekce seedu pro začátek vyhledávacího algoritmu se dá zlepšit. Mezi seedy se může vytvořit další graf nejbližších sousedů, který je propojuje. Výběrem náhodného seedu a hill-climbing algoritmem se kvalita seedu iterativně zlepšuje až nakonec dojde k nejkvalitnějšímu seedu, který je nejblíž vyhledávanému bodu. Metoda je obecně rychlejší a potřebuje menší počet výpočtu vzdáleností, než naivní algoritmus selekce seedu.

Pokud se zůstane u myšlenky grafu nejbližších sousedů pro selekci seedu, je možné ještě upravit náhodnou inicializaci startovního bodu. Byly navrženy metody [19] s použitím rozdělení prostoru pomocí LSH nebo ITQ[3]. Z vyhledávaného bodu se vypočítá hash a hill-climbing algoritmus se inicializuje jedním ze seedů se stejným hashem za pomocí hashovací tabulky. Inicializační bod má oproti náhodnému výběru stejný region jako vyhledávací bod, tudíž je vyhledávacímu bodu bližší a hill-climbing algoritmus bude potřebovat k dosažení nejkvalitnějšího seedu méně iterací.

Munlika Rattaphun [24] Proces selekce seedu zdokonalil použitím neuronové sítě, která má na vstupu vektor vyhledávaného bodu a výstup jsou pravděpodobnosti příslušnosti bodu k jednotlivým shlukům z k-means podle jejich blízkosti. Výsledný pravděpodobnostní seznam se seřadí sestupně a navrátí se tzv. top-$R$ shluk, který se použije jako inicializační seed pro hill-climbing algoritmus.

[3]Podobně jako LSH, ITQ (Iterative Quantization) je metoda tvorby hashe podle lokality v prostoru.

**Algorithm 10** Vyhledávání v KNN grafu

```
function KNNGraphQuery(Graph graph, Point query)
    i ← graph.randomPoint()
    while dist(q, i) ≤ dist(q, s), s ∈ i.neighbours do
        neighbours ← i.neighbours
        for all s in neighbours do
            if dist(q, s) ≤ dist(q, i) then
                i ← s
            end if
        end for
    end while
    return i
end function
```

### 3.5.3 Složitosti

Při konstrukci se pro každý bod najde $M$ nejbližších sousedů. To vede k složitosti $\mathcal{O}(n^2)$, což není vhodné. Byly navrženy algoritmy konstrukcí grafů, které škálují lépe, třeba rekurzivní Lanczosova bisekce [5], která vytváří aproximovaný KNN graf v čase $\Theta(n^t)$, kde $t \in \{1, 2\}$ je interní parametr konstrukčního algoritmu.

U KNN grafů se složitosti typicky měří empiricky. Pro HNSW byla odměřena složitost vyhledávání [18], která je přibližně stejná s očekávanou $\mathcal{O}\log^2(n)$. Jeden log představuje průměrnou délku cesty průchodu grafem, druhý počet kalkulací vzdálenosti – použití vzdálenostní funkce.

Pro *NNDescent* algoritmus byla naměřena empirická složitost přibližně $\mathcal{O}(n^{1.14})$ 3.4.

| **Datová sada a metrika** | **Empirická složitost** |
|---|---|
| Corel/$l_2$ | $\mathcal{O}(n^{1.11})$ |
| Audio/$l_2$ | $\mathcal{O}(n^{1.14})$ |
| Shape/$l_2$ | $\mathcal{O}(n^{1.11})$ |
| DBLP/cos | $\mathcal{O}(n^{1.11})$ |
| Flickr/EMD | $\mathcal{O}(n^{1.14})$ |

Tabulka 3.4: Empirická složitost *NNDescent* měřena na různých datových sadách [10]. Metrika $l_2$ představuje Minkowského vzdálenost s $p = 2$, tedy Euklidovskou vzdálenost.

# Kapitola 4

# Návrh měření a experimentů

Pro porovnání jednotlivých metod mezi sebou byla navržena implementace měřícího programu *compare* 4.1, který umí vyhledávat nejbližší sousedy pomocí zmíněných aproximačních metod 3 a naivním algoritmem. Data pro experimentování jsou generované datové sady a datové sady z reálných pozorování.

## 4.1 Implementace

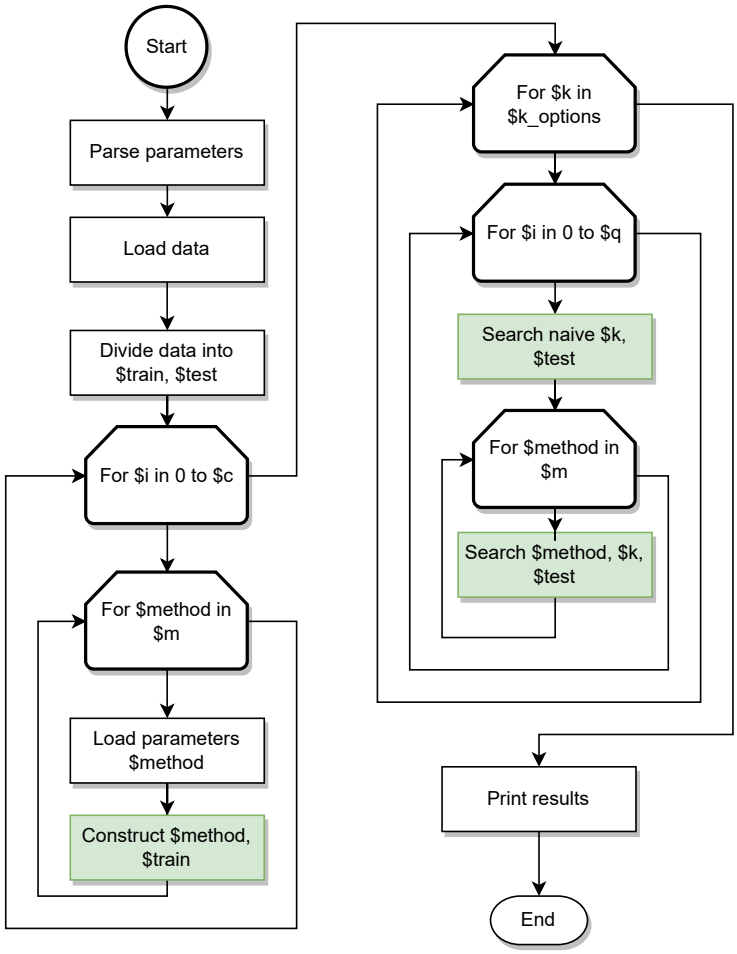
Program `compare` měří prostorovou zátěž algoritmů, čas konstrukce datových struktur aproximačních metod, čas vyhledání KNN a recall s různými nastaveními vlastních parametrů, různým $k$ a různými datovými sadami.

Cílem je porovnat metody mezi sebou a s lineárním algoritmem, jak už z hlediska zmíněných měření, tak jejich schopnosti najít správné NN, tzn. správnosti aproximace. Na každý vyhledávací problém program nejdříve aplikuje naivní algoritmus pro časové referenční porovnání a získání správných NN. Poté spustí vyhledávání požadovaných aproximačních metod.

Rozhraní programu dovoluje nastavit:

- `-i` Vstupní soubor obsahující data.
- `-n` Výpis výsledků naivního algoritmu.
- `-c` Počet opakování konstrukce datových struktur aproximačních metod.
- `-q` Počet vyhledávání.
- `-k` Parametr $k$ pro vyhledávání KNN, možnost zadat více hodnot oddělených čárkou.
- `-m` Názvy aproximačních metod, které se mají použít, odděleny čárkou.
- `-v` Podrobný výpis.

Parametry počtu konstrukcí a počtu vyhledávání v rozhraní programu byly uvedeny proto, aby se výsledky měření neopíraly pouze o jednu naměřenou hodnotu, ale byly průměrem vícero stejných měření. Tímto způsobem se redukují odchylky v měření. Každá z aproximačních metod má konfigurační soubor dovolující nastavit jejich vlastní parametry.




Obrázek 4.1: Zjednodušený diagram programu *compare*. `$c` je nastavený počet konstrukcí, `$k_options` jsou různé hodnoty parametru $k$ algoritmu KNN, `$m` jsou požadované aproximační metody a `$q` je požadovaný počet vyhledávání.

Program `compare` je implementován v jazyce Python. Podporuje vyhledávání pomocí všech zmíněných metod 3. Použité knihovny s implementacemi jednotlivých porovnávaných algoritmů budou představeny v dalších sekcích. Běh programu je u některých experimentů značně dlouhý, proto program na konci vydá zvukové upozornění, aby uživatel věděl, že experimenty proběhly a může započít další. Naivní algoritmus je v `compare` implementován přímočarým způsobem; vypočte se vzdálenost vyhledávaného bodu ke všem z datové sady, výsledek se seřadí a vrátí prvních $k$ prvků ze seznamu 4.1.

```
def euclidean(v1, v2):
    d = ((v1 - v2) ** 2).sum()
    return math.sqrt(d)

def get_knn_naive(points, query_point, k):
    answer = []
    for i in range(points.shape[0]):
        ip = points[i, :]
        x = (euclidean(ip, query_point), i, ip)
        answer.append(x)
    return sorted(answer, key = lambda x: x[0])[:k]
```

Výpis 4.1: Implementace naivního KNN algoritmu v programu `compare`.

## 4.2 Data

Pro účel experimentů poslouží datová sada o Covid-19 pacientech [20] a vlastní generovaná data. Sada o pacientech slouží pro porovnání nad reálnými daty a generované data pro experimenty s různými velikostmi datové sady a různými počty dimenzí.

### 4.2.1 Datová sada o pacientech Covid-19

Datová sada s pacienty čítá 1,048,576 záznamů s 21 dimenzemi:

- pohlaví: 1 pro ženu a 2 pro muže
- věk: věk pacienta
- klasifikace: Výsledky testu na covid. Hodnoty 1-3 znamenají, že u pacienta byl diagnostikován covid v různých stupních. 4 nebo vyšší znamená, že pacient není přenašečem covidu nebo že test je neprůkazný.
- typ pacienta: typ péče, kterou pacient na jednotce obdržel; 1 pro návrat domů a 2 pro hospitalizaci
- zápal plic: zda pacient již má zánět vzduchových vaků nebo ne
- těhotenství: binární atribut, vyjadřující zda je pacientka těhotná
- diabetes: binární atribut, vyjadřující zda má pacient cukrovku
- copd: binární atribut, vyjadřující zda má pacient chronickou obstrukční plicní nemoc
- astma: binární atribut, vyjadřující zda má pacient astma
- inmsupr: binární atribut, vyjadřující zda je pacient imunosuprimovaný (mající oslabený/potlačený imunitní systém)
- hypertenze: binární atribut, vyjadřující zda má pacient hypertenzi
- kardiovaskulární: binární atribut, vyjadřující zda má pacient onemocnění související se srdcem nebo krevními cévami

- renální chronické: binární atribut, vyjadřující zda má pacient chronické onemocnění ledvin
- jiné onemocnění: binární atribut, vyjadřující zda má pacient jiné onemocnění
- obezita: binární atribut, vyjadřující zda je pacient obézní
- tabák: binární atribut, vyjadřující zda je pacient uživatelem tabáku
- usmr: udává, zda pacienta ošetřovala lékařská jednotka první, druhé nebo třetí úrovně
- zdravotnická jednotka: typ instituce národního zdravotního systému, která poskytovala péči
- intubed: binární atribut, vyjadřující zda byl pacient připojen k ventilátoru
- icu: binární atribut, vyjadřující zda byl pacient přijat na jednotku intenzivní péče
- datum úmrtí: Pokud pacient zemřel, datum úmrtí, jinak 9999-99-99

V booleovských datech se pravda označuje 1 a nepravda 2. Chybějící data jsou označena hodnotami 97 a 99. Řádky s chybějícími hodnotami budou odstraněny, protože datová sada je dostatečně veliká na to, aby se experimenty bez těchto řádků daly provést. Datum úmrtí bude převedeno na další booleovskou hodnotu, která bude značit, zda-li pacient zemřel. U žen je těhotenství značeno 1 a 2, u mužů má tento příznak hodnotu 97. ten bude převeden na prostou nepravdu.

### 4.2.2 Generované data

Generovaná data jsou čistě číselné a nekorelované. Jedná se o shluky generované normálním rozložením s různými parametry v každé dimenzi. Generátor pracuje s konfiguračním souborem 4.2 ve formátu YAML, pomocí kterého je generování řízeno. První tři parametry konfiguračního souboru značí počty generovaných bodů, dimenzí a shluků, mezi které se data rovnoměrně rozdělí. Volitelný parametr `seed` definuje semeno pro generování náhodných čísel, jinak je náhodnost generování podmíněna aktuálním časem. Nastavení `default_cluster` definuje hranice pro parametry normálních rozložení, kterými budou shluky generovány v každé dimenzi. Shluky se můžou také definovat ručně a tím přepsat výchozí nastavení. Jednotlivé dimenze ve shluku se také dají ručně nastavit. Nenastavené shluky/dimenze použijí výchozí nastavení. U shluku se dá nastavit parametr `share`, který značí, jaký podíl v datech daný shluk má. Nenakonfigurované shluky si zbytek podílu rovnoměrně rozdělí, součet podílů nesmí přesáhnout hodnotu 1, při nastavení všech shluků musí mít přesnou hodnotu 1. Samozřejmě nesmí být nakonfigurováno více shluků, než je jejich zadefinovaný počet a nesmí být nastaveno více shluků, než počet generovaných bodů.

```
number_of_datapoints: 10000
number_of_dimensions: 20
number_of_clusters: 5
seed: 58454
default_cluster:
    center_min: 0
    center_max: 100
    deviation_min: 1
    deviation_max: 5
clusters:
```

```
–
    share: 0.1
    dimensions:
        – { center: 50.75, deviation: 5.8 }
    center_min: 25
    center_max: 50
    deviation_min: 1
    deviation_max: 20
–
    share: 0.3
    dimensions:
        – { center: 5, deviation: 4 }
        – { center: 7.4, deviation: 12 }
        ...
...
```

Výpis 4.2: Formát konfiguračního souboru pro generátor dat.

Tímto způsobem se dají generovat data jak na míru pro KNN či jednotlivé metody, tak extrémní případy, ve kterých se dají testovat meze algoritmů. Způsob dovoluje libovolné rozmezí mezi kontrolou nad generovanými daty a náhodností.

Výstupem je CSV soubor nerozlišující příslušnost shlukům (jednotlivé body bez příznaku).

## 4.3 Knihovny

Pro každou ze zmíněných KNN metod byla vybrána vhodná implementace v jazyce Python.

### 4.3.1 KD-Strom

Kandidáti pro KD-Strom jsou knihovny *pykdtree*[1], *PythonKD-Tree*[2] a *kdtree*[3]. V experimentech je použita *pykdtree* od autora David Hoese, protože dovoluje nastavit míru aproximace a je udržovaná. Autor zmiňuje, že cílem jeho implementace bylo vytvořit rychlý vyhledávací KNN algoritmus. Doporučuje ho používat v nízko dimenzionálních datech (vyplývá z nátury KD-Strom algoritmu). Práce je inspirována knihovnami *scipy.spatial.cKDTree* a *libANN*, přičemž autor tvrdí, že kombinací vybral z obou knihoven „to nejlepší“. Implementace podporuje multithreading, ten ale nebude použit z důvodu, že by při porovnáních znamenal značnou výhodu oproti ostatním algoritmům, které tuto možnost nepodporují. Při tvorbě stromu je možné nastavit parametr `leafsize`, který značí maximální počet bodů v listových uzlech s výchozí hodnotou `leafsize=16`. Vzdálenost mezi body je počítána čtvercově. Vypočítá se pro každou dimenzi zvlášť, umocní a výsledky se sčítají. Jedná se o něco mezi Manhattanovskou a Euklidovskou vzdáleností.

V každé úrovni stromu algoritmus vybírá dimenzi pro rozdělení prostoru podle toho, která má nejširší záběr v datech, největší rozpětí hodnot.

Implementace při vyhledávání dovoluje nastavit parametr `epsilon`, který ovlivňuje aproximaci; navštívení dříve nenavštívené větve, pokud poloměr kružnice od středu $q$ do $k$-tého vybraného souseda přesahuje rozdělující hyperrovinu. To znamená, že za rozdělující hyperrovinou může být bližší bod. Druhá strana prostoru se navštíví, pokud vzdálenost $d(q, r)$, kde $r$ je $k$-tý nejbližší soused přesáhne vzdálenost $hyp\_dist * (1 + epsilon)$, kde

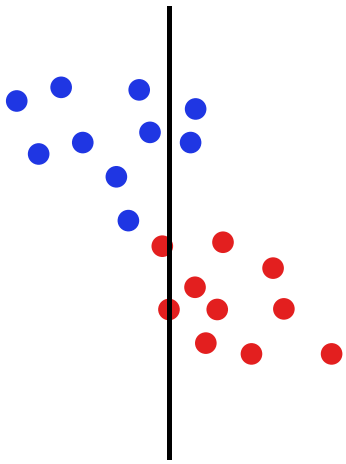
[1] https://github.com/storpipfugl/pykdtree
[2] https://github.com/Vectorized/Python-KD-Tree
[3] https://github.com/stefankoegl/kdtree

*hyp_dist* je vzdálenost od $q$ po hyperrovinu. To znamená, že pokud je parametr `epsilon` nastavený na 0, algoritmus by měl najít všechny nejbližší sousedy se 100% úspěšností (samozřejmě za cenu delšího času vyhledávání). Jinak jsou dostupné parametry `sqr_dists`, který určuje, zda-li mají být navrácené vzdálenosti v Čtvercové nebo Euklidovské vzdálenosti (algoritmus pracuje ve Čtvercových vzdálenostech nehledě na tento parametr) a parametr `distance_upper_bound`, který určuje maximální vzdálenost sousedů a je použit k ořezávání větví. Ani jeden z těchto parametrů není pro porovnávání relevantní; ani `distance_upper_bound`, protože vzdálenost nejvzdálenějšího z $k$ sousedů není předem známá.

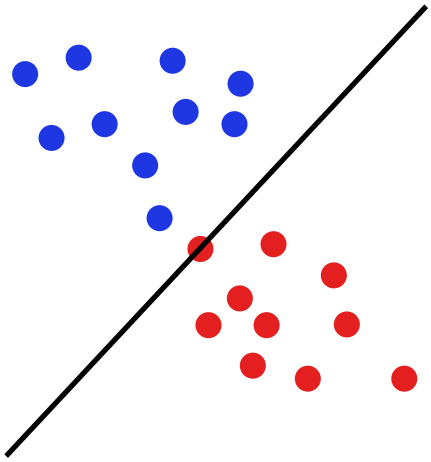
Jedná se o solidní a složitější implementaci, která bere největší ohled na vyhledávací a konstrukční časy. Používá výpočetně jednoduché heuristiky. Na githubu od uživatelů obdržel 170 hvězd k lednu 2023.

### 4.3.2 Kulovitý strom

Pro Kulovitý strom nebyly nalezeny úctyhodné implementace. Buď jsou určené pro klasifikaci[4], nebo pokud ne, tak není využit potenciál kulovitého stromu neuvažovat dimenze při konstrukci stromu 4.3. Je tomu tak v případě autora Juliet Nasar *BallTree*[5] nebo knihovny *sklearn.neighbors.BallTree*[6]. V obou případech není využit ani základní algoritmus nalezení rozdělovací roviny. Rozdělovací rovina je nalezena pomocí mediánu v dimenzi s největším rozpětím hodnot; stejně, jako v případě knihoven KD-Stromu 4.3.1.



Obrázek 4.2: Rozdělovací rovina nad ukázkovými daty nalezená podle dimenze s největším rozpětím hodnot. Shluky nejsou dobře rozděleny.



Obrázek 4.3: Rozdělovací rovina nad ukázkovými daty nalezená pomocí nejvzdálenějších bodů. Rozdělovací rovina je jednoznačně lepší a vytvořené hyperkoule pomocí tohoto rozdělení boudou prostor lépe popisovat.

Pořád je ale přítomna druhá výhoda oproti KD-Stromům - při sestupu se uvažuje vzdálenost od vyhledávaného bodu k centroidům napříč všemi dimenzemi, ne pouze jedné určené pro nějakou z úrovní stromu, jak je tomu v případě KD-Stromu.

[4] https://github.com/JKnighten/k-nearest-neighbors
[5] https://github.com/julietnasar/BallTree
[6] https://github.com/scikit-learn/scikit-learn

Pro experimenty byla vybrána knihovna *sklearn.neighbors.BallTree* kvůli její oblíbenosti. Při konstrukci nabízí parametry `leaf_size` a `metric`, přičemž nabízí širokou škálu použitelných metrik:

- Manhattanovská vzdálenost
- Cosinová vzdálenost
- Euklidovská vzdálenost
- Oblouková vzdálenost (Haversinova/vzdálenost velkého kruhu)
- Minkowskeho vzdálenost
- ...

Použití některé ze vzdáleností ve skutečnosti znamená použití takzvané redukované vzdálenosti. Jedná se o využití jednodušeji spočitatelné metriky, která přibližně zachovává pořadí vzdáleností původní metriky. Pro Euklidovskou vzdálenost je to vynechání posledního matematického úkonu - odmocnění, stejně jako u předchozích knihoven. U některých vzdáleností je skutečná i redukovaná vzdálenost stejná, třeba u Manhattanovské vzdálenosti.

Při vyhledávání je možné nastavit, zda-li má algoritmus vyhledávat do hloubky nebo do šířky. Pokud bod leží vně koule, uzel, který danou kouli reprezentuje je zařazen buď do fronty nebo zásobníku. Pokud bod neleží vně koule, algoritmus tuto kouli neuvažuje a ořeže podstrom. Další nastavení vyhledávání jsou pro experimenty nezajímavé.

### 4.3.3 Locality-Sensitive Hashing

Pro Locality-Sensitive Hashing metodu byla ze dvou kandidátů *lshash*[7] a *lshashing*[8] vybrána knihovna *lshashing* od autora Muhammad Fawi, protože její rozhraní dovoluje nastavit vícero vyhledávacích parametrů. Implementace zahrnuje všechny klíčové aspekty LSH a návratové hodnoty jsou aproximovaní nejbližší sousedé, ne pravděpodobnost nebo nějaký index podobnosti, jako tomu u některých implementací bývá. Při konstrukci je možné zvolit parametr `l` - počet hashovacích tabulek, rozdělení prostoru. Autor uvádí, že podporuje pouze náhodné projekce (generování hashovacích funkcí podle náhodných hyperrovin), to je ale základní, prvotní myšlenka algoritmu a také způsob, který je v této práci rozveden. Kód je kvalitní, podporuje i paralelismus 3.2.4 jak při konstrukci, tak při vyhledávání.

Podporuje pouze Euklidovskou vzdálenost. V kódu je naznačeno, že původně měla knihovna podporovat vícero vzdálenostních funkcí.

Pokud je v regionu, do kterého vyhledávaný bod patří, méně bodů, než je požadované $k$, algoritmus začne prohledávat sousední regiony získané pomocí mutace hashe. Jelikož každý bit v hashi znamená pozici bodu vůči nějaké rovině, je veliká šance, že záměnou libovolného bitu vznikne hash sousedního regionu. Mohou vzniknout také hashe pro regiony, které neexistují 3.2.1. Ty jsou ignorovány. Mutace hashe se ovládá parametrem `radius`, který značí počet bitů, které budou obráceny. Čím větší počet obrácených bitů, tím dále se od původního regionu nový hash bude nacházet (od toho název parametru `radius`).

Druhá možnost pro návštěvu sousedního regionu je parametrem `buckets`, který značí pevný počet regionů, které algoritmus navštíví. Tudíž algoritmus navštěvuje regiony dokud

[7] https://github.com/loretoparisi/lshash
[8] https://github.com/MNoorFawi/lshashing

nedosáhne počet kandidátů $k$ nebo dokud je počet navštívených regionů menší než zadaný počet regionů ($visited < buckets$).

Při samotném sběru kandidátů nedochází k řazení podle vzdálenosti od vyhledávaného bodu. Až jsou obě podmínky navštěvování regionů splněny, kandidáti se seřadí a navrátí se z nich nejbližších $k$.

### 4.3.4 Strom náhodných projekcí

Pro tuto metodu jsou vhodné knihovny *Annoy*[9] (Approximate Nearest Neighbours Oh Yeah) autora Erik Bernhardsson a *rpforest* od firmy Lyst[10]. Vybrána byla *Annoy*, protože je vzhledem k počtu github hvězd 50krát populárnější a je použita ve známé hudební aplikaci Spotify pro doporučení skladeb, kde vyhledává nad obrovským množstvím dat. Pozornost u knihovny byla věnována prostorové složitosti a možnosti sdílení indexů napříč vícero procesy. Modely jsou ukládány na disk a po vytvoření do nich není možno vkládat nová data. Z disku jej pak mohou procesy načíst.

Počet stromů v lese je dám konstrukčním parametrem `n_trees`, nejedná se tedy o jeden strom, ale les. Obecně platí, že čím víc stromů se zkonstruuje, tím je vyšší přesnost vyhledávání. Volitelný parametr při vyhledávání `search_k` umožňuje za běhu upravovat poměr mezi vyhledávacím časem a přesností vyhledání. Vyhledání prozkoumá do `search_k` uzlů stromu, výchozí hodnota je $search\_k = n\_trees * k$.

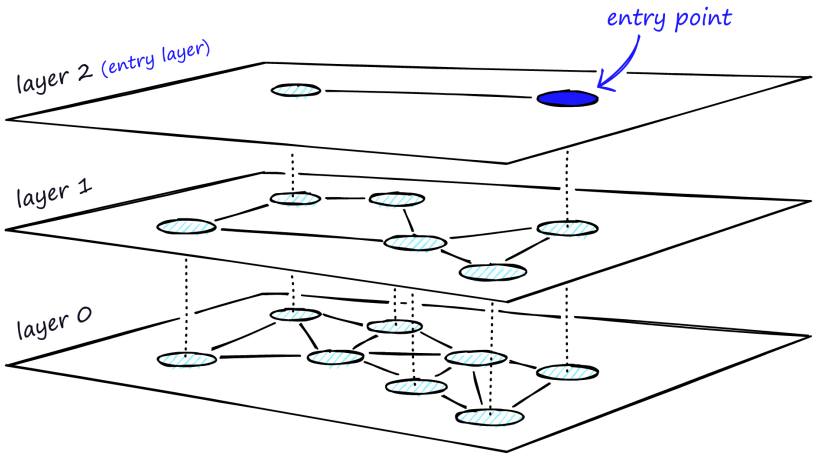
Je možné použít vícero vzdálenostních funkcí:

- Euklidovská vzdálenost
- Manhattanovská vzdálenost
- Cosinová vzdálenost
- Hammingova vzdálenost
- Skalární součin

### 4.3.5 Algoritmy hledání KNN založené na grafu

Rodina grafových algoritmů je zastoupena implementací Hierarchical Navigable Small Worlds (HNSW) *hnswlib*[11]. Používá neorientované hrany a koncept vrstvených grafů 4.4.

[9] https://github.com/spotify/annoy
[10] https://www.lyst.com/
[11] https://github.com/nmslib/hnswlib




Obrázek 4.4: Vrstvy v HNSW. „Hierarchical“ v názvu značí prezenci vrstev. Obrázek byl převzat z https://www.pinecone.io/learn/hnsw/.

V první (horní) vrstvě se nachází nejdelší hrany a uzly, které tyto hrany spojují 4.4. Sestupem se algoritmus dostává do grafové struktury s vícero uzly a kratšími hranami. Tyto hrany zachycují jemnější rozdíly ve vzdálenostech. Algoritmus sestoupí do nižší vrstvy pokud už vyčerpal možnosti vyšší vrstvy a dostal se k nejbližšímu uzlu k vyhledávanému bodu. Analogicky se postup dá přirovnat k vyhledávání ulice na mapě světa, kde nejvyšší vrstvy sestupně obsahují uzly reprezentující kontinenty, státy, města a nakonec ulice. Vrstvení grafů je možné přirovnat vytvoření grafu nejbližších sousedů ze skupiny seedů nad shluky dat 3.5.2. Ze skupiny seedů se jeden náhodně vybere a průchodem grafu se získá nejbližší seed k vyhledávanému bodu. Tento graf seedů a první vrstva v HNSW je v podstatě to samé.

Mezi podporované vzdálenostní funkce patří:

- Čtvercová vzdálenost
- Skalární součin
- Cosinová vzdálenost
- Vlastní vzdálenostní funkce definované uživatelem

Konstrukční parametr `M` značí počet neorientovaných hran jednoho uzlu. Autoři uvádí, že rozumný počet pro `M` je v rozsahu 2–100. Větší `M` funguje dobře pro datové sady s vyšší dimenzionalitou a/nebo když je požadovaná vysoká přesnost, nižší `M` pro datové sady s menší dimenzionalitou a/nebo když vysoká přesnost není požadovaná. Parametr `ef_construction` značí velikost dynamického seznamu kandidátů nejbližších sousedů pro uzel ve fázi konstrukce. Vyšší `ef_construction` vede k lepší aproximaci gragu nejbližších sousedů, zvyšuje ale čas potřebný ke konstrukci. Parametr `num_elements` značí maximální počet uzlů grafu.

V procesu vyhledávání je možné nastavit parametr `ef`, který má stejný význam jako `ef_construction`, jenom je určen pro velikost seznamu kandidátů při vyhledávání, ne konstrukci (vyhledávání i konstrukce využívá stejný algoritmus nacházení sousedů).

Další implementací grafových algoritmů je *PyNNDescent*[12] od autora Leland McInnes insirováno prací od Dong Wei, Charikar Moses a Kai Li. Při konstrukci využívá myšlenky

[12] https://github.com/lmcinnes/pynndescent

„soused mého souseda je pravděpodobně i můj soused“ a používá les Stromů náhodných projekcí k inicializaci konstrukčního procesu. Iterativně vylepšuje graf nejbližších sousedů, přičemž metoda je škálovatelná a dobře funguje i na velkém počtu dat. Na konci konstrukce algoritmus prořeže graf, odstraňuje HUBy a nejdelší hrany v trojúhelnících vzniklých ze tří libovolných uzlů a hranami mezi nimi. Implementace podporuje vysoký počet vzdálenostních funkcí (27) s možností využití uživatelem definované vzdálenostní funkce. Autor se soustředil na vytvoření jednoduchého rozhraní a je možné knihovnu používat bez nastavení jakéhokoliv parametru.

Každopádně, nastavitelných parametrů je také vysoký počet:

- `n_neighbours` počet hran uzlu v grafu
- `n_trees` počet stromů v lese při konstrukci
- `leaf_size` počet bodů v listovém uzlu stromu náhodných projekcí
- `pruning_degree_multiplier` násobitel `n_neighbours`. *pruning_degree_multiplier*∗ *n_neighbours* je maximální počet hran uzlu. Maximální hranicí se omezují HUBy.
- `tree_init` zda-li se má použít les na inicializaci konstrukčního procesu
- `random_state` seed pro generování náhodných čísel
- `low_memory` zda-li použít nižší prostorovou zátěž při konstrukci za cenu vyšší výpočetní zátěže
- `n_iters` maximální počet iterací při konstrukci grafu
- `delta` umožňuje brzké ukončení konstrukčního algoritmu, když je iterační přínos nízký. Vyšší `delta` ukončí konstrukci dříve za cenu nižší kvality grafu.

Jiné parametry jsou buď zastaralé nebo nejsou pro experimenty zajímavé.

# Kapitola 5

# Experimenty

Pro experimenty byl využit porovnávací program `compare`. Všechny experimenty jsou prováděny na procesoru AMD Ryzen 7 5700x. Pro každý závěr bylo provedeno vícero vyhledávání/konstrukcí, ze kterých byl vypočítán průměr. Body pro vyhledávání jsou náhodně získány z datové sady, nad kterou se vyhledávání provádí.

## 5.1 Vlastní parametry aproximačních metod

Vybrané knihovny *pykdtree*, *sklearn.neighbors.BallTree*, *lshashing*, *Annoy*, *hnswlib* a *pynndescent* mají každá konstrukční a vyhledávací parametry, které ovlivňují čas, využitou paměť a recall. V první části této kapitoly budou vysvětleny efekty těchto parametrů.

### 5.1.1 KD-Strom

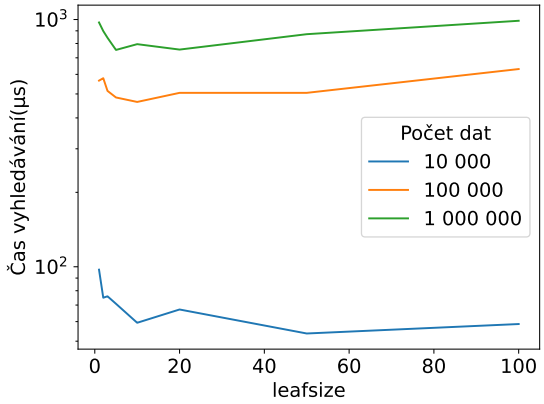
Knihovna *pykdtree* poskytuje parametry `leafsize` a `epsilon`.

Konstrukční parametr `leafsize` ovlivňuje konstrukční čas, využitou paměť a čas vyhledávání. Vyhledávací parametr `epsilon` ovlivňuje čas vyhledávání a recall.

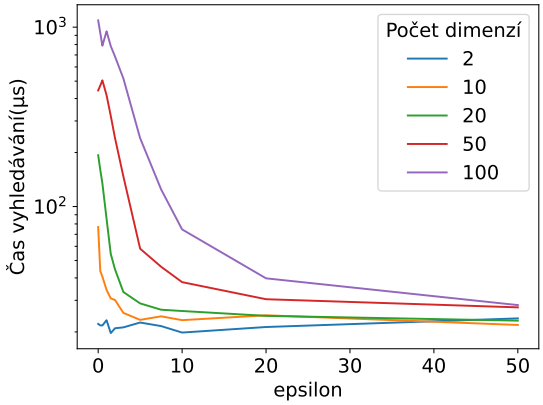
Z experimentů knihovny *pykdtree* se dá vyvodit několik zřejmých faktů. Pokud má strom větší listový uzel (parametr `leafsize`), zmenšuje se nárok na paměťovou zátěž A.2 a zmenšuje se konstrukční čas A.1, přičemž čas vyhledávání nepatrně roste 5.1. V nízkých hodnotách tohoto parametru jsou rozdíly znatelné. Ve vyšších hodnotách, 50 a více, už nepřináší takový užitek v ušetření paměti a času konstrukce (viz. A.2, A.1).

Metoda dobře škáluje s velikostí datové sady a je vidět, že mezi vyhledávacím časem ve 100 000 a 1 000 000 datech není veliký rozdíl 5.1.

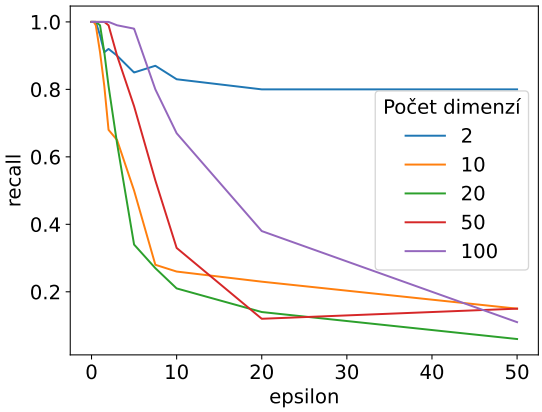
Parametr backtrackingu `epsilon`, pokud je nastavený na 0 způsobí to, že vyhledávání navštíví každý podstrom s potenciálem lepšího výsledku a z KD-Stromu vytvoří metodu s hodnotou recall 1. Parametr v nízko dimenzionálních prostorách nemá veliký vliv ani na čas vyhledávání 5.2, ani na recall 5.3. Bez backtrackingu byl KD-Strom schopen najít nejbližšího souseda v 80 % případů. Ve vyšších dimenzích je ale vidět, jak je metoda bez backtrackingu nepřesná, nedokáže na první pokus průchodu stromem najít nejbližšího souseda. Povšimněme si markantního rozdílu už mezi 2 a 10 dimenzemi. U hodnoty `epsilon` = 0 je v čase vyhledávání rozdíl mezi měřeními v různých počtech dimenzí znatelný, protože s narůstajícím počtem dimenzí narůstá i počet navštívených podstromů. U hodnoty `epsilon` = 50 jsou časy vyhledávání skoro stejné, protože se jedná o jeden průchod stromu v každém měření, rozdíly jsou tvořeny pouze náročnějším výpočtem Euklidovské vzdálenosti.




Obrázek 5.1: Čas vyhledávání *pykdtree* s různými hodnotami parametru `leafsize`.




Obrázek 5.2: Čas vyhledávání *pykdtree* s různými hodnotami parametru `epsilon` a počtem dimenzí.



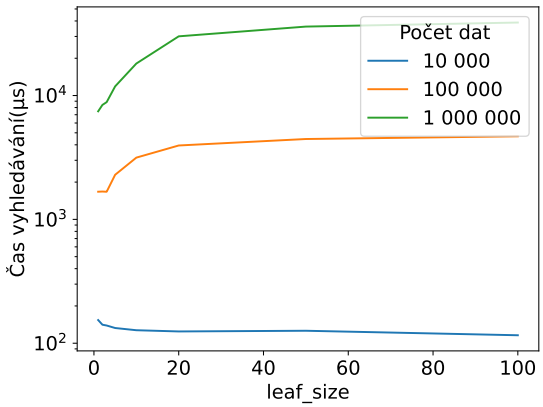

Obrázek 5.3: Recall *pykdtree* s různými hodnotami parametru `epsilon` a počtem dimenzí.

### 5.1.2 Kulovitý strom

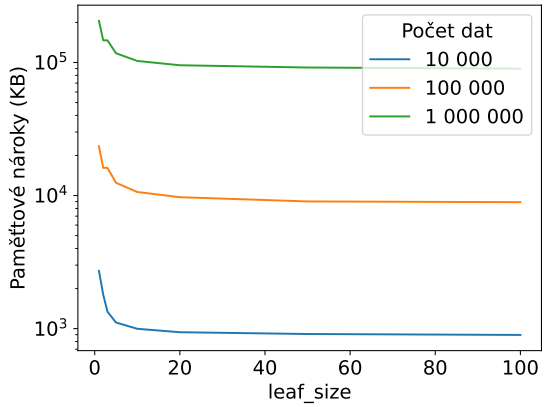
Pro kulovitý strom knihovna *sklearn.neighbors.BallTree* nabízí konstrukční parametr `leaf_size` a vyhledávací parametr `breadth_first`. Naneštěstí nenabízí žádný parametr, který by uživatel mohl ovládat Recall-Queries per second trade-off. To znamená, že algoritmus vždy navštíví větev v podstromu, pokud obsahuje potenciální kandidáty pro lepší sousedy, tudíž recall bude vždy 1.

U knihovny *sklearn.neighbors.BallTree* zastupující Kulovitý strom se nedá měřit recall, protože algoritmus není aproximační a navštíví každý podstrom s potenciálně lepším sousedem. Velikost listového uzlu zde nehraje tak markantní roli (viz. A.3, 5.5, 5.4) jako u KD-Stromu, znovu ale platí, že nastavení je citlivější v nižších hodnotách, snižuje konstrukční čas a paměťovou zátěž a zároveň prodlužuje čas vyhledávání. Ten škáluje s velikostí datové sady poměrně dobře, protože rozdíly v časech vyhledávání mezi měřeními v datových sadách různé velikosti narůstají logaritmicky 5.4. Stejně tak konstrukční čas a paměťové nároky.

Při prohledávání do šířky metoda vykázala zrychlení u datových sad do 5 dimenzí, nebo o 10 a víc dimenzích A.4. Při 50 dimenzích už ale rozdíl ve zrychlení byl pouze kolem 3 %.




Obrázek 5.4: Čas vyhledávání *sklearn.neighbors.BallTree* s různými hodnotami parametru `leaf_size`.




Obrázek 5.5: Prostorová zátěž *sklearn.neighbors.BallTree* s různými hodnotami parametru `leaf_size`.

### 5.1.3 Locality-Sensitive Hashing

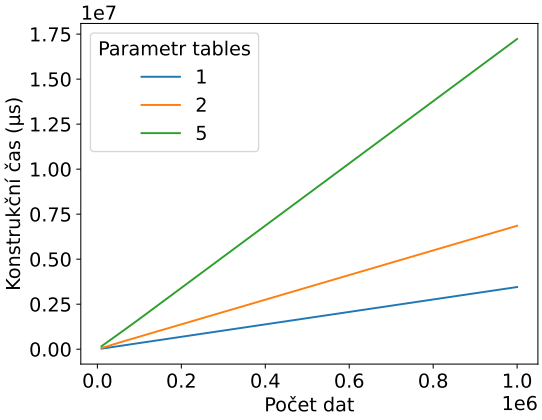
Knihovna *lshashing* nabízí dva konstrukční parametry. Parametr `tables` (počet hashovacích tabulek) a parametr `hash_length` (počet rozdělovacích hyperrovin). Pro vyhledávání existují parametry `radius` (mutace hashe, počet obrácených bitů) a `buckets` (počet regionů, které se mají navštívit). Oboje ovládají Recall-Queries per second trade-off.

LSH se v konstrukčních metrikách ukázala jako velice stabilní metoda s předpověditelnými výsledky. Parametr `tables`, předvídatelně, ovlivňoval konstrukční čas, paměťovou zátěž a čas vyhledávání lineárně (viz. 5.6 A.5 A.6). Samotné konstrukční časy nejsou ovlivněny počtem dimenzí, pouze velikostí datové sady, jak je vidět na grafech 5.7 a 5.6. Paměťová zátěž také není ovlivněna počtem dimenzí 5.8, protože v hashovacích tabulkách jsou uloženy pouze indexy do pole dat, nad kterým byly tabulky zkonstruovány. Recall je ale parametrem `tables` značně ovlivněn 5.9 a už při nízkých hodnotách tohoto parametru metoda aproximuje poměrně dobře.

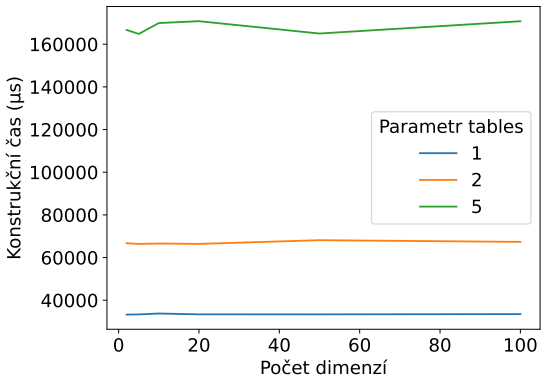
Obdobně jako `tables`, parametr `hash_length` ovlivňuje konstrukční čas lineárně A.7. Prostorová zátěž jím ve vztahu k narůstajícímu počtu dat není nijak ovlivněna A.8, protože se indexy do pole dat jednoduše přerozdělí do vícero skupin v tabulce, jejich počet ale zůstává stejný. Co se ale ve vztahu k narůstajícímu počtu dimenzí týče, prostorová zátěž se lineárně zvětšuje (v malém měřítku, při nastavení `hash_length` = 1 byla paměťová zátěž *lshashing* nad 100 dimenzionálním prostorem 366KB, při nastavení `hash_length` = 20 byla zátěž 390KB) A.10. Čas vyhledávání a recall prudce klesají (viz. 5.10, 5.11). Je to způsobeno tím, že čím je větší `hash_length`, tím je menší průměrný počet dat v regionu a metoda v něm má menší šanci najít nejbližšího souseda.

Metoda dokáže prohledávat vícero regionů pomocí parametru `buckets`. Jeho zvýšením se pravděpodobnost nalezení regionu s nejbližším sousedem zvětšuje. Čas vyhledávání zvětšováním `buckets` roste lineárně 5.12, recall je ním zlepšen už při nízkých hodnotách 5.13.

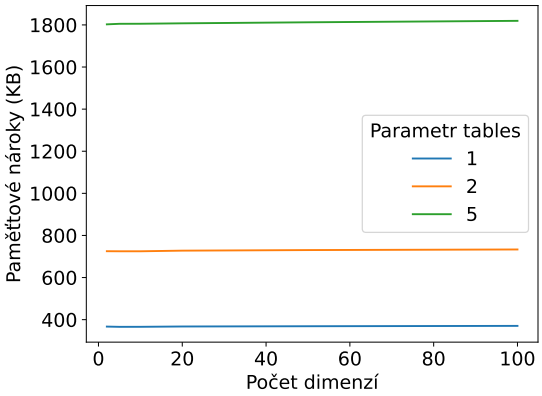
Parametr `radius` dodává variaci vybraným regionům k prohledání a ovlivňuje vzdálenosti mezi nimi. Nevypadá to ale, že by měl nějaký zásadní vliv na čas vyhledávání A.11, protože metoda prohledá stejný počet regionů. Stejně to nevypadá, že by měl zásadní vliv na recall A.12.



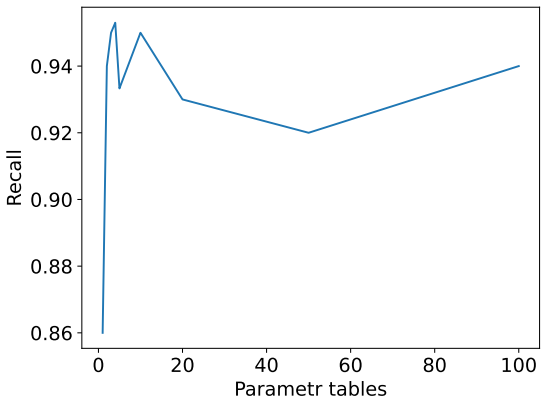

Obrázek 5.6: Konstrukční čas *lshashing* s různými hodnotami parametru `tables` a počtem dat.



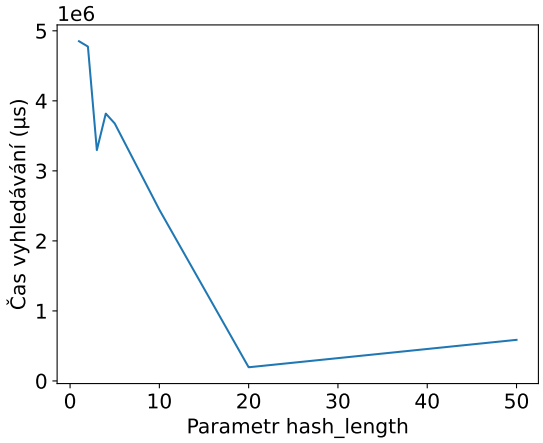

Obrázek 5.7: Konstrukční čas *lshashing* s různými hodnotami parametru `tables` a počtem dimenzí prostoru.



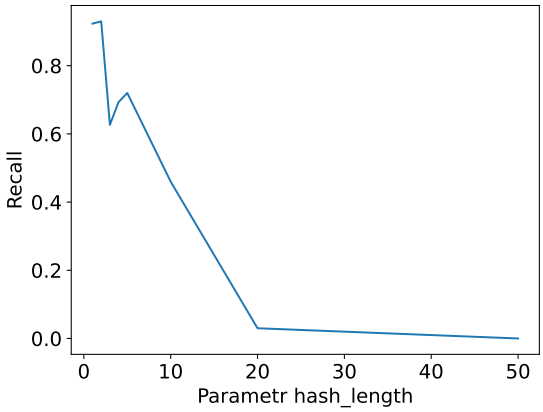

Obrázek 5.8: Prostorová zátěž *lshashing* s různými hodnotami parametru `tables` a počtem dimenzí prostoru. Povšimněme si, že prostorovou zátěž počet dimenzí nijak neovlivňuje.



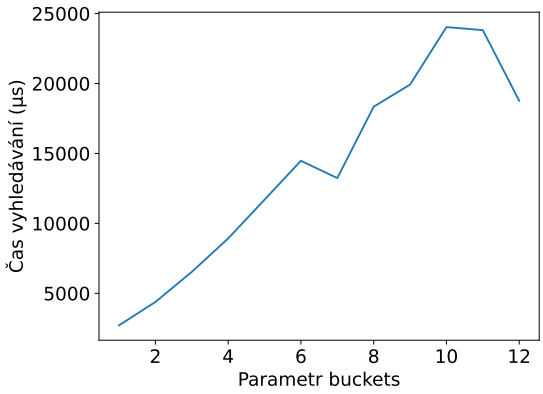

Obrázek 5.9: Recall *lshashing* s různými hodnotami parametru `tables`.



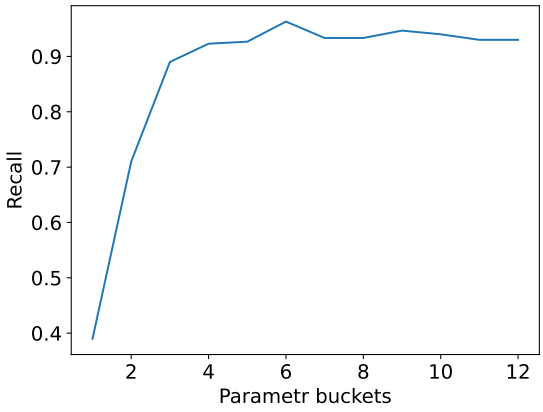

Obrázek 5.10: Čas vyhledávání *lshashing* s různými hodnotami parametru `hash_length`.




Obrázek 5.11: Recall *lshashing* s různými hodnotami parametru `hash_length`.




Obrázek 5.12: Čas vyhledávání *lshashing* s různými hodnotami parametru `buckets`.




Obrázek 5.13: Recall *lshashing* s různými hodnotami parametru `buckets`.

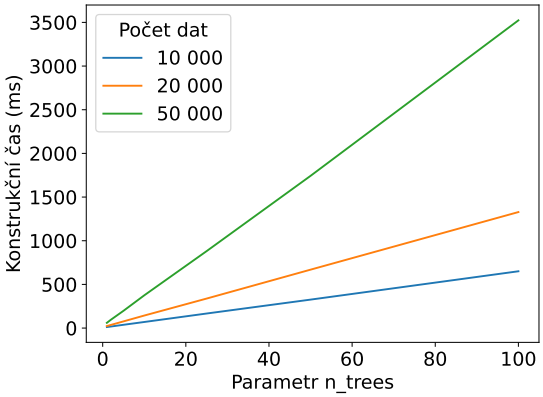
### 5.1.4 Strom náhodných projekcí

V knihovně *Annoy* je možné nastavit pouze dva parametry. Konstrukční parametr `n_trees` určující počet stromů v lese ovlivňuje konstrukční čas, paměťovou zátěž, vyhledávací čas a recall. Parametr vyhledávání `search_k` ovlivňuje jak čas vyhledávání, tak recall.

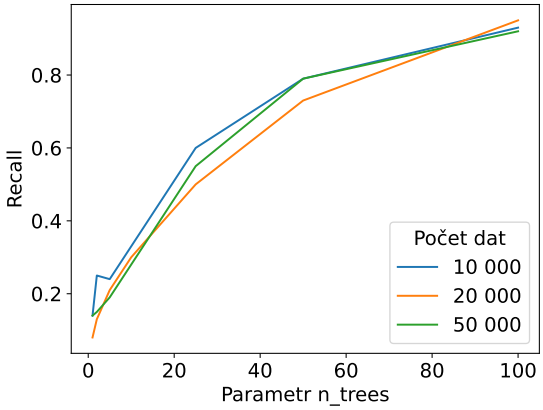
V knihovně *Annoy* se parametr `n_trees` chová obdobně, jako parametr `tables` knihovny *lshashing*. Oboje znamenají v podstatě totéž; `n_trees` znamená velikost lesa stromů náhodných projekcí a `tables` počet hashovacích tabulek. Konstrukční čas, čas vyhledávání i paměťové nároky rostou lineárně (viz. 5.14, A.13, A.14). Recall se zlepšuje a dosahuje maxima při nastavení přibližně `n_trees` = 100 pro datové sady o různých velikostech 5.15. Jiné je to s počtem dimenzí. Pro větší počet je nutné nastavit parametr `n_trees` na větší hodnotu 5.16. Dimenze ale nijak neovlivňují konstrukční čas, paměťové nároky nebo čas vyhledávání (viz. A.15, A.16, A.17).

Metoda je citlivá v nastavení vyhledávacího parametru `search_k`, hlavně co se týče výsledného recallu. Je tomu tak u datových sad s různými velikostmi 5.17 i u datových

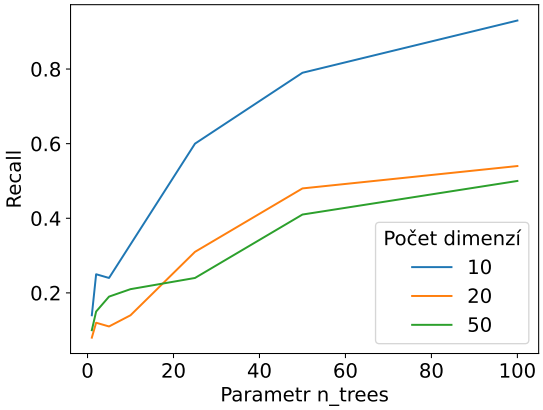
sad s různým počtem dimenzí 5.18. Z výsledků vyplývá, že metoda velmi dobře škáluje s počtem dat a dá se také nastavit pro vyhledávání ve vysoko dimenzionálních prostorách za přijatelného zpomalení vyhledávacího času. To roste s vyššími nastaveními parametrů lineárně.



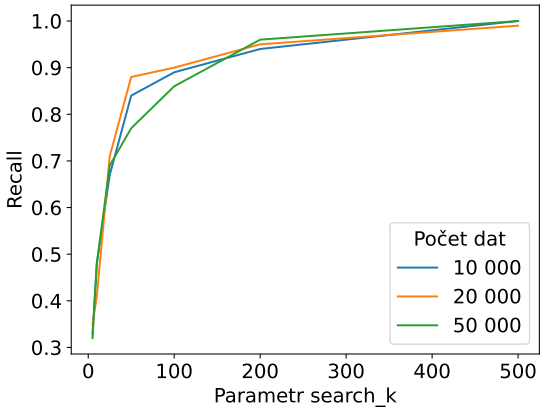

Obrázek 5.14: Konstrukční čas *Annoy* s různými hodnotami parametru `n_trees` a počtem dat.



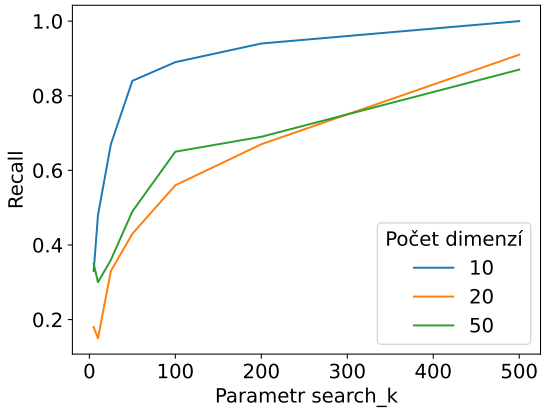

Obrázek 5.15: Recall *Annoy* s různými hodnotami parametru `n_trees` a počtem dat.




Obrázek 5.16: Recall *Annoy* s různými hodnotami parametru `n_trees` a počtem dimenzí.




Obrázek 5.17: Recall *Annoy* s různými hodnotami parametru `search_k` a počtem dat.




Obrázek 5.18: Recall *Annoy* s různými hodnotami parametru `search_k` a počtem dimenzí.

### 5.1.5 Algoritmy hledání KNN založené na grafu

V algoritmu knihovny *hnswlib* se dají nastavit konstrukční parametry `M` a `ef_construction`. `M` ovlivňuje paměťovou zátěž, čas konstrukce, recall a čas vyhledávání. Vyšší parametr `ef_construction` zlepšuje kvalitu grafu za cenu konstrukčního času. Pro měření kvality grafu je použit recall vyhledávání nejbližších sousedů s $ef = ef\_construction$.

*PyNNDescent* dovoluje nastavit pouze konstrukční parametry. Je možné měřit, jak parametr `n_neighbours` ovlivňuje konstrukční čas, paměťovou zátěž, čas vyhledávání a recall, jak `n_iters` a `delta` ovlivňují konstrukční čas a recall, jak `low_memory` ovlivňuje konstrukční čas a `pruning_degree_multiplier` ovlivňuje čas vyhledávání a recall.

Knihovna *hnswlib* se osvědčila jako stálá, rychlá a velice přesná v aproximacích. Parametr `M` lineárně zvyšuje konstrukční čas a paměťové nároky (viz. A.20, A.21). Počet dimenzí nijak neovlivňuje paměťové nároky A.23, ty ovlivňuje pouze počet dat. Čas vyhledávání se s rostoucím `M` a počtem dat nijak zásadně nezvyšuje 5.19, počet dimenzí ale čas vyhledávání s vysokým `M` značně zvýšil 5.21. Zajímavé je to, jak je recall citlivý na nastavení `M`. Už při nízkých hodnotách `M` sledujeme dobrou aproximaci 5.20, kterou neovlivňuje počet dat, zvlášť pro nízko dimenzionální prostory 5.22.

Parametr `ef_construction` zlepšující kvalitu grafu ovlivňuje konstrukční čas lineárně A.24. Čím kvalitnější graf, tím vyšší recall při vyhledávání 5.23. V nízkých hodnotách má `ef_construction` radikální vliv na recall. Ve vyšších se kvalita grafu zlepšuje minimálně, tudíž i vliv na recall je menší.

Vyhledávací parametr `ef` lineárně ovlivňuje čas vyhledávání datových sad s různými velikostmi i datových sad s různým počtem dimenzí (viz. A.25, A.26). Stejně jako u `M`, i na `ef` je recall velice citlivý 5.25, obzvlášť v nízko dimenzionálních prostorách 5.24. Metoda se zvyšováním `M`, `ef_construction` a `ef` dá nastavit na datové sady různých velikostí a počtu dimenzí.

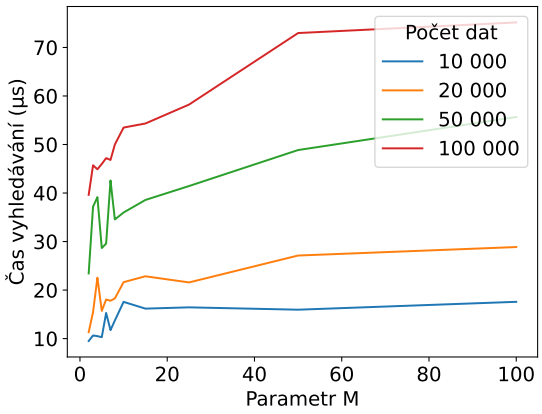
Parametr `n_iters` knihovny *PyNNDescent* ovlivňuje konstrukční čas pouze do určitého momentu A.27, kdy už se konstrukce zastaví z důvodu malého přínosu pro kvalitu grafu. Je ale vidět, že konstrukční čas *PyNNDescent* závisí pouze na počtu dat A.27 (a počtu hran 5.27), nikoliv na počtu dimenzí A.28. Recall metody, stejně jako u ostatních, se snižuje s narůstajícím počtem dimenzí a počet dat na něj nemá moc veliký vliv (viz. 5.26, A.29). Nejlepší dosažený recall byl pouze 0.7 5.29, což je nejméně ze všech měřených metod.

Platí, že čím více dimenzí prostor má, tím více iterací metoda potřebuje pro vytvoření kvalitního grafu. Pro vytvoření grafu nad 10 dimenzionálním prostorem metoda potřebovala do 10 iterací, s vícero dimenzemi pokračovala v konstrukci, i když s malými kvalitativními přírůstky A.28.

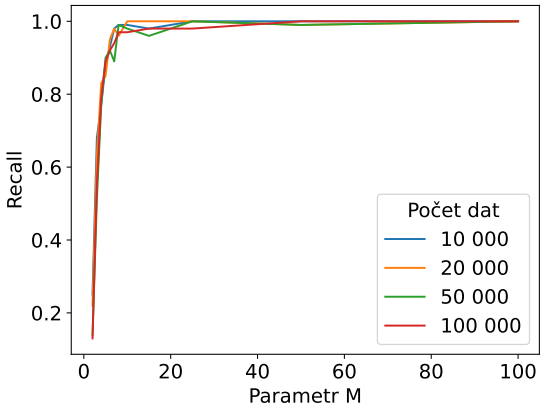
Pro parametr `n_neighbors`, počet hran, platí, že ovlivňuje konstrukční čas lineárně 5.27 a časové nároky na konstrukci hrany se zvětšují pouze počtem dat, ne počtem dimenzí A.38. Paměťové nároky s počtem hran rostou, dosahují ale stropu (viz. A.36, A.39), který formují prořezávací postkonstrukční techniky. Čas vyhledávání (i když s možnými výkyvy) `n_neighbors` ovlivňuje pouze v nízkých hodnotách, posléze je stálý A.37, stejně jako recall 5.28.

Parametr `delta`, umožňující brzké ukončení konstrukce za cenu méně kvalitnějšího grafu, snižuje konstrukční čas pouze minimálně A.30 A.32. Recall nevypadá být ním zvlášť ovlivněn (viz. A.31, A.33), tudíž se zdá, že parametr slouží pro jemné doladění poměru mezi konstrukčním a vyhledávacím časem.

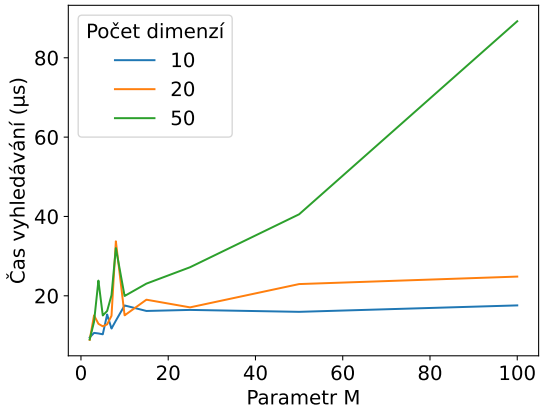
Nastavitelná prořezávací technika pro odstranění HUBů pomocí parametru `pruning_degree_multiplier` vyšším nastavením zvyšuje vyhledávací čas A.34, hlavně u vysoko dimenzionálních dat, kde se HUBy často vyskytují. V nízkých hodnotách pro tyto prostory značně zrychlí vyhledávání, moc nízkým nastavením ale metoda prořezává užitečné hrany a zmenšuje se výsledný recall A.35. Povšimněme si, že v prostorách s mnoha dimenzemi je prořezávání agresivnější 5.30, protože je zde větší šance výskytu uzlů, které mají víc než `n_neighbors` $*$ `pruning_degree_multiplier` hran.



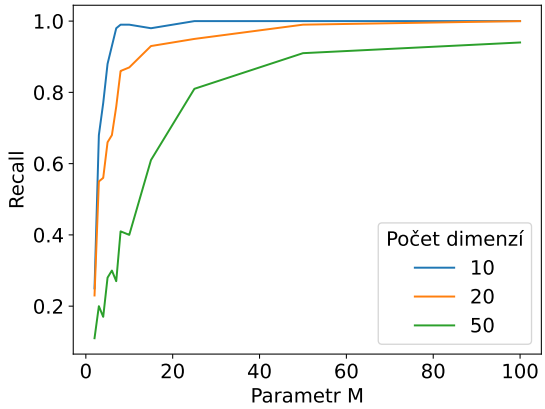

Obrázek 5.19: Čas vyhledávání *hnswlib* s různými hodnotami parametru `M` a počtem dat.



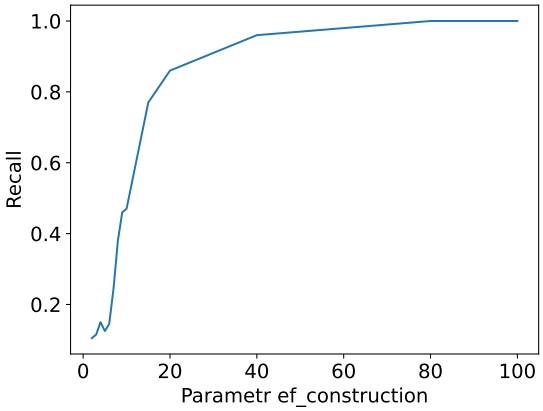

Obrázek 5.20: Recall *hnswlib* s různými hodnotami parametru `M` a počtem dat.



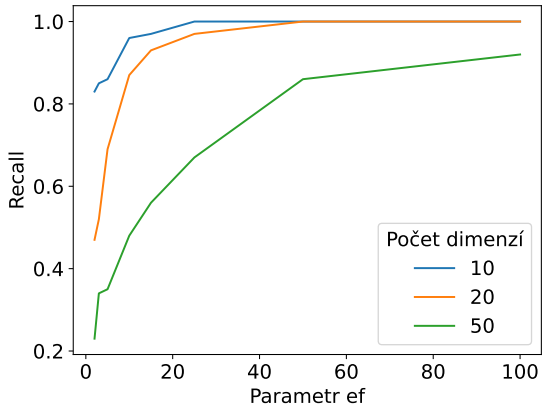

Obrázek 5.21: Čas vyhledávání *hnswlib* s různými hodnotami parametru `M` a počtem dimenzí.



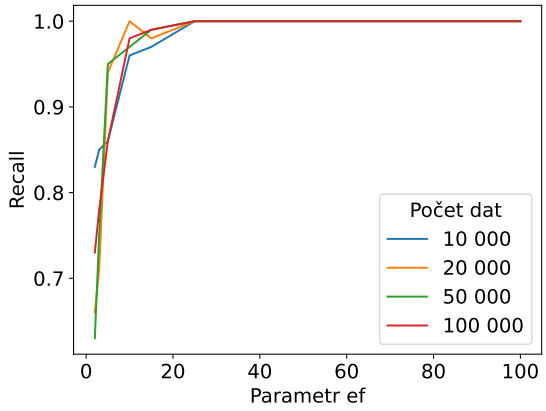

Obrázek 5.22: Recall *hnswlib* s různými hodnotami parametru `M` a počtem dimenzí.



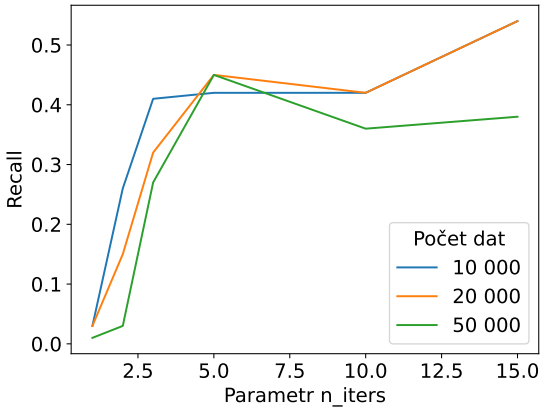

Obrázek 5.23: Recall *hnswlib* s různými hodnotami parametru `ef_construction`.



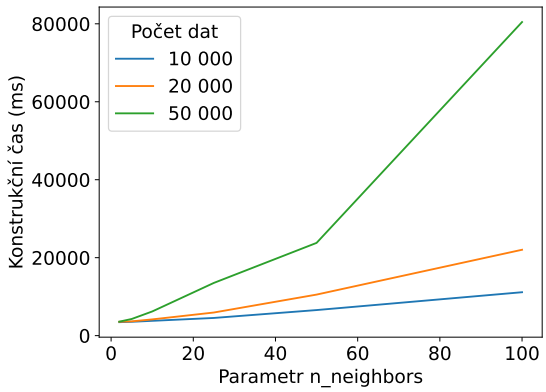

Obrázek 5.24: Recall *hnswlib* s různými hodnotami parametru `ef` a počtem dimenzí.



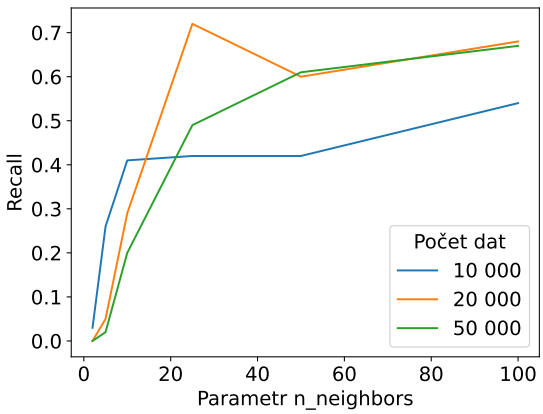

Obrázek 5.25: Recall *hnswlib* s různými hodnotami parametru `ef` a počtem dat.



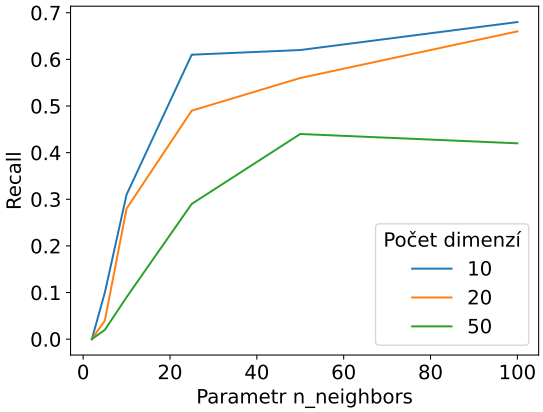

Obrázek 5.26: Recall *PyNNDescent* s různými hodnotami parametru `n_iters` a počtem dat.



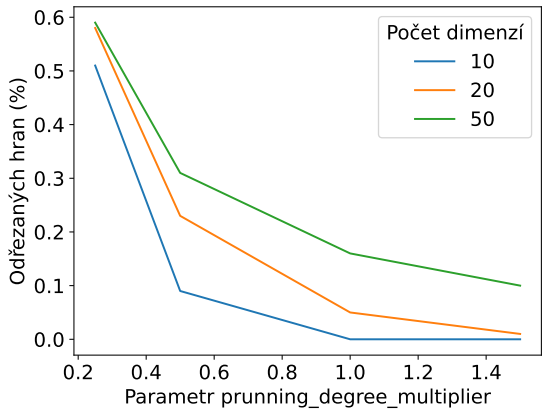

Obrázek 5.27: Konstrukční čas *PyNNDescent* s různými hodnotami parametru `n_neighbors` a počtem dat.




Obrázek 5.28: Recall *PyNNDescent* s různými hodnotami parametru `n_neighbors` a počtem dat.




Obrázek 5.29: Recall *PyNNDescent* s různými hodnotami parametru `n_neighbors` a počtem dimenzí.




Obrázek 5.30: Počet odstraněných hran grafu *PyNNDescent* s různými hodnotami parametru `pruning_degree_multiplier` a počtem dimenzí.

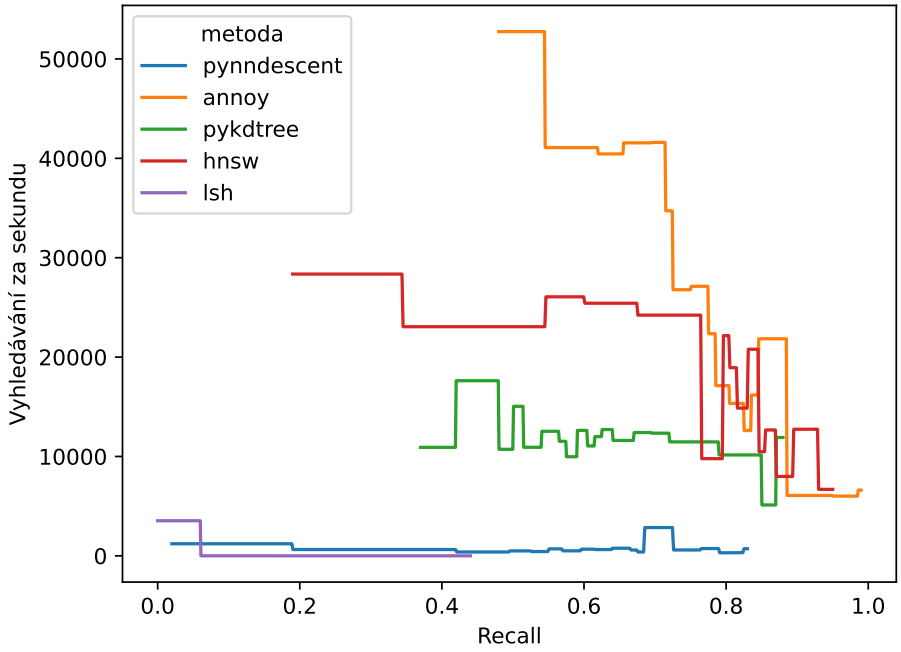
## 5.2 Společné měření

Metriky Recall-Queries per second trade-off a Recall@K jsou důležitými ukazateli aproximačních metod, které ukazují, jak je metoda výkonná při různých nastaveních vlastních parametrů a parametru vyhledávání `K`. Metody byly testovány na datových sadách:

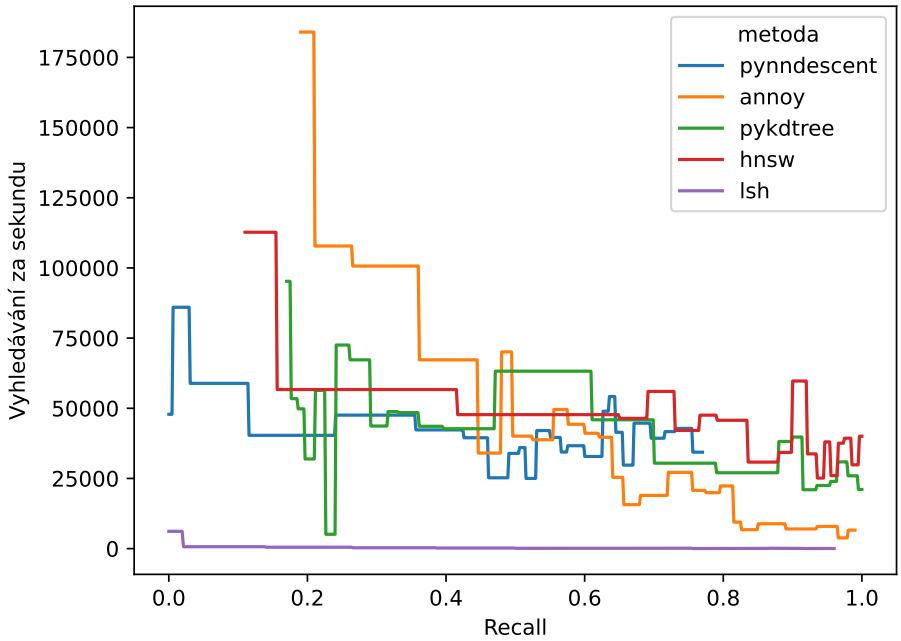
- o Covid-19 pacientech
- vygenerovaná sada jednoho shluku, 10 000 prvků, 10 dimenzí
- vygenerovaná sada 15 shluků, 100 000 prvků, 20 dimenzí

Metoda kulovitého stromu nebyla testována, protože postrádá nastavení mezi aproximací a vyhledávacím časem.

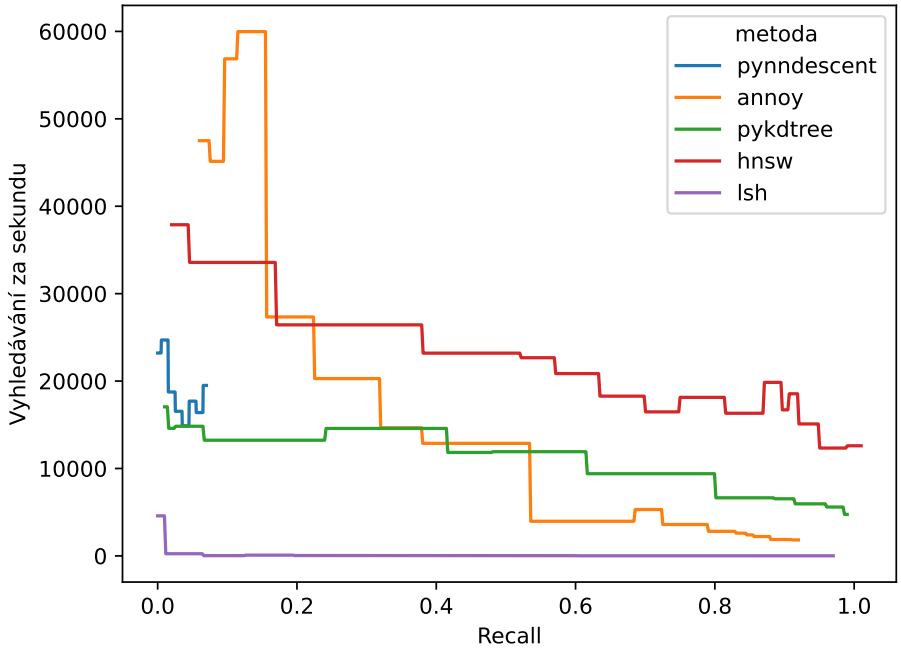
Pro měření metriky Recall@K byly nastaveny parametry metod tak, aby na hodnotě $K = 1$ byla hodnota recall přibližně 0.5.



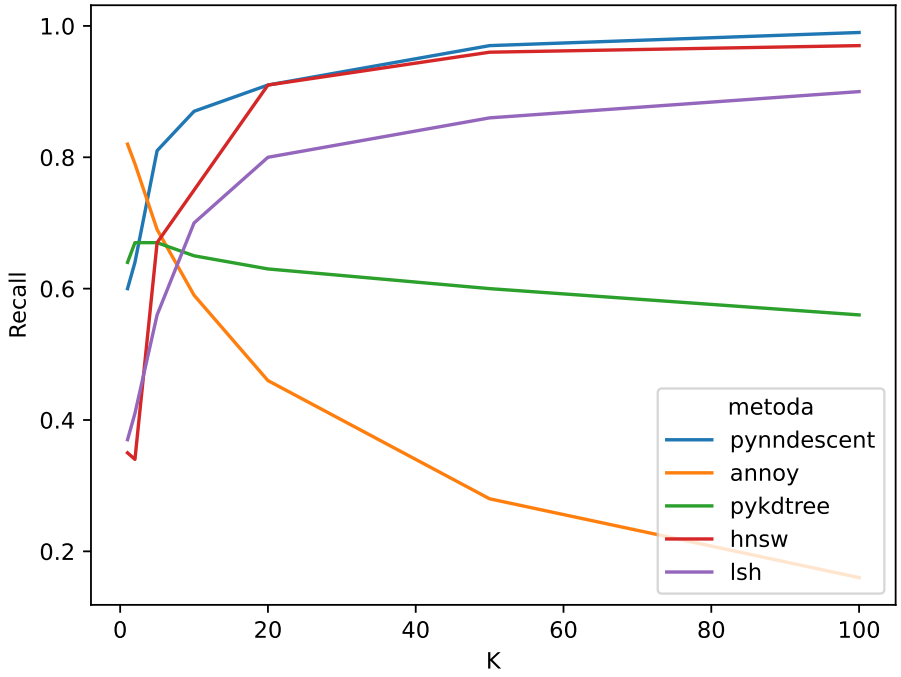

Obrázek 5.31: Metrika Recall-Queries per second trade-off měřena na sadě o Covid-19 pacientech. Průměrný vyhledávací čas naivní metody je 2.05s. Průměrný vyhledávací čas *sklearn.neighbors.BallTree* je 240 694ns, což odpovídá 4 154 vyhledávání za sekundu.



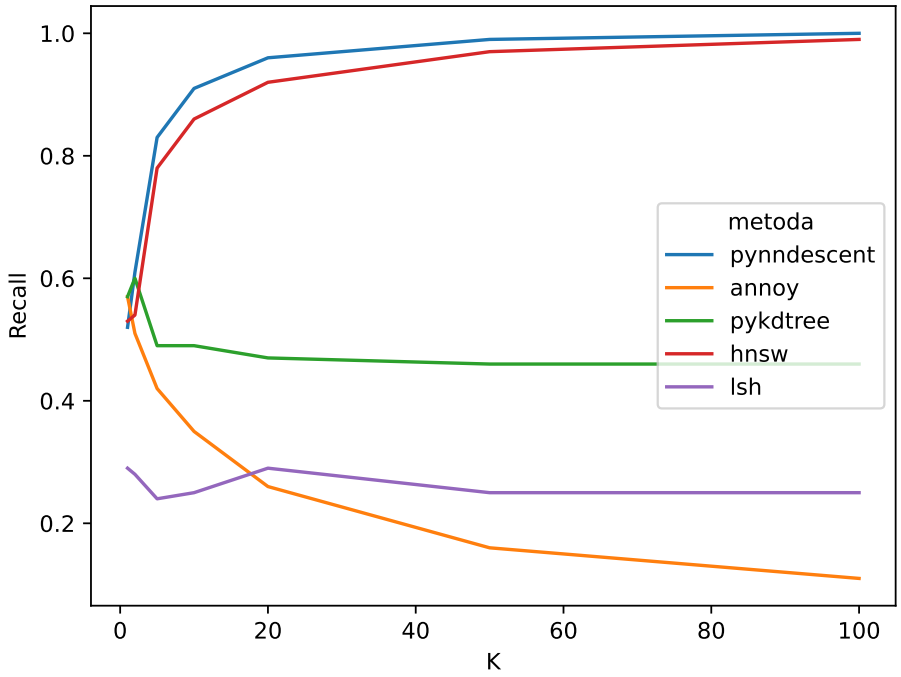

Obrázek 5.32: Metrika Recall-Queries per second trade-off měřena na sadě jednoho shluku, 10 000 prvků a 10 dimenzí. Průměrný vyhledávací čas naivní metody je 17.5ms. Průměrný vyhledávací čas *sklearn.neighbors.BallTree* je 281 534ns, což odpovídá 3 551 vyhledávání za sekundu.



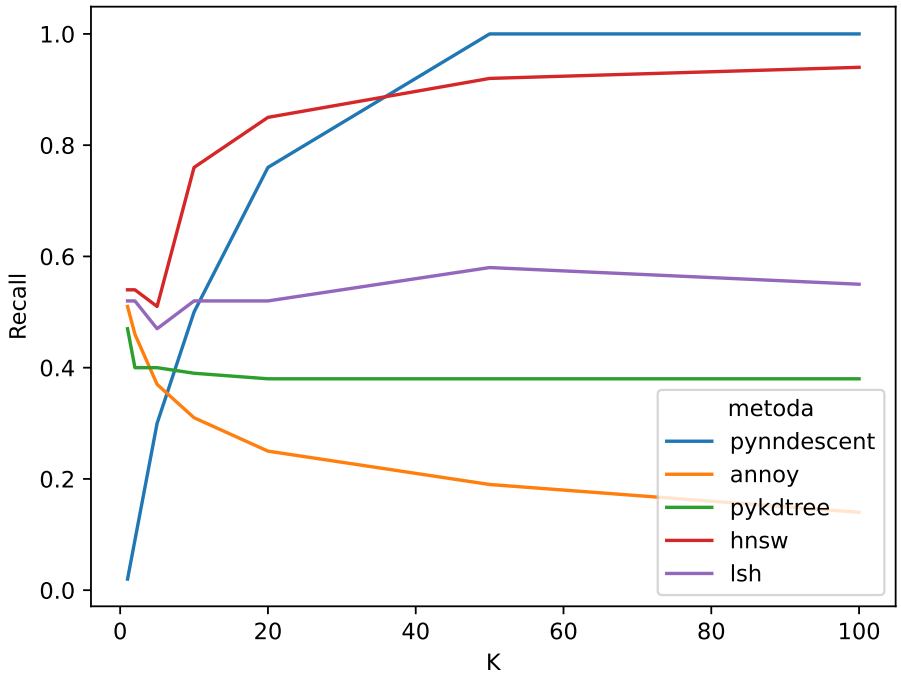

Obrázek 5.33: Metrika Recall-Queries per second trade-off měřena na sadě 15 shluků, 100 000 prvků a 20 dimenzí. Průměrný vyhledávací čas naivní metody je 186ms. Průměrný vyhledávací čas *sklearn.neighbors.BallTree* je 1 046 361ns, což odpovídá 955 vyhledávání za sekundu.




Obrázek 5.34: Metrika Recall@K měřena na sadě o Covid-19 pacientech.




Obrázek 5.35: Metrika Recall@K měřena na sadě jednoho shluku, 10 000 prvků a 10 dimenzí.




Obrázek 5.36: Metrika Recall@K měřena na sadě 15 shluků, 100 000 prvků a 20 dimenzí.

### 5.2.1 Vyhodnocení

Ze společných měření vyplývá, že metoda *Annoy* jednoznačně vede v případech, kdy není zapotřebí dobré aproximace, ale záleží na času vyhledávání (viz. 5.31, 5.32, 5.33). Při testování na datové sadě o pacientech onemocnění Covid-19 dosahuje proti naivnímu algoritmu 100000násobné zrychlení při recallu 0.5. U sady s 10 000 prvky a 10 dimenzemi dosahuje přibližně 1300násobné zrychlení a u poslední sady 2800násobné zrychlení při stejném recallu 0.5. Ve vyšších hodnotách recall jí předběhnou knihovny *hnswlib* a *pykdtree* v každé datové sadě. Pokles počtu vyhledávání za sekundu při vyšších nastaveních je totiž větší u *Annoy*, než u ostatních knihoven.

Pokles počtu vyhledávání vůči lepší aproximaci metody *hnswlib* není tak razantní. Nastavení lepších parametrů vysoce ovlivňuje recall, počet vyhledání za sekundu ale klesá mnohem pomaleji. Celkově se jedná o metodu se stabilnějším vyhledávacím časem pro různé nastavení. Pro recall přibližně 1 vykazuje nejvyšší počet vyhledání za sekundu se zrychlením vůči naivnímu algoritmu 20 000 5.31, 800 5.32 a 2800 5.33.

*PyNNDescent* nevykázal schopnost dosáhnout recallu 1 ani při extrémních nastaveních u žádné z datových sad při počtu hledaných sousedů $k = 1$ 5.31. V tomto jsou si opakem s knihovnou *Annoy*. Která při nízkých počtech $k$ pracuje velmi dobře, u vyšších ale výkonnost klesá. Čas vyhledávání je ale vysoce stabilní při jakýchkoliv nastaveních, zvlášť v datové sadě o pacientech. Počet vyhledávání za sekundu je ale v této datové sadě malý pouze kolem 700 vyhledání za sekundu.

Knihovna *pykdtree* měla nejstabilnější vyhledávací čas v generovaných datových sadách (viz. 5.32, 5.33). Konkuruje grafovým algoritmům ve vyšších dimenzích, i když se jedná o metodu vhodnou pouze pro nízko dimenzionální prostory. V sadě o pacientech sdílí podobné vyhledávací časy s *Annoy* a *hnswlib* 5.31, v generovaných sadách je vyhledávací čas (při recallu přibližně 1) poloviční vůči hnswlib.

Zajímavý je výsledek knihovny *lshashing*. I když s nejmenším počtem vyhledání za sekundu, žádné nastavení tento čas nijak radikálně neovlivňuje. V datové sadě o pacientech metoda nebyla schopna dosáhnout lepší hodnoty recall, než 0.44 5.31.

Při měření Recall@K bylo zjištěno, že nehledě na datovou sadu, grafové algoritmy recall vyšší hodnotou $k$ zlepšují (viz. 5.34, 5.35, 5.36). Naopak recall metody Stromu náhodných projekcí se zhoršuje. Zde je vidět, že *Annoy* a *PyNNDescent* jsou si ve výkonnosti vzhledem k hodnotě $k$ opačné. *PyNNDescent* v generované datové sadě o 100 000 prvků a hodnotě $k = 1$ nedokázal při jakýchkoliv nastaveních dosáhnout hodnoty recall vyšší než 0.1 5.33, *Annoy* excelovala v počtu vyhledávání za sekundu. Když se jednalo o vyhledávání počtu sousedů $k = 100$, *PyNNDescent* dosahuje nejlepší recall ze všech metod, *Annoy* nejhorší 5.36.

Recall *pykdtree* se s rostoucím $k$ pomalu zhoršuje, je ale poměrně stálý.

Knihovna *lshashing* svoji hodnotu recall nijak nemění, pouze v případě datové sady o pacientech 5.34. Je vidět, že při $k = 1$ nedokázala dosáhnout vyšší recall, než 0.44 5.31, pokud je ale hodnota $k$ vyšší, recall se zlepšuje také. Tato anomálie nastává, protože v dané datové sadě se nachází velké množství duplicit. Ze stejného důvodu *pykdtree* v této datové sadě jako jediné nedokázal dosáhnout recall 1 i když nastavením parametru `epsilon` = 0 se aproximace vylučuje 5.31.

Z pozorování a teorie vyplývá, že metody lze vybírat podle datové sady, nad kterou bude prováděno vyhledávání. Co se paměťové zátěže týče, každá metoda škáluje s počtem dat lineárně. Nicméně, největší paměťovou zátěž vykazují grafové algoritmy kvůli nezanedbatelnému počtu hran. Metody lesa Stromů náhodných projekcí a LSH mohou teoreticky

zabírat veliké množství paměti, protože jako jediné škálují do šířky. Nicméně, pro dobré výsledky nepožadují mnoho stromů nebo tabulek. Pokud je uživatel omezen pamětí, měl by se vyhnout grafovým metodám, které kvůli hranám zabírají nejvíce paměti. Řebříčku vede LSH 5.1, protože každá nová tabulka obsahuje pouze indexy do pole dat. Každou novou tabulkou vzrůstá paměťová zátěž jako kdyby měly data o dimenzi navíc.

Pokud uživateli hrozí výpadky systému a potřebuje rychle sestavit vyhledávací struktury, nedoporučuje se použít *PyNNDescent*, protože má nejhorší konstrukční časy. Druhá knihovna založena na grafu KNN, *hnswlib*, je na tom o něco lépe. Knihovna *Annoy* podporuje navíc uložení indexu do vnější paměti, ze které se dá po výpadku přečíst.

Pokud má datová sada mnoho dimenzí, určitě se nedoporučuje použít KD-Strom. Nemá totiž schopnost přizpůsobit se vnitřní dimenzionalitě sady a její přístup k nalezení lepších sousedů vede k prohledání celé datové sady 3.1.2. Ostatní metody s vysokou dimenzionalitou nemají problém. *PyNNDescent* obsahuje techniku ořezávání větví, která pomáhá redukovat vyhledávací čas v takovýchto sadách 4.3.5.

Vysoké hodnoty recall jsou schopny všechny zmíněné metody. Pokud uživatele zajímá pouze malý počet nejbližších sousedů ($k \leq 5$), nedoporučuje se použít knihovnu *PyNNDescent*, protože její Recall@K s malým $k$ je velmi nízké 5.36. Obě knihovny založené na grafu ale s větším $k$ recall zlepšují a jsou schopny dobré aproximace i s nižšími nastaveními vlastních parametrů. Na ostatní knihovny, co se týče recallu, nemá hodnota $k$ zásadní vliv.

V rychlosti vyhledávání vyčnívá *Annoy*, *hnswlib* a *pykdtree*. *Annoy* ale pouze u nízkých nastavení a nízkého recallu 5.33. Knihovna *lshashing* neprokázala veliké zrychlení oproti naivnímu algoritmu a je z vybraných metod ve vyhledávání nejpomalejší 5.33.

| | **1.** | **2.** | **3.** | **4.** | **5.** | **6.** |
|---|---|---|---|---|---|---|
| **Paměť** | lshashing | pykdtree | BallTree | Annoy | hnswlib | PyNNDescent |
| **Rychlost konstr.** | Annoy | lshashing | pykdtree | BallTree | hnswlib | PyNNDescent |
| **Recall** | BallTree | hnswlib | Annoy | pykdtree | PyNNDescent | lshashing |
| **Rychlost vyhl.** | hnswlib | Annoy | pykdtree | PyNNDescent | BallTree | lshashing |
| **Velký počet dim.** | hnswlib | lshashing | PyNNDescent | Annoy | BallTree | pykdtree |

Tabulka 5.1: Autorova doporučení jednotlivých knihoven v podobě žebříčků sestavených na základě různých kritérií. Zelenou barvou jsou označeny excelující knihovny, červenou zcela nedoporučené.

# Kapitola 6

# Závěr

V práci byla popsána základní myšlenka algoritmu KNN a jeho metod pro nalezení nejbližších sousedů. Došlo k vyvození, že jeho oblíbenost vyplývá z jednoduché interpretovatelnosti. Bylo popsáno matematické pozadí, možnosti hodnocení aproximačních algoritmů, využití vzdálenostní funkce a problémy z ní plynoucí. Práce může sloužit jako doporučení tohoto algoritmu, pokud data problému splňují zmíněné podmínky 2.3. Ještě před výběrem metody KNN je totiž důležité zhodnotit, zda-li je algoritmus obecně vhodný vzhledem k datové sadě, nad kterou bude vyhledávání prováděno.

V kapitole o metodách byly popsány KD-Strom, Kulovitý strom, Locality-Sensitive Hashing, Strom náhodných projekcí a rodina algoritmů založené na grafu nejbližších sousedů. Detaily byly doplněny o vysvětlení postupů, pseudokódy a asymptotické složitosti. Metody byly uvedeny v základní podobě i s mnoha vylepšeními, které různé implementace nemusí obsahovat. Většina z metod (mimo grafové) se snaží zmenšit prohledávaný prostor, což se dá označit za konvenční přístup.

Pro každou ze zmíněných metod byla vybrána knihovna, zastupitel. Za pomocí měření v programu `compare` nad generovanými i reálnými daty bylo empiricky vyvozeno chování metod na datových sadách různých tvarů s různými nastaveními vlastních parametrů. Bylo poukázáno na případy, ve kterých konkrétní metody excelují ale také případy, kdy je vhodnější použít metodu jinou. Mnohé z výsledků experimentů byly předpovězeny v teorii a potvrzeny experimenty. Poznatků z experimentů není málo, výsledky jsou shrnuty ve vyhodnocení 5.2.1, které je doplněné o autorova doporučení 5.1 k případu použití jednotlivých metod/knihoven. Jelikož metody se liší přístupem řešení problému naivního algoritmu, každou se hodí použít za jiných podmínek. Žádná z testovaných metod není nejlepší v každé z hodnocených kategorií, tudíž neexistuje jedna nejlepší pro každý případ. Nýbrž je potřebné před výběrem zhodnotit, co je pro danou aplikaci KNN v systému důležité.

# Literatura


[1] *K-nearest neighbors (KNN)* [online]. Říjen 2022 [cit. 2023-16-01]. Dostupné z: https://www.ibm.com/docs/en/ias?topic=procedures-k-nearest-neighbors-knn.

[2] Aumüller, M., Bernhardsson, E. a Faithfull, A. J. ANN-Benchmarks: A Benchmarking Tool for Approximate Nearest Neighbor Algorithms. *CoRR*. 2018, abs/1807.05614. Dostupné z: http://arxiv.org/abs/1807.05614.

[3] Banerjee, A. *Computational Complexity of PCA* [online]. Srpen 2020 [cit. 2023-16-01]. Dostupné z: https://alekhyo.medium.com/computational-complexity-of-pca-4cb61143b7e5.

[4] Burgetová, I. *Shluková analýza*. FIT VUT v Brně, 2022 [cit. 2023-16-01].

[5] Chen, J., Fang, H. ren a Saad, Y. Fast Approximate kNN Graph Construction for High Dimensional Data via Recursive Lanczos Bisection. *Journal of Machine Learning Research*. 2009, sv. 10, č. 69, s. 1989–2012. Dostupné z: http://jmlr.org/papers/v10/chen09b.html.

[6] Chow, R. *K-Nearest Neighbors Algorithm: Classification and Regression Star* [online]. Srpen 2021 [cit. 2023-16-01]. Dostupné z: https://www.historyofdatascience.com/k-nearest-neighbors-algorithm-classification-and-regression-star/.

[7] Dasgupta, S. Random projection trees and low dimensional manifolds. In:. Květen 2008, s. 537–546. DOI: 10.1145/1374376.1374452.

[8] Dhulipalla, H. *Median of an unsorted array using Quick Select Algorithm* [online]. Listopad 2022 [cit. 2023-16-01]. Dostupné z: https://www.geeksforgeeks.org/median-of-an-unsorted-array-in-liner-time-on/.

[9] Dolatshah, M., Hadian, A. a Minaei, B. Ball*-tree: Efficient spatial indexing for constrained nearest-neighbor search in metric spaces. Listopad 2015.

[10] Dong, W., Charikar, M. a Li, K. Efficient K-nearest neighbor graph construction for generic similarity measures. In:. Březen 2011, s. 577–586. DOI: 10.1145/1963405.1963487.

[11] Emiris, I. *Computational Geometry: Search in High dimension and kd-trees*. ATHENA Research and Innovation Center, Greece, 2018 [cit. 2023-16-01]. Dostupné z: https://eclass.uoa.gr/modules/document/file.php/D42/%CE%94%CE%B9%CE%B1%CF%86%CE%AC%CE%BD%CE%B5%CE%B9%CE%B5%CF%82/3a.tree.pdf.

[12] Greenspan, M. a Yurick, M. Approximate k-d tree search for efficient ICP. In: *Fourth International Conference on 3-D Digital Imaging and Modeling, 2003. 3DIM 2003. Proceedings.* 2003, s. 442–448. DOI: 10.1109/IM.2003.1240280.

[13] Hard, R. *Diogenes the Cynic: Sayings and Anecdotes.* 1. vyd. Oxford University Press, 2012. ISBN 978-0-19-958924-1.

[14] Jayaswal, V. K-Nearest Neighbors (KNN) algorithm: An algorithm which finds the nearest neighbors. Srpen 2020.

[15] Kibriya, A. M. a Frank, E. An Empirical Comparison of Exact Nearest Neighbour Algorithms. In: Kok, J. N., Koronacki, J., Mantaras, R. Lopez de, Matwin, S., Mladenič, D. et al., ed. *Knowledge Discovery in Databases: PKDD 2007.* Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, s. 140–151. ISBN 978-3-540-74976-9.

[16] Macintosh, D. *Plato: A Theory of Forms* [online]. 2012 [cit. 2023-16-01]. Dostupné z: https://philosophynow.org/issues/90/Plato_A_Theory_of_Forms.

[17] Makwana, A. Understanding Recommendation system and KNN with project — Book Recommendation System. [online]. Prosinec 2020, [cit. 2023-09-05]. Dostupné z: https://aman-makwana101932.medium.com/understanding-recommendation-system-and-knn-with-project-book-recommendation-system-c648e47ff4f6.

[18] Malkov, Y., Ponomarenko, A., Logvinov, A. a Krylov, V. Approximate nearest neighbor algorithm based on navigable small world graphs. *Information Systems.* 2014, sv. 45, s. 61–68. DOI: https://doi.org/10.1016/j.is.2013.10.006. ISSN 0306-4379. Dostupné z: https://www.sciencedirect.com/science/article/pii/S0306437913001300.

[19] Munlika, R., Amorntip, P. a Chih Yi, C. Indexing in k-Nearest Neighbor Graph by Hash-Based Hill-Climbing. In: *2019 16th International Conference on Machine Vision Applications (MVA).* 2019, s. 1–4. DOI: 10.23919/MVA.2019.8758021.

[20] Nizri, M. *COVID-19 Dataset* [online]. Listopad 2022 [cit. 2023-16-01]. Dostupné z: https://www.kaggle.com/datasets/meirnizri/covid19-dataset.

[21] Ozaki, K. Mutual k-Nearest Neighbor Graphs for Semi-Supervised Learning. In:. 2011.

[22] Sangani, R. *Locality Sensitive Hashing in NLP.* Říjen 2021 [cit. 2023-16-01]. Dostupné z: https://towardsdatascience.com/locality-sensitive-hashing-in-nlp-1fb3d4a7ba9f.

[23] Singh, A. KNN algorithm: Introduction to K-Nearest Neighbors Algorithm for Regression. [online]. Srpen 2018, [cit. 2023-09-05]. Dostupné z: https://www.analyticsvidhya.com/blog/2018/08/k-nearest-neighbor-introduction-regression-python.

[24] Songsri in, M. R. A. P. C.-Y. C. K. Improving Cluster-Based Index Structure for Approximate Nearest Neighbor Graph Search by Deep Learning-Based Hill-Climbing. Září 2022.

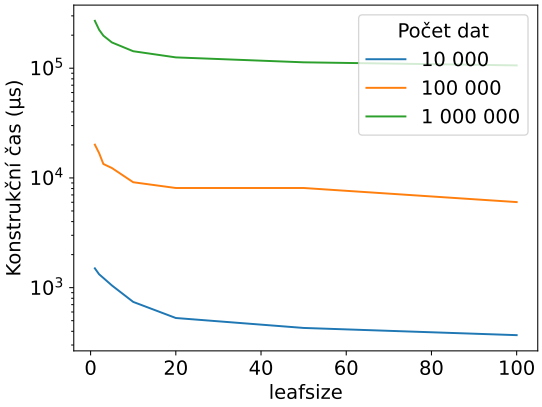
[25] Ting Liu, A. G. *New Algorithms for Efficient High-Dimensional Nonparametric Classification* [online]. Journal of Machine Learning Research, 2006 [cit. 2023-16-01]. Dostupné z: http://people.ee.duke.edu/~lcarin/liu06a.pdf.

[26] Tomašev, N., Radovanovic, M., Mladenić, D. a Ivanovic, M. A probabilistic approach to nearest-neighbor classification: Naive hubness Bayesian kNN. In:. říjen 2011, s. 2173–2176. DOI: 10.1145/2063576.2063919.

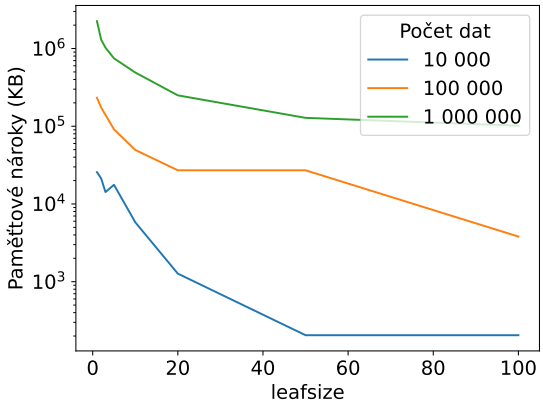
[27] Yiu, T. *The Curse of Dimensionality: Why High Dimensional Data Can Be So Troublesome* [online]. Červenec 2019 [cit. 2023-16-01]. Dostupné z: https://towardsdatascience.com/the-curse-of-dimensionality-50dc6e49aa1e.

# Příloha A

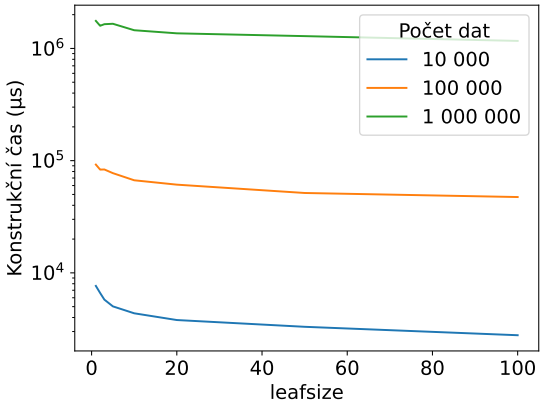
# Zbylé měření vlastních parametrů aproximačních metod



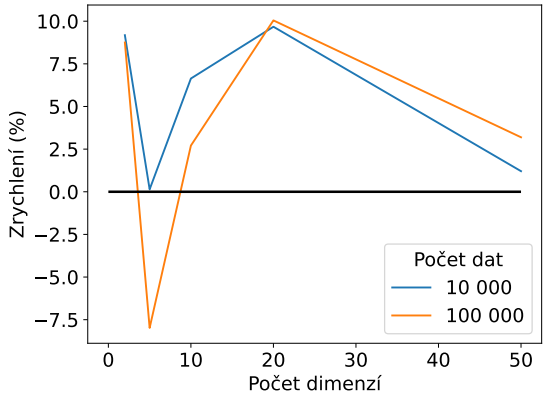

Obrázek A.1: Konstrukční čas *pykdtree* s různými hodnotami parametru `leafsize`.



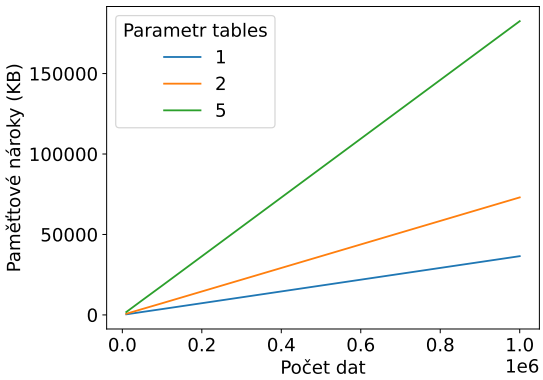

Obrázek A.2: Prostorová zátěž *pykdtree* s různými hodnotami parametru `leafsize`.



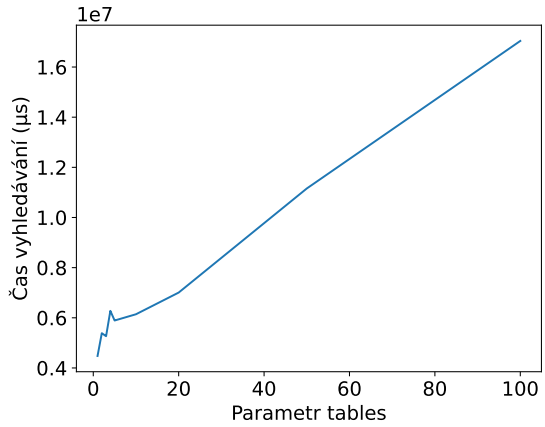

Obrázek A.3: Konstrukční čas *sklearn.neighbors.BallTree* s různými hodnotami parametru `leaf_size`.



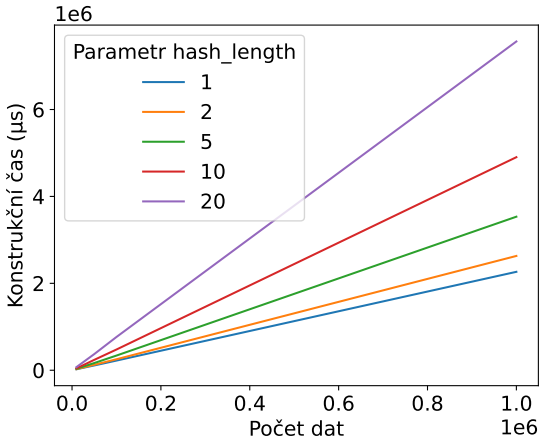

Obrázek A.4: Zrychlení při prohledávání do šířky oproti prohledávání do hloubky (parametr `breadth_first`) měřeno na datových sadách délky 10000 a 100000 s různým počtem dimenzí.



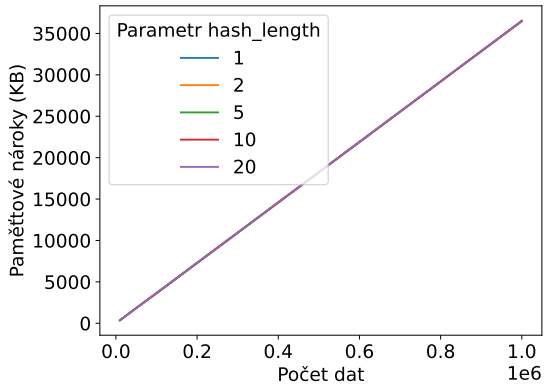

Obrázek A.5: Prostorová zátěž *lshashing* s různými hodnotami parametru `tables` a počtem dat.



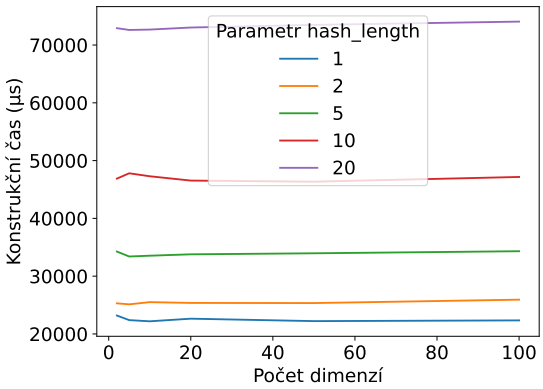

Obrázek A.6: Čas vyhledávání *lshashing* s různými hodnotami parametru `tables`.



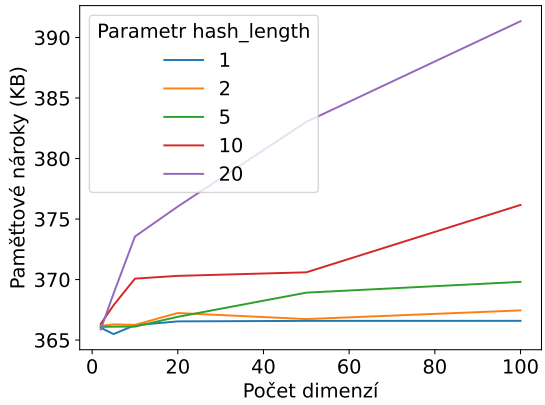

Obrázek A.7: Konstrukční čas *lsha-sshing* s různými hodnotami parametru `hash_length` a počtem dat.



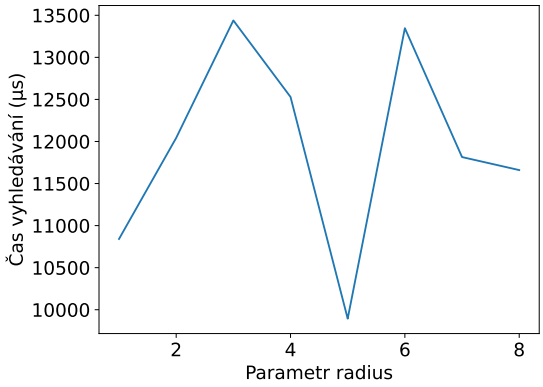

Obrázek A.8: Prostorová zátěž *lsha-sshing* s různými hodnotami parametru `hash_length` a počtem dat. Funkce jsou stejné a překrývají se.



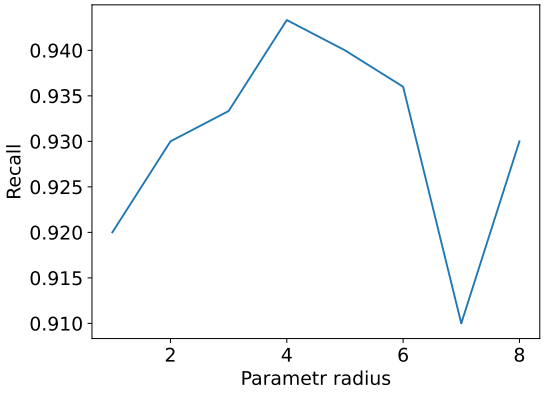

Obrázek A.9: Konstrukční čas *lsha-sshing* s různými hodnotami parametru `hash_length` a počtem dimenzí prostoru.



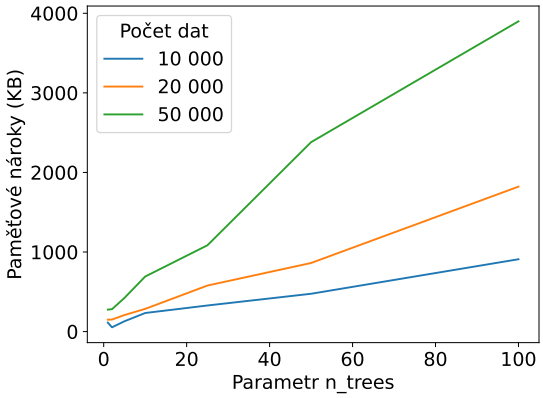

Obrázek A.10: Prostorová zátěž *lsha-sshing* s různými hodnotami parametru `hash_length` a počtem dimenzí prostoru.



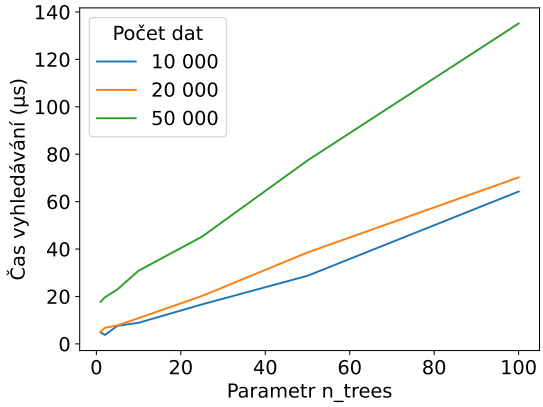

Obrázek A.11: Čas vyhledávání *lshashing* s různými hodnotami parametru `radius`.



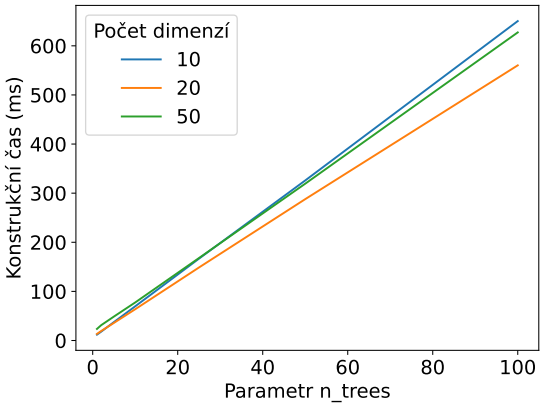

Obrázek A.12: Recall *lshashing* s různými hodnotami parametru `radius`.



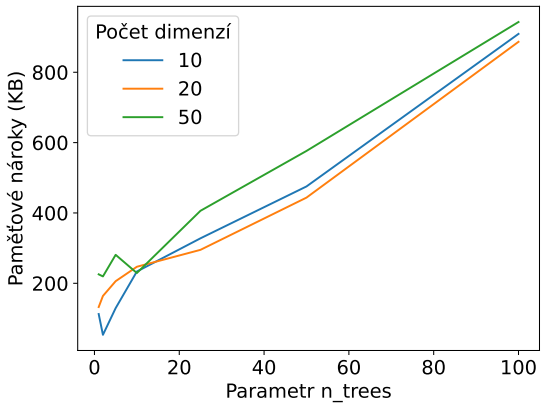

Obrázek A.13: Paměťové nároky *Annoy* s různými hodnotami parametru `n_trees` a počtem dat.



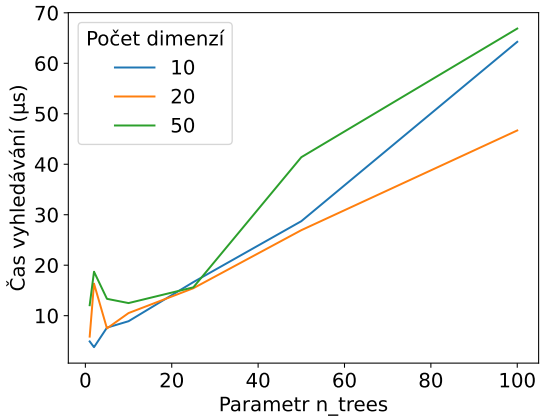

Obrázek A.14: Čas vyhledávání *Annoy* s různými hodnotami parametru `n_trees` a počtem dat.



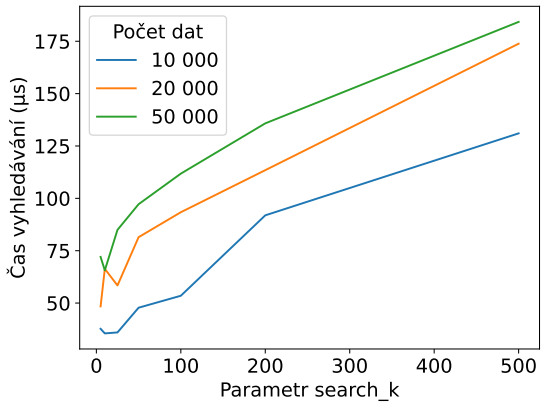

Obrázek A.15: Konstrukční čas *Annoy* s různými hodnotami parametru `n_trees` a počtem dimenzí.



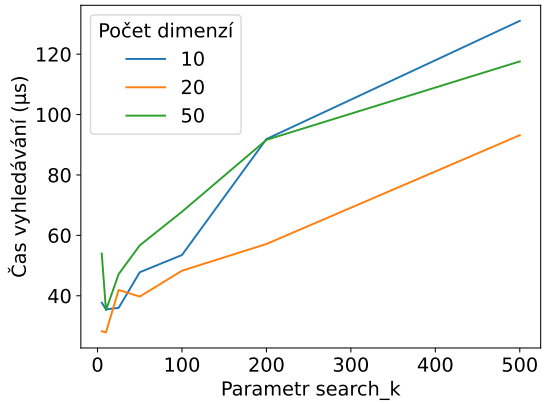

Obrázek A.16: Paměťové nároky *Annoy* s různými hodnotami parametru `n_trees` a počtem dimenzí.



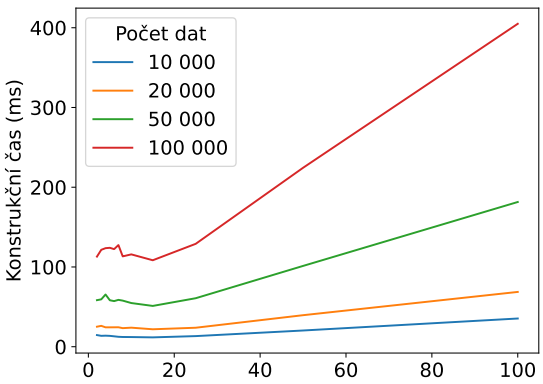

Obrázek A.17: Čas vyhledávání *Annoy* s různými hodnotami parametru `n_trees` a počtem dimenzí.



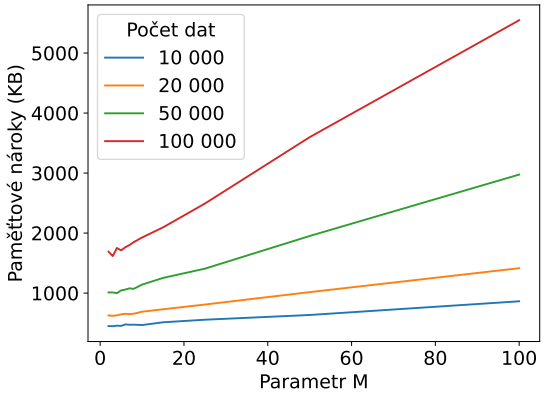

Obrázek A.18: Čas vyhledávání *Annoy* s různými hodnotami parametru `search_k` a počtem dat.



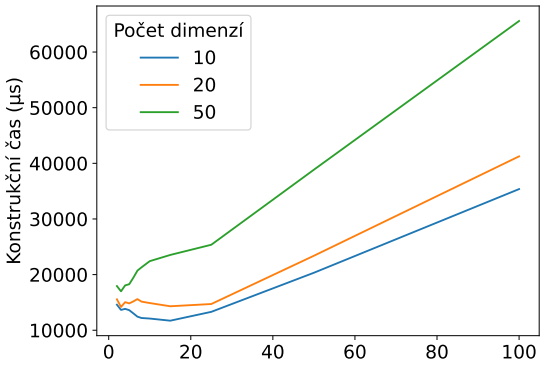

Obrázek A.19: Čas vyhledávání *Annoy* s různými hodnotami parametru `search_k` a počtem dimenzí.



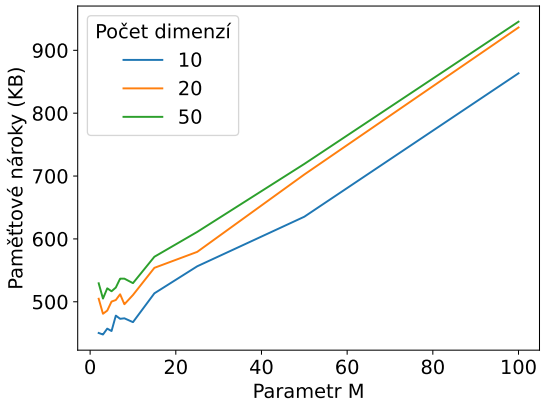

Obrázek A.20: Konstrukční čas *hnswlib* s různými hodnotami parametru `M` a počtem dat.



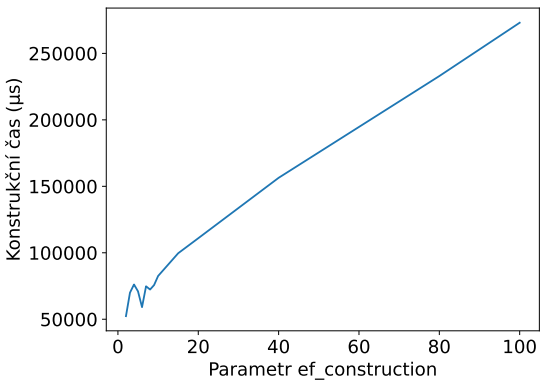

Obrázek A.21: Paměťové nároky *hnswlib* s různými hodnotami parametru `M` a počtem dat.



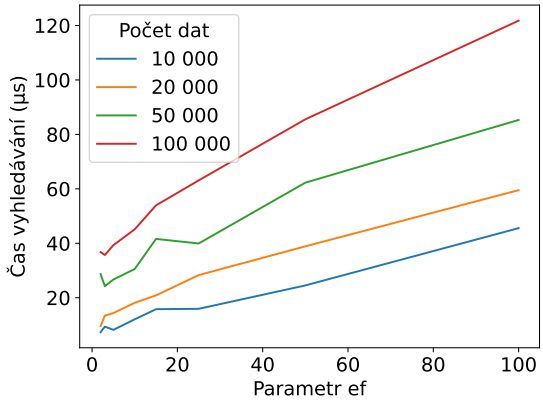

Obrázek A.22: Konstrukční čas *hnswlib* s různými hodnotami parametru `M` a počtem dimenzí.



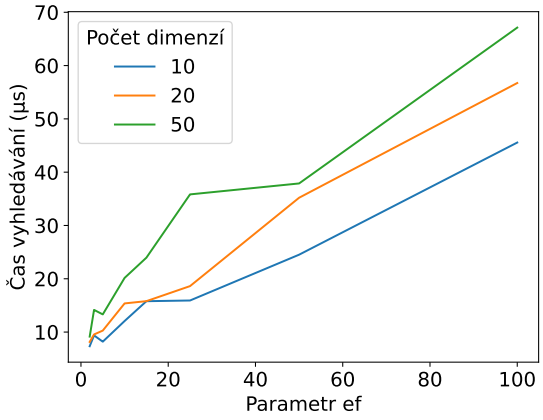

Obrázek A.23: Paměťové nároky *hnswlib* s různými hodnotami parametru `M` a počtem dimenzí.



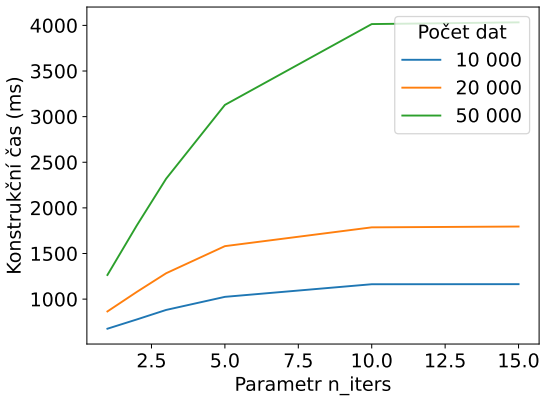

Obrázek A.24: Konstrukční čas *hnswlib* s různými hodnotami parametru `ef_construction`.



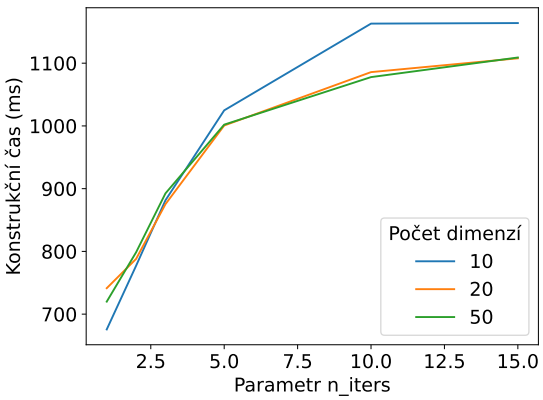

Obrázek A.25: Čas vyhledávání *hnswlib* s různými hodnotami parametru `ef` a počtem dat.



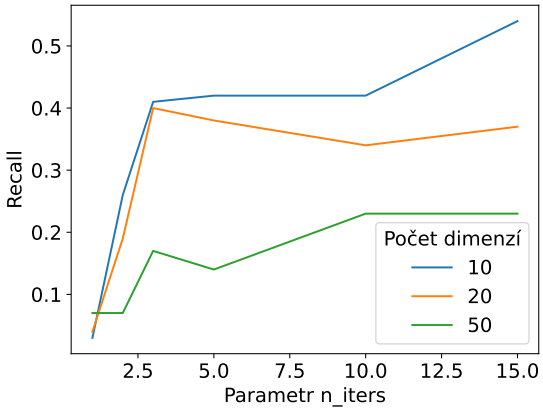

Obrázek A.26: Čas vyhledávání *hnswlib* s různými hodnotami parametru `ef` a počtem dimenzí.



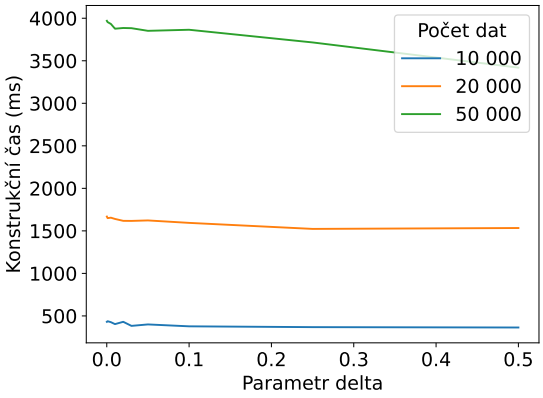

Obrázek A.27: Konstrukční čas *PyNNDescent* s různými hodnotami parametru `n_iters` a počtem dat.



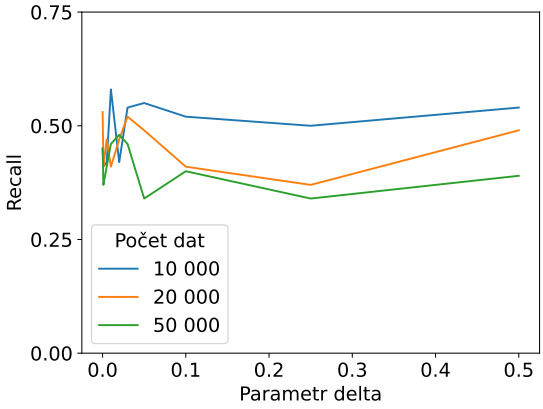

Obrázek A.28: Konstrukční čas *PyNNDescent* s různými hodnotami parametru `n_iters` a počtem dimenzí.



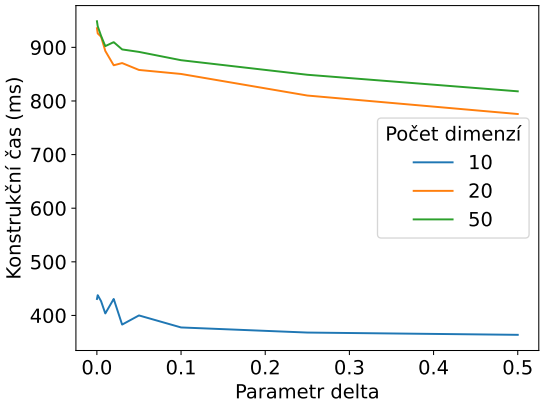

Obrázek A.29: Recall *PyNNDescent* s různými hodnotami parametru `n_iters` a počtem dimenzí.



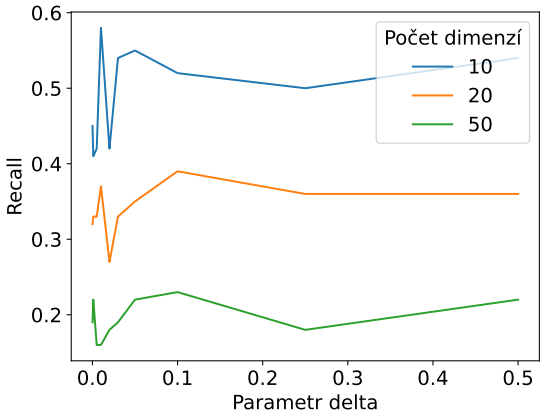

Obrázek A.30: Konstrukční čas *PyNNDescent* s různými hodnotami parametru `delta` a počtem dat.



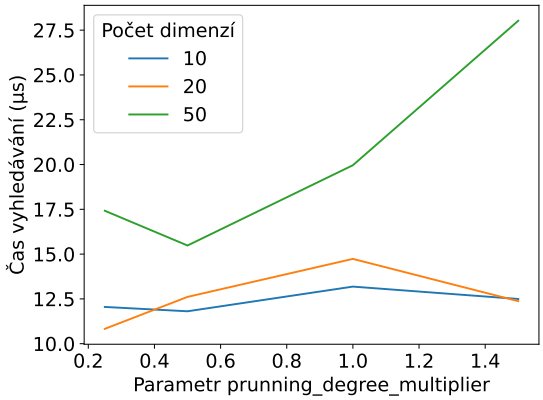

Obrázek A.31: Recall *PyNNDescent* s různými hodnotami parametru `delta` a počtem dat.



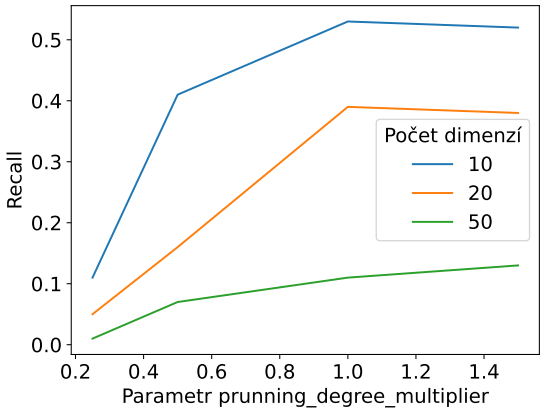

Obrázek A.32: Konstrukční čas *PyNNDescent* s různými hodnotami parametru `delta` a počtem dimenzí.



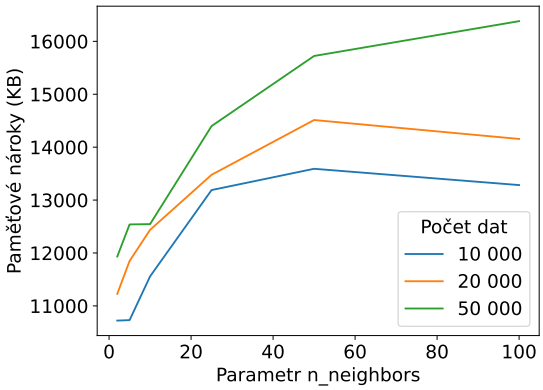

Obrázek A.33: Recall *PyNNDescent* s různými hodnotami parametru `delta` a počtem dimenzí.



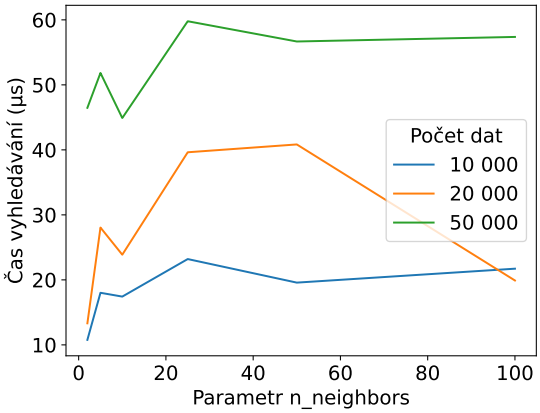

Obrázek A.34: Čas vyhledávání *PyNNDescent* s různými hodnotami parametru `prunning_degree_multiplier` a počtem dimenzí.



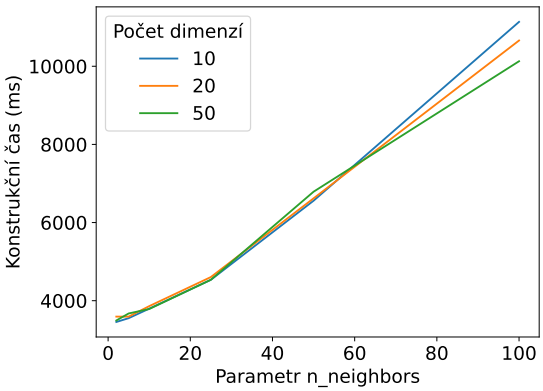

Obrázek A.35: Recall *PyNNDescent* s různými hodnotami parametru `prunning_degree_multiplier` a počtem dimenzí.



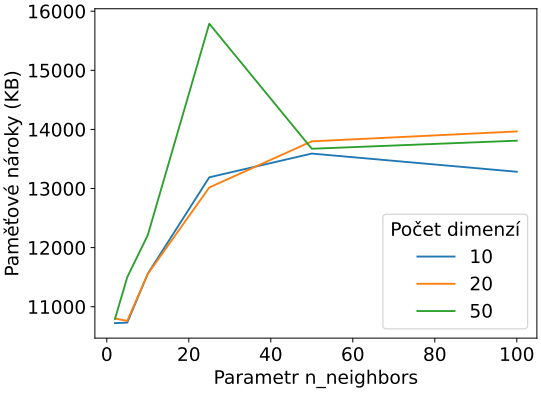

Obrázek A.36: Paměťové nároky *PyNN-Descent* s různými hodnotami parametru `n_neighbors` a počtem dat.



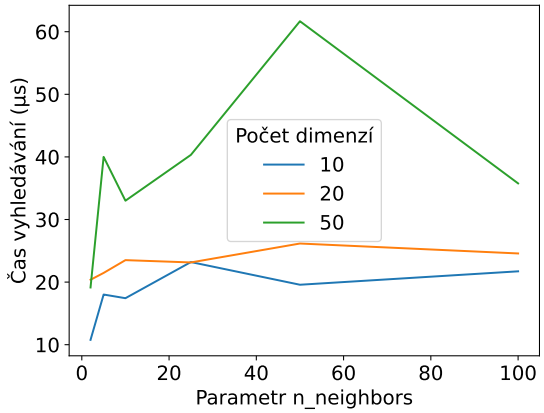

Obrázek A.37: Čas vyhledávání *PyNN-Descent* s různými hodnotami parametru `n_neighbors` a počtem dat.




Obrázek A.38: Konstrukční čas *PyNN-Descent* s různými hodnotami parametru `n_neighbors` a počtem dimenzí.




Obrázek A.39: Paměťové nároky *PyNN-Descent* s různými hodnotami parametru `n_neighbors` a počtem dimenzí.




Obrázek A.40: Čas vyhledávání *PyNNDescent* s různými hodnotami parametru `n_neighbors` a počtem dimenzí.