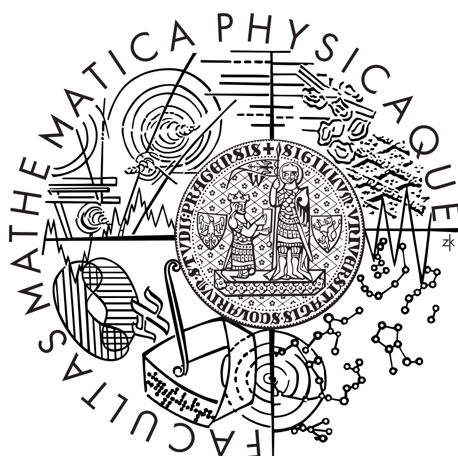
Univerzita Karlova v Praze

Matematicko-fyzikální fakulta

BAKALÁŘSKÁ PRÁCE



Ján Vojt

Vyhľadávanie relevantných článkov v rozsiahlych kolekciách

Katedra softwarového inženýrství

Vedúci bakalárskej práce: Ing. Jiří Novák

Študijný program: Informatika

Študijný obor: ISPS

2012

Na tomto mieste sa chcem poďakovať Ing. Jiřímu Novákovi za čas, ochotu a energiu vynaloženú na vedenie tejto bakalárskej práce. Ďalej sa chcem poďakovať RNDr. Vladislavovi Kuboňovi, Ph.D. za odborné rady a usmernenie v lingvistickej oblasti. Takisto ďakujem RNDr. Jaroslave Hlaváčovej, Ph.D. za predstavenie programu pre morfologickú analýzu českého jazyka.

Prehlasujem, že som túto bakalársku prácu vypracoval samostatne, pod vedením školiteľa Ing. Jiřího Nováka a že som všetky použité pramene riadne citoval.

Som si vedomý toho, že prípadné využitie výsledkov, získaných v tejto práci, mimo Univerzitu Karlovu v Prahe je možné iba po písomnom súhlase tejto univerzity.

V Prahe dňa ....   Podpis autora

Názov práce: Vyhľadávanie relevantných článkov v rozsiahlych kolekciách

Autor: Ján Vojt

Katedra: Katedra softwarového inženýrství

Vedúci bakalárskej práce: Ing. Jiří Novák, Katedra softwarového inženýrství

Abstrakt: Vyhľadávanie textu v článkoch sa štandardne rieši fulltextovým vyhľadávaním. Pri použití pokročilejších metód je možné dosiahnuť výrazne lepších výsledkov. Predmetom tejto práce je vytvoriť univerzálnu knižnicu na prehľadávanie rozsiahlych kolekcií, ktorá je prispôsobená pre český jazyk. Využíva nástroje schopné pracovať s morfológiou a zohľadňovať dôležitosť slov. Súčasťou je experiment so slovnými spojeniami, ktoré do vyhľadávania zapájajú kontext. Miera úspešnosti experimentu je overená na rozsiahlej kolekcii dát. Vytvorená knižnica je tak unikátnym nástrojom na spracovávanie rozsiahlych kolekcií českého textu, pričom je pripravená na rozšírenie o ďalšie jazyky a metódy.

Kľúčové slová: vyhľadávanie informácií, tf-idf, morfológia, stemming

Title: Searching relevant articles in extensive collections

Author: Ján Vojt

Department: Department of Software Engineering

Supervisor: Ing. Jiří Novák, Department of Software Engineering

Abstract: Searching text in articles is usually implemented with fulltext search. Using more advanced techniques however, it is possible to achieve significantly better results. The subject of this work is to create a universal library for searching extensible collections, specialized in czech language. The library makes use of tools capable of working with morphology while considering importance of words. It also conducts an experiment with word pairs, which adds context into the search process. The success rate of this experiment is tried on an extensible collection of data. Created library is a unique tool for processing extensible collections of czech text, while at the same time it is ready for further extension by new languages and methods.

Keywords: information retrieval, tf-idf, morphology, stemming

Obsah

Úvod **2**
Motivácia 2
Cieľ práce 2

1 Prehľad technológií **3**
1.1 Prehľad webových aplikácií 3
1.1.1 Vyhľadávanie podľa dotazu 3
1.1.2 Vyhľadávanie podľa dokumentu 4
1.1.3 Výber relevantnej reklamy 4
1.2 Prehľad používaných metód 5
1.2.1 Vektorový model 5
1.2.2 Metóda tf-idf 5
1.2.3 Lingvistické nástroje 8

2 Analýza návrhu **10**
2.1 Programovací jazyk 10
2.2 Databázový systém 10
2.3 Použité metódy 11
2.4 ER model 11
2.5 Objektový model 14
2.5.1 Reprezentácia dokumentov kolekcie 15
2.5.2 Načítanie dokumentov z externých zdrojov 15
2.5.3 Spracovanie dokumentu procesormi 15
2.5.4 Vyhľadávanie relevantných dokumentov 16
2.5.5 Správa cache 16
2.6 Minimálne požiadavky 16

3 API **17**
3.1 Modul pre spracovanie kolekcie 17
3.2 Vyhľadávací modul 19
3.3 Aktualizácia kolekcie 20

4 Výsledky **22**
4.1 Testovacie dáta 22
4.2 Metodika merania 22
4.3 Vyhodnotenie a diskusia 23

Záver **25**

Zoznam použitej literatúry **26**

Zoznam použitých termínov **27**

Prílohy **27**

Úvod

Motivácia

V dnešnej dobe sa často využívajú algoritmy na hľadanie relevantného obsahu k práve zobrazovanému textu. Majú naozaj široké využitie, od zobrazovania relevantnej reklamy cielenej na užívateľa s konkrétnymi záujmami, cez odhaľovanie plagiátorstva, až po prelinkovanie na ďalší podobný obsah s cieľom udržať si užívateľa na svojom portáli čo najdlhšie. Najtriviálnejšie využitie je vyhľadávanie v kolekcii dokumentov. Tu sa morfológia takmer nevyužíva ani u najpopulárnejších vyhľadávačoch webu. Na spravodajských portáloch sa väčšinou na konci článku zobrazí zoznam niekoľkých relevantných článkov. Avšak tieto články sú zobrazené, lebo sú z rovnakej rubriky, alebo sú pridané manuálne. V internetovej reklame sa točia veľké peniaze, no stále sa tu dostatočne nevyužívajú algoritmy schopné pracovať s morfológiou konkrétneho jazyka. Výskum v lingvistike neustále napreduje, no komerčná sféra je pozadu. Vládne tu názor, že zapojenie morfológie do analýzy textu neprináša výrazne presnejšie výsledky. V mojej práci budem mať za cieľ preskúmať, či sa takýto názor zakladá na pravde, hľadanie relevantného textu plne zautomatizovať, zefektívniť, a prispôsobiť gramatike a morfológii českého jazyka.

Cieľ práce

Cieľom práce je hľadanie relevantného textu s použitím stopwords a stemmera, prípadne iného morfologického nástroja. Program bude napísaný vo forme knižnice a optimalizovaný tak, aby sa dal univerzálne využívať najmä vo webových aplikáciách. Bude bežať na strane servera. Preto musí byť pripravený na veľký počet dotazov a tým pádom pracovať efektívne. Bude pracovať s databázou dokumentov, v ktorej bude vyhľadávať relevantné dokumenty k dokumentu danému na vstupe. Každému nájdenému priradí skóre relevantnosti. Program bude napísaný vo forme knižnice, aby bol ľahko implementovateľný do ľubovolnej webovej aplikácie. Program bude možné jednoducho rozšíriť o ďalšiu funkcionalitu, čo zabezpečí jeho univerzalitu.

1. Prehľad technológií

Pre začiatok sa rozhliadneme po už používaných riešeniach nášho problému. Internet sa dá nazvať obrovskou kolekciou kolekcií dokumentov, ktoré je nutné organizovať. To je samozrejme veľmi pracné a preto existuje veľa automatických nástrojov, ktoré sú schopné dokumenty kategorizovať, prehľadávať, prípadne porovnávať. Samozrejme väčšina spoločností si efektívne metódy a algoritmy chráni, preto nie je jednoduché (prakticky ani možné) detailne popísať napríklad algoritmus vyhľadávača Google. Pre naše účely ale postačí len povrchová analýza, z ktorej sa dá odhadnúť, ako približne daný algoritmus pracuje.

1.1 Prehľad webových aplikácií

V práci sa chceme zamerať na webové aplikácie. V tejto časti si teda predstavíme existujúce aplikácie, ktoré riešeniu nášho problému čelili, postavili sa proti nemu, prípadne sa proti nemu postaviť mali. Vždy sa budeme podľa možností snažiť vybrať čo najlepšie vyriešenú aplikáciu, aby sme v konkurencii našli aj inšpiráciu, nielen povzbudenie.

1.1.1 Vyhľadávanie podľa dotazu

Dotazom sa myslí malá skupina slov, na základe ktorej chceme nájsť dokument relevantný k danému dotazu. Tu sa treba zamyslieť, čo presne znamená slovo relevantný. Obecne môže byť dotazom aj otázka, ku ktorej je každá odpoveď relevantná. My sa ale v práci nebudeme snažiť odpovedať na otázky, pretože by sme museli používať umelú inteligenciu. Pre naše účely si teda definujme relevantnosť ako významovú podobnosť. V tomto ponímaní budeme používať pojem relevantnosť vo zvyšku práce. Všimnime si, že sa jedná o podobnosť významu, nie o zhodu slov, prípadne koreňov slov. Dôvodom je, aby pod našu definíciu relevantnosti spadali aj synonymá.

Najväčším reprezentantom vyhľadávania podľa dotazu je jednoznačne Google. Internet si môžeme predstaviť ako obrovskú kolekciu dokumentov, v ktorej Google vyhľadáva. Dostaneme tak práve problém, ktorý riešime. Skúsme si teda stručne rozobrať ako funguje Google. Na indexovanie používa takzvaný invertovaný index. Znamená to, že index stavia nad slovami obsiahnutými v dokumentoch. Vezme každé slovo z dotazu, nájde v ktorých dokumentoch sa nachádza, a vráti zoznam dokumentov, v ktorých sa nachádzajú všetky slová z dotazu. Vzhľadom k obrovskej kolekcii dokumentov sa toto samozrejme deje paralelne na niekoľko sto serveroch. Takto sa získa množina relevantných dokumentov, ktorú je ale potrebné zoradiť. To Google robí na základe cca 200 rôznych vstupov (PageRank, vek domény, lokalita servera, štrukturovanosť HTML dokumentu, atď.), ale to už je mimo nášho záujmu. Za zmienku tu stojí aj obmedzenie dĺžky dotazu. Bývalo to 10 slov, no dnes už nie je také prísne s 32 slovami. Do obmedzenia sa nerátajú stopwords. Význam obmedzenia súvisí so zložitosťou algoritmu, ktorá priamo závisí na počte slov v dotaze, čím sa budeme podrobnejšie zaoberať v kapitole o implementácii. My určite použijeme invertovaný index, pretože budeme dokumenty analyzovať a vyhľadávať v nich na úrovni slov.

Kam dál?

- Bouřky a přeháňky ustanou o víkendu, pak přijde mírné ochlazení
- Silné bouřky zatopily Klatovsko, hasiči museli evakuovat desítky lidí
- **Další články k tématu: Počasí**
- **Vstup do diskuse**, zpět na hlavní stranu: **Zprávy**

Obr. 1.1: Manuálna kategorizácia článkov na idnes.cz

Tu ale stojí zato si všimnúť, ako sa Google stavia a staval k morfologickému spracovaniu slov. Zo začiatku sa pomerne ostro postavil proti stemmovaniu, kedy tvrdil, že stemmer nepoužíva pre čo najpresnejšie výsledky k užívateľskému vstupu [1]. Neskôr v roku 2003 ale začal stemmovať základné anglické koncovky a robí tak dodnes [2]. Čo sa týka stopwords, tie Google používa takmer vždy s výnimkou slovných spojení ako napríklad „the matrix“ (názov filmu).

1.1.2 Vyhľadávanie podľa dokumentu

Tento typ vyhľadávania sa príliš nelíši od predchádzajúceho. V podstate jediným rozdielom je, že tu sa ako dotaz použije celý text hľadaného dokumentu. Google takto vyhľadávať neumožňuje kvôli spomínanému hornému obmedzeniu počtu slov v dotaze. Skúsime teda nájsť iný príklad. Ako prvé nás určite napadnú anti-plagiátorske systémy. My sa ale chceme venovať internetu, kde sú iné podmienky a požiadavky. Určite nás napadnú spravodajské portály, kde často bývajú na konci článku odporúčania relevantných článkov. Ideou je osloviť užívateľa témou, o ktorú už prejavil záujem prečítaním článku. Budeme sa zaujímať najmä o český jazyk, tak sa pozrime, ako to funguje napríklad v idnes.cz. Z obrázku 1.1 je zjavné, že články sú kategorizované manuálne. Považujem to nielen za neefektívne, ale môžu takto ľahko ujsť aj veľmi relevantné články. Podobne je tomu aj na iných českých spravodajských portáloch. V mojej práci sa budeme snažiť tento problém riešiť automaticky.

1.1.3 Výber relevantnej reklamy

Drvivá väčšina obsahu internetu je vo forme textu. Samotný text webovej stránky nám dokáže poskytnúť určité predpoklady o návštevníkovi. Užívateľ číta len čo ho zaujíma, preto je rozumné cieliť reklamu podľa obsahu okolo ktorého sa zobrazuje. Práve tu sa dá využiť náš program. Stačí aby inzerent k reklame pridal krátky text alebo skupinu slov a môžeme si kolekciu reklám predstaviť ako kolekciu dokumentov. Tieto dokumenty by mali predstavovať záujmy zákazníkov, ktorých chceme reklamou osloviť. Náš algoritmus potom dokáže k danému dokumentu priradiť najrelevantnejšiu reklamu. Toto samozrejme už dlho používa veľa spoločností na českom aj celosvetovom trhu s internetovou reklamou. My chceme vylepšiť algoritmus pre výber dokumentov tak, aby boli viac relevantné.

1.2 Prehľad používaných metód

Povrchne sme si popísali, prípadne odhadli, ako fungujú existujúce webové aplikácie. Doteraz sme ale nepriniesli žiadne exaktné a podrobné popísanie problému a jeho riešenia. V tejto časti si opíšme a zadefinujme ako presne fungujú najčastejšie používané metódy na vyhľadávanie textu. Na rozdiel od predošlej časti, tu sú postupy ľahko dostupné a podrobne popísané vo forme vedeckých článkov. Umožní nám to vybrať si pre nás najvýhodnejšie postupy, vhodne ich skombinovať a prípadne vylepšiť ich o vlastné riešenia.

$$\begin{pmatrix} & t_1 & t_1 & \dots & t_m \\ d_1 & w_{11} & w_{21} & \dots & w_{m1} \\ d_2 & w_{12} & w_{22} & \dots & w_{m2} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ d_n & w_{1n} & w_{2n} & \dots & w_{mn} \end{pmatrix}$$

Obr. 1.2: Reprezentácia kolekcie dokumentov maticou.

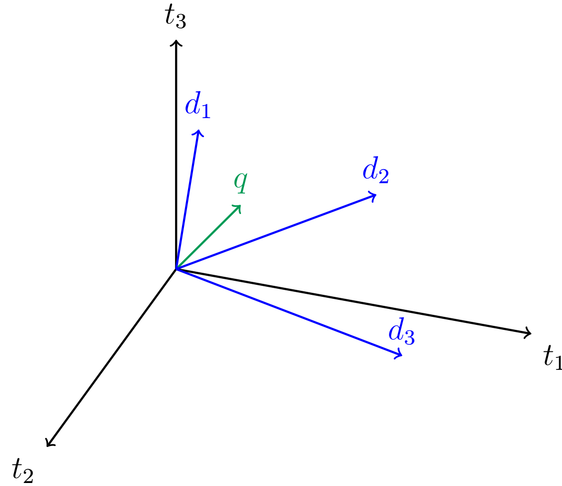
1.2.1 Vektorový model

Vektorový model je pomerne častá algebraická reprezentácia textových dokumentov ako vektorov. Každý rozmer vektora odpovedá práve jednému slovu zo slovníka a jeho veľkosť je váhou tohto slova. Slová, ktoré sa v dokumente nachádzajú, majú nenulovú váhu. Ostatné váhy sú nulové, čo nám pomôže algoritmus na prácu s vektormi podstatne zefektívniť. Kolekcia dokumentov je tak reprezentovaná maticou na obrázku 1.2. V literatúre sa nazýva „term–by–document“ matica, kde d_i je dokument, t_j slovo a w_{ji} váha slova t_j v dokumente d_i. Existuje niekoľko metód na presné určovanie váh. Pri takejto reprezentácii sa vyhľadávanie v kolekcii implementuje jednoduchým porovnávaním vektorov. Dajú sa tu využiť rôzne metriky, z ktorých niektoré ako si ukážeme budú viac vyhovujúce ako iné. Pri vyhľadávaní podľa dokumentu použijeme jeho vektor, ku ktorému budeme hľadať najbližšie vektory podľa zvolenej metriky. Ilustrácia je na obrázku 1.3. Pri vyhľadávaní podľa dotazu urobíme z dotazu dokument, z neho vyrobíme vektor a budeme ďalej postupovať rovnako ako pri hľadaní podľa dokumentu.

1.2.2 Metóda tf-idf

Táto metóda sa snaží každému slovu priradiť váhu podľa jeho dôležitosti v danom dokumente. Využíva pritom štatistiku z celej kolekcie. Váha slova stúpa priamo úmerne s výskytom slova v dokumente, ale pritom výrazne klesá s vysokou frekvenciou slova v celej kolekcii. Vďaka tomu je možné pomerne jednoducho určiť kľúčové slová pre každý dokument. Celá kolekcia je reprezentovaná vektorovým modelom popísaným v sekcii 1.2.1. Je to zrejme najpopulárnejšia metóda určovania váh slov. Poďme si detailne zadefinovať ako funguje.

Majme slovník termínov $D = \{t_1, t_2, \dots, t_m\}$ a kolekciu dokumentov $C = \{d_1, d_2, \dots, d_n\}$. Potom pre dokument $d_j \in C$ majme vektor $v_j = (w_1, w_2, \dots, w_m)$,



Obr. 1.3: Ukážka vektoru dokumentov a dotazu pri slovníku o veľkosti 3

kde komponenty vektora sú dané nejakou funkciou $\varphi_{tf-idf}(d_j, t_i) = w_i$ vo vektore v_j. Pred zadefinovaním funkcie φ sa zamyslime, čo k nej budeme potrebovať. Dôležité slová bývajú v dokumente spomenuté viackrát, čo nám môže dopomôcť pri ich určovaní. Definujme si preto:

$$f_{ij} = \text{počet výskytov } t_i \text{ v dokumente } d_j \tag{1.1}$$

Samotná frekvencia termínu v dokumente ale ako miera jeho dôležitosti nestačí. Predstavme si dokument d_x a d_y, kde d_y je dvakrát konkatenovaný d_x. Takto by d_y získal dvojnásobné váhy, pričom jeho význam by bol rovnaký. Tomuto problému predídeme normalizovaním frekvencie termínu nasledovným spôsobom:

$$tf_{ij} = \frac{f_{ij}}{max(\bigcup_{0<k<n}\{f_{ik}\})}, \tag{1.2}$$

kde $max(\bigcup_{0<k<n}\{f_{ik}\})$ je najvyššia frekvencia termínu t_i v rámci celej kolekcie C. Takto získané normalizované váhy termínov budú v intervale $\langle 0, 1\rangle$.

Výpočtom tf_{ij} získame štatistiku o slovách v dokumentoch. Avšak hodilo by sa nám zahrnúť aj štatistiku vypovedajúcu o slovách v celej kolekcii. Zaveďme si nasledovné označenie.

$$df_i = \text{počet dokumentov, v ktorých sa nachádza termín } t_i \tag{1.3}$$

Po krátkej úvahe zistíme, že df_i veľmi nevyhovuje našim potrebám. V prvom rade je nutné do výpočtu zakomponovať veľkosť kolekcie. Dostaneme tak $\frac{df_i}{n}$, no stále nám nevyhovuje nepriama úmera. Pri stúpajúcom df_i klesá dôležitosť t_i. To vyriešime jednoducho, stačí nám invertovať na $\frac{n}{df_i}$. Ďalej pre prispôsobenie rozsiahlym kolekciám dokumentov (veľkému n) zlogaritmujeme a dostávame tzv. inverznú frekvenciu termínu t_i v dokumentoch.

$$idf_i = \log_2(\frac{n}{df_i}) \tag{1.4}$$

Zostáva nám výpočet váhy slova, ktorý dostaneme skombinovaním predošlých štatistík slov v dokumentoch a v celej kolekcii. Chceme mať pritom pre dôležité slová vysokú váhu. S dôležitosťou slova stúpa tf_{ij} aj idf_i, z čoho dostaneme

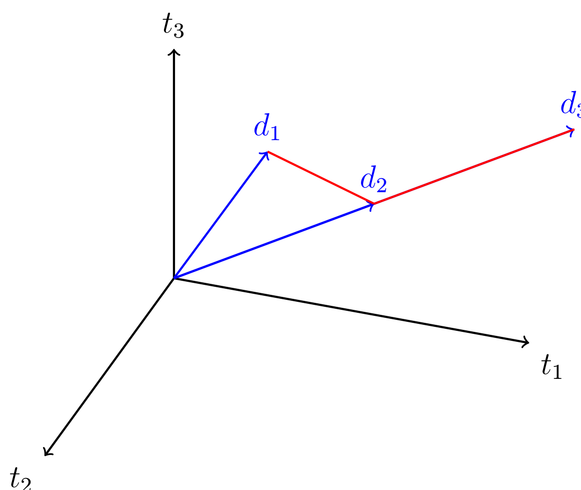
vhodný vzťah pre váhu slova jednoduchým vynásobením (1.2) a (1.4).

$$\varphi_{tf-idf}(d_j, t_i) = tf_{ij}idf_i = \frac{f_{ij}}{max(\bigcup_{0<k<n}\{f_{ik}\})} \log_2(\frac{n}{df_i}) = w_{ij} \tag{1.5}$$

Máme definované váhy, z nich vyplývajúci vektorový priestor, zostáva sa zamyslieť nad vhodným spôsobom určovania podobnosti vektorov. Intuitívne skúsme použiť euklidovskú metriku (L_2). Vzdialenosť dvoch vektorov by v tomto prípade bola definovaná nasledovne.

$$L_2(d_i, d_j) = \sqrt{\sum_{k=1}^{m} |w_{kj} - w_{ki}|} \tag{1.6}$$

Zamyslime sa nad prípadom, keď máme dokumenty d_1, d_2 a d_3, kde d_3 je dvakrát konkatenovaný d_2. V tomto prípade bude mať d_3 každý rozmer dvojnásobný. Ako je znázornené na obrázku 1.4, veľmi ľahko sa môže stať, že vzdialenosť medzi d_2 a d_3 je podľa euklidovskej metriky väčšia ako medzi d_1 a d_2. Pritom d_1 môže mať úplne odlišný text, zatiaľčo d_2 môže byť zhrnutý d_3. Z tohto dôvodu euklidovská metrika nie je príliš vhodná pre určovanie relevantnosti. Je namieste poznamenať, že tento nedostatok sa dá odstránť normalizovaním vektorov. To by ale prinieslo implementačný problém, ktorý ešte vysvetlíme.



Obr. 1.4: Ukážka nevhodnosti euklidovskej metriky pri určovaní podobnosti dokumentov

Keďže vzdialenosti sa ukázali ako neideálne, skúsme porovnávať vektory podľa uhlov, ktoré zvierajú. Nato môžeme využiť skalárny súčin. Definujme si teda podobnosť vektorov.

$$sim(d_i, d_j) = d_i \cdot d_j = \sum_{k=1}^{m} w_{ki} w_{kj} \tag{1.7}$$

Skalárny súčin je použiteľný, ale bolo by praktickejšie mať podobnosť vyjadrenú v pevnom intervale, napr. $\langle 0, 1 \rangle$. To docielime počítaním kosínusu pre uhol medzi vektormi. Uvedomme si, že hodnota kosínusu nebude nikdy záporná, pretože všetky váhy slov sú kladné a týmpádom každý vektor bude v kladnom hyperoktante.

Dostávame tak definíciu kosínusovej podobnosti, kde väčšia hodnota znamená menší uhol a teda relevantnejší dokument.

$$CosSim(d_i, d_j) = \frac{d_i \cdot d_j}{|d_i| \cdot |d_j|} = \frac{\sum_{k=1}^{m} w_{ki} w_{kj}}{\sum_{k=1}^{m} w_{ki}^2 \sum_{k=1}^{m} w_{kj}^2} \tag{1.8}$$

1.2.3 Lingvistické nástroje

Vieme ako reprezentovať dokumenty a slová, ale čo s vlastnosťami konkrétnych jazykov? Samozrejme, tie sa dajú ignorovať. Každú kombináciu písmen môžeme považovať za iné slovo. Avšak zrejme lepšie riešenie by bolo za „slovo" považovať množinu všetkých slov utvorených zo slovotvorného základu pomocou gramatických pravidiel daného jazyka. Ešte lepšie, môžeme ho považovať za množinu kombinácií písmen, ktoré nesú rovnaký význam. Aby sme v ďalšom texte mali jasno o čom hovoríme, zadefinujme si pár pojmov.

Definícia. *Slovo je kombinácia znakov z abecedy nejakého jazyka, ktorá neobsahuje medzeru.*

Definícia. *Morfém je v jazyku najmenšia jednotka nesúca význam.*

Definícia. *Lexém je základná jednotka slovnej zásoby. Je to množina všetkých tvarov určitého slova alebo slovného spojenia.*

Definícia. *Lemma je kanonický (slovníkový) tvar slova. V prirodzenom jazyku je lexém identifikovaný práve pomocou lemma jeho slov.*

Pre automatické spracovanie textu by bolo ideálne pracovať s lexémami. Avšak prirodzené jazyky majú často bohatú morfológiu a preto je zložité priradiť slovo ku lexému. Existujú rôzne nástroje, ktoré si určovanie lexémov kladú za cieľ, no žiadny to nedokáže so stopercentnou úspešnosťou. Niektoré sú k tomu ale dostatočne blízko. Obecne sa delia na stemmery a lemmatizéry. Stručne si každý predstavíme.

Stemmery pracujú pomerne triviálne, ale zato na rozdiel od iných metód nie sú výpočetne náročné. Obsahujú vopred pevne danú množinu pravidiel, ktorými dané slovo transformujú. Výsledkom týchto transformácií by malo byť lemma. Väčšina takýchto pravidiel len orezáva prefixy alebo suffixy slov, prípadne ich nahrádza za iné. Dôležité je poznamenať, že výsledné lemma tak nemusí byť platné slovo jazyka. Avšak pre potreby zachytenia významu to vôbec nie je problém. Ide len o priradenie slova do správneho lexému, a teda nejakej množiny. Čím túto množinu identifikujeme je nepodstatné. So stemmermi si uspokojivo vystačíme pri morfologicky chudobných jazykoch, akým je napríklad angličtina. Tu sa stemmery aj najviac používajú. Najznámejší a často považovaný za štandard je Porter Stemmer [3].

Lemmatizér je už podstatne zložitejší nástroj. Je vytvorený pre konkrétny jazyk. Výstupom je znovu lemma, avšak väčšina lemmatizérov k nemu vracia aj množinu morfologických značiek. Značka obsahuje informáciu o gramatických kategóriách slova. Lemmatizér ďalej obsahuje bohatý slovník, v ktorom má ku slovným tvarom uložené aj možné značky. Slovník samozrejme nie je kompletný, čo ale nemusí byť na škodu. Pri rozsiahlom slovníku sa totiž pridávaním okrajových slov jazyka zvyšuje homonymita, ktorá potom spôsobuje ďalšie problémy [4]. Často má

slovo viacero možných značiek, niekedy dokonca viacero lemma. V takých prípadoch je možné využiť morfologický tagger, ktorý podľa kontextu určí (prípadne skúsi uhádnuť) správnu značku. Tagger funguje na základe analýzy susedných slov a ich gramatických kategórií. Ak sa ani týmto spôsobom nevylúčia všetky značky až na jednu, tagger použije štatistické metódy a jednu vyberie. Lemmatizér by mal byť v každom prípade úspešnejší v určovaní lemma ako stemmer. Pri jazykoch s jednoduchou gramatikou je však tento rozdiel vzhľadom na jeho zložitosť zanedbateľný. Naopak pri jazykoch s bohatou morfológiou je spravidla podstatne úspešnejší [5].

2. Analýza návrhu

Pred začatím implementácie si rozoberme, čo vlastne budeme robiť a ako to budeme robiť. Chceme napísať univerzálnu knižnicu zameranú najmä na webové aplikácie. Budeme potrebovať databázu, v ktorej budeme vyhľadávať dokumenty. Táto databáza bude pomerne veľká, keďže bude pracovať s rozsiahlymi kolekciami dokumentov. Celá knižnica bude mať dva hlavné moduly. Jeden bude spracovávať kolekciu a bude spúšťaný buď v pravidelných intervaloch, alebo vždy po zmene v kolekcii. Druhý bude v spracovaných dátach vyhľadávať. Ten bude spúšťaný veľmi často a preto sa budeme snažiť o optimalizáciu najmä na tejto strane.

2.1 Programovací jazyk

Chceme mať univerzálnu knižnicu pre weby. V tejto oblasti je jednoznačne najpopulárnejší jazyk PHP. Je najpoužívanejší na strane developerov a takisto ho podporuje drvivá väčšina web serverov. Je stále aktívne vyvíjaný a má okolo seba obrovskú komunitu. Od verzie 5.3.1 má veľmi rozvinuté napríklad objekty, interfacy, namespacy a anonymné funkcie. To nám umožní napísať kvalitnú knižnicu, ktorá bude prehľadne rozdelená do častí, kde každá časť bude nezávazne riešiť svoj problém. Takisto bude knižnica ľahko rozšíriteľná, načo nebudeme zabúdať počas vývoja. Nevýhodou PHP je, že sa jedná o dynamický skriptovací jazyk. Tie sú obecne menej efektívne ako kompilované jazyky. Napriek tomu si myslím, že pozitíva prevyšujú a teda programovací jazyk sme si vybrali.

2.2 Databázový systém

Knižnica bude pracovať s rozsiahlou databázou. Najpopulárnejšie dostupné databázy sú Oracle, MS SQL Server, PostreSQL a MySQL. Okrem nich sú ešte občas využívané micro-databázy ako SQLLite. Tými sa ale nebudeme zaoberať, pretože sú veľmi obmedzené, neefektívne a týmpádom nevhodné pre našu robustnú databázu. Oracle je výkonná databáza, ktorá je ale dosť drahá. Jej cena je pre majiteľov webov často nevyhovujúca a tak volia iné varianty. U MS SQL Server je problém najmä platforma, kde je podporovaný len OS od Microsoftu. Ďalším problémom podobne ako u Oracle je cena. PostgreSQL je sama sebou označovaná ako najviac pokročilá open-source databáza. S týmto tvrdením sa dá súhlasiť. Veľa webových vývojárov siaha po PostgreSQL, no stále je u nich populárnejšia databáza MySQL. Tá je taktiež vydaná pod open-source licenciou a taktiež je multi-platformová. My budeme testovať na databáze MySQL, ale budeme sa knižnicu snažiť napísať tak, aby prechod na inú databázu nebol problém.

Keďže nechceme byť zaťažovaní programovaním vrstvy na komunikáciu s databázou, využijeme jednu z mnohých dostupných knižníc v PHP. Rozhodol som sa pre českú dibi [6]. Táto knižnica nám poskytne možnosť meniť používanú databázu zmenou jediného riadku v konfigurácii. Je to umožnené vďaka driverom, ktoré dibi obsahuje pre všetky rozšírené databázové servery. V prípade chýbajúceho drivera je samozrejme možné ho dopísať a všetko ostatné by malo fungovať.

2.3 Použité metódy

Máme vybrané prostriedky, poďme si rozmyslieť, ktoré z metód predstavených v sekcii 1.2 využijeme. Náš cieľ je vytvoriť knižnicu pracujúcu nad kolekciou dokumentov v konkrétnom jazyku. Multijazyčné kolekcie nebudeme podporovať. Preto je pre nás výhodné pracovať na úrovni slov. Tu sa javí ako vhodné použiť vektorový model. Jeho výhodou pre nás bude aj jeho vysoká parametrizovatelnosť cez váhy slov. Budeme tak mať možnosť posúvať váhu slova napríklad na základe našich znalostí o jeho morfológii.

V zadaní práce sme si stanovili aj využitie stopwords. Keď sa zamyslíme nad metódou tf-idf, uvedomíme si, že je vlastne zobecnením zoznamu stopwords. Zatiaľčo zvyčajne ho tvoria slová, ktoré sú úplne ignorované, tf-idf im priradí extrémne nízku váhu. Nemusíme si pritom vytvárať žiadny zoznam. Ako je vysvetlené v sekcii 1.2.2, dôležitosť slov je možné vypočítať automaticky na základe štatistík získaných z kolekcie.

Pre zaradenie slov do lexémov si vytvoríme český stemmer. Ten bude pracovať pomerne rýchlo, avšak pri pestrosti českého jazyka bude často robiť chyby. Preto nám príde vhod český lemmatizér MORFO, ktorý má vo svojom slovníku drvivú väčšinu českých lemma spolu s ich rôznymi tvarmi [7]. MORFO tak nahradí stemmer za cenu nižšej efektivity. To ale nie je problém. Môžeme nechať autora webovej aplikácie, ktorý bude našu knižnicu používať, nech sa rozhodne ktorá varianta mu najviac vyhovuje.

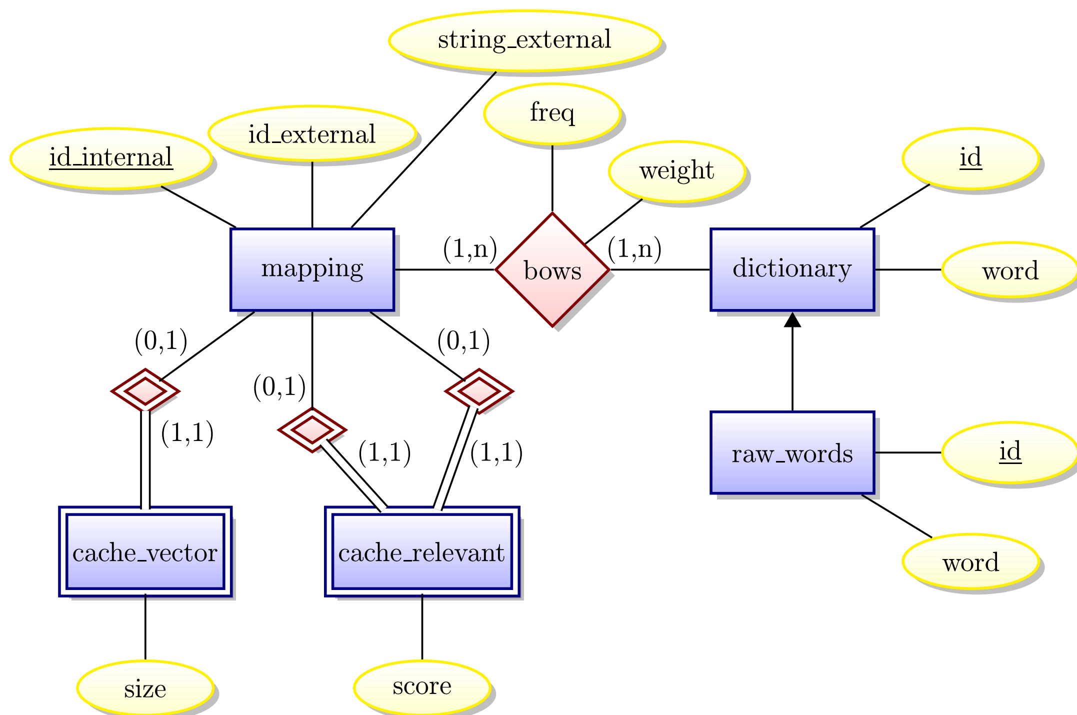
Okrem lemma slova získame aj jeho značky, ktoré využijeme na získanie slovných spojení. Pôjde o vlastný experiment tejto práce. Budeme hľadať vždy spojenia dvoch slov a spojenie následne pridáme do slovníka. Ďalej s ním pracujeme akoby to bolo jedno slovo. Aby sme dve slová považovali za spojenie, musia splňať všetky podmienky jednej z nasledujúcich skupín, kde L je ľavé slovo spojenia, R je pravé slovo spojenia:

1. (a) L je prídavné meno alebo číslovka, R je podstatné meno
   (b) medzi L a R nie je žiadne podstatné meno ani sloveso
   (c) L a R majú zhodné gramatické kategórie
   (d) medzi L a R nie je interpunkčné znamienko
2. (a) L a R sú podstatné mená
   (b) medzi L a R nie je žiadne iné slovo okrem predložky
   (c) L a R sú v rovnakom páde alebo R je v 2. páde (genitív)
   (d) medzi L a R nie je interpunkčné znamieko

Pri kombinácii s metódou tf-idf by mali vzhľadom k malému výskytu v kolekcii dostať vysokú váhu. Spojenia by tak podstatnou mierou do vyhľadávania zapojili kontext. Je to hypotéza, ktorú sa ešte pokúsime v ďalšom texte overiť.

2.4 ER model

Najväčšie nároky pri vyhľadávaní relevantných dokumentov budú kladené na databázu. Poďme preto hneď na začiatku navrhnúť entitno-relačný model 2.1.



Obr. 2.1: Entitno-relačný diagram.

Základnými entitami bude lexém a dokument. Kľúčovou tabuľkou pri vyhľadávaní bude tabuľka `bows`, ktorá bude spájať tieto entity. Dokument bude reprezentovaný ako multimnožina lexémov. Anglicky sa pojem často označuje ako „bag of words" – odtiaľ názov tabuľky. Bude obsahovať odkaz na lexém, odkaz na dokument, počet výskytov lexému v dokumente a váhu lexému v dokumente. Dôležité je správne tu nadefinovať indexy, pretože tabuľka bude robustná. Budú sa v nej veľmi často vyhľadávať lexémy a následne bude pre každý dokument spočítaná ich váha podľa vzorca 1.8. Aby sme po nájdení lexémov v indexe nemuseli z HDD ešte načítavať váhy, urobíme si pre slová zložený index spolu s váhami. MySQL ich potom vie využiť, čo nám ako sa ukázalo pri optimalizovaní dotazu na vyhľadávanie ušetrilo polovicu času. Pre úplnosť nasleduje definícia tabuľky.

```
CREATE TABLE `bows` (
  `id` int(11) unsigned NOT NULL AUTO_INCREMENT,
  `id_word` int(11) unsigned NOT NULL,
  `freq` tinyint(3) unsigned NOT NULL DEFAULT '1',
  `weight` float unsigned DEFAULT NULL,
  PRIMARY KEY (`id`,`id_word`),
  KEY `word_weight` (`id_word`,`weight`),
  CONSTRAINT `bows_ibfk_1` FOREIGN KEY (`id_word`)
   REFERENCES `dictionary` (`id`)
    ON DELETE CASCADE ON UPDATE CASCADE
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

Tabuľka `dictionary` obsahuje zoznam lexémov. Každý lexém má umelé ID, na ktoré sa odkazuje tabuľka `bows`, aby sme minimalizovali veľkosť domény indexu

pre lexémy. Okrem ID už obsahuje jediný stĺpec pre lexém samotný – základný tvar slova. Pri vyhľadávaní podľa dokumentu z kolekcie nám konkrétne slová budú zbytočné, pretože nám postačí pracovať s ich umelým ID. Avšak pri hľadaní podľa užívateľom zadaného dotazu budeme potrebovať práve túto tabuľku, aby sme zo zadaných slov vytvorili vektor.

```
CREATE TABLE `dictionary` (
  `id` int(11) unsigned NOT NULL AUTO_INCREMENT,
  `word` varchar(45)
   CHARACTER SET utf8 COLLATE utf8_bin NOT NULL,
  PRIMARY KEY (`id`),
  UNIQUE KEY `word` (`word`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

Tabuľka `raw_words` obsahuje vlastné umelé ID, slovo, ako bolo uvedené v pôvodnom texte, a dedí lexém od tabuľky `dictionary`, kde sa odkazuje cudzím kľúčom. Nie je veľmi dôležitá a svoje využitie má najmä pri hľadaní chýb v kóde. Umožňuje vývojárovi vidieť aké slovo bolo priradéne k akému lexému.

```
CREATE TABLE `raw_words` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `id_word` int(11) unsigned NOT NULL,
  `word` varchar(45) DEFAULT NULL,
  PRIMARY KEY (`id`),
  KEY `id_word` (`id_word`),
  CONSTRAINT `raw_words_ibfk_1`
   FOREIGN KEY (`id_word`)
   REFERENCES `dictionary` (`id`)
   ON DELETE CASCADE ON UPDATE CASCADE
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

Tabuľka `mapping` slúži na mapovanie externého identifikátora dokumentu na interný identifikátor knižnice. Umožňuje tak odkazovať sa na dokumenty v aplikácii navonok ľubovolným reťazcom a pritom interne používať efektívnejšie identifikátory s malou doménou.

```
CREATE TABLE `mapping` (
  `id_internal` int(11) unsigned NOT NULL,
  `id_external` int(11) unsigned DEFAULT NULL,
  `string_external` varchar(55) DEFAULT NULL,
  PRIMARY KEY (`id_internal`),
  UNIQUE KEY `string_external` (`string_external`),
  UNIQUE KEY `id_external` (`id_external`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

Tabuľka `cache_vector` obsahuje predpočítané veľkosti vektorov. Tie využijeme vo vzorci 1.8 ($CosSim(d_i, d_j) = \frac{d_i \cdot d_j}{|d_i| \cdot |d_j|}$) pri počítaní kosínusovej podobnosti. Stačí predpočítané veľkosti porovnávaných vektorov vynásobiť a máme hotový menovateľ. Čitateľ zlomku sa už ale musí počítať pri každom hľadaní, pretože tu sú komponenty vektorov násobené navzájom vo vnútri sumy. Pri ich cachovaní by sme sa museli uspokojiť s kvadratickou náročnosťou vzhľadom k celkovému počtu dokumentov, čomu sa chceme vyvarovať.

```
CREATE TABLE `cache_vector` (
  `bow_id` int(11) unsigned NOT NULL,
  `size` float unsigned NOT NULL,
  PRIMARY KEY (`bow_id`),
  CONSTRAINT `cache_vector_ibfk_1`
   FOREIGN KEY (`bow_id`)
   REFERENCES `mapping` (`id_internal`)
   ON DELETE CASCADE ON UPDATE CASCADE
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

Tabuľka `cache_relevant` slúži ako cache pre vyhľadávenie podľa dokumentu z kolekcie. Dotaz na vyhľadávanie je dobre optimalizovaný, ale pri obrovských kolekciách dokumentov je vyhľadať relevantné stále zložitá operácia. Pritom veľká časť dokumentov z kolekcie nie je používaná napríklad kvôli neaktuálnosti. Naopak populárne dokumenty sa prezerajú veľmi často. Keď už nejaké relevantné dokumenty nájdeme, je rozumné si výsledok uložiť. Pri ďalšom prezretí iným užívateľom tak budeme schopný načítať výsledky hľadania z tejto tabuľky takmer okamžite.

```
CREATE TABLE `cache_relevant` (
  `bow_id_1` int(10) unsigned NOT NULL,
  `bow_id_2` int(10) unsigned NOT NULL,
  `score` float unsigned DEFAULT NULL,
  PRIMARY KEY (`bow_id_1`,`bow_id_2`),
  KEY `bow_id_2` (`bow_id_2`),
  CONSTRAINT `cache_relevant_ibfk_1`
   FOREIGN KEY (`bow_id_2`)
   REFERENCES `mapping` (`id_internal`)
   ON DELETE CASCADE ON UPDATE CASCADE,
  CONSTRAINT `cache_relevant_ibfk_2`
   FOREIGN KEY (`bow_id_1`)
   REFERENCES `mapping` (`id_internal`)
   ON DELETE CASCADE ON UPDATE CASCADE
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

2.5 Objektový model

Vytvárame univerzálnu a rozšíriteľnú knižnicu, preto si potrebujeme dobre premyslieť objektový model. Celú úlohu si rozdelíme na podproblémy a tie budeme

riešiť v jednotlivých objektoch. Umožní to jednoduchú úpravu kódu, kde každú vlastnosť knižnice bude možné upraviť na jednom mieste bez potreby refaktorovania existujúceho kódu. Toto je minimálne náš ideál, ku ktorému sa budeme snažiť priblížiť rozumným návrhom objektového modelu. Vyjasnime si, aké problémy bude naša knižnica riešiť.

- Reprezentácia dokumentov kolekcie
- Načítanie dokumentov z externých zdrojov
- Spracovanie dokumentu procesormi
- Vyhľadávanie relevantných dokumentov
- Správa cache

Všimnime si, že v zozname sa nenachádza zmienka o ukladaní entít do databázy. Toto si totiž bude riešiť každá entita sama. Použitím ORM by sme síce dosiahli lepšej enkapsulácie a prehľadnosti kódu, avšak znížilo by to efektivitu. Keď sa každá entita bude sama starať o svoju perzistenciu v databáze, bude možné tento proces optimalizovať na pomerne nízkej úrovni. Kompletný UML diagram objektového modelu sa nachádza na priloženom disku.

2.5.1 Reprezentácia dokumentov kolekcie

Lexémy bude reprezentovať objekt `Word`. Od neho bude dediť objekt `RawWord` a bude reprezentovať slovo. Okrem pôvodného tvaru slova z dokumentu bude teda obsahovať aj jeho lemma. Dokument bude reprezentovaný objektom `WordCollection`, ktorý predstavuje multimnožinu slov. Pre efektívnejšiu prácu si `WordCollection` stavia index. V ňom si po pridaní slova do kolekcie zaindexuje jeho lemma a uloží si tu objekt `ScoredWord`, ktorý obsahuje odkaz na `Word` spolu s jeho skóre (frekvenciou výskytu v dokumente).

2.5.2 Načítanie dokumentov z externých zdrojov

Tento problém riešia implementácie interfacu `IDataConverter`, ktorý konvertuje dokumenty z externého úložišťa na `WordCollection`. My si ich pre demo ukážky naimplementujeme hneď niekoľko. `DatabaseDataConverter` na načítanie dát z databázy, `FileDataConvertor` na načítanie dát zo súborov uložených na disku, `ResourceConverter` na načítanie dát z ľubovolného PHP resourcu a `StringDataConverter` pre načítanie dát z reťazca uloženého v premennej. Užívateľ našej knižnice si takto môže jednoducho doimplementovať ľubovolný zdroj dát, čo je veľmi flexibilné. Ukážky použitia napríklad `DatabaseDataConverter`u nájdeme v kapitole 3, ďalšie ukážky v priloženom disku.

2.5.3 Spracovanie dokumentu procesormi

V kontexte práce s dokumentami budeme procesorom nazývať nástroj, ktorý dokáže nejako spracovať slovo. Našim cieľom bude spracovať slovo na jeho lemma. Budeme mať dva typy procesorov. Prvý bude spracovávať slová a implementovať

interface `IWordProcessor`. Druhý bude implementovať `ICollectionProcessor` a bude spracovávať `WordCollection`. Napríklad stemmer bude pracovať vždy len na úrovni slov, preto patrí do prvej skupiny. Lemmatizér často pracuje s kontextom, a preto bude pracovať s celou multimnožinou slov a spracovávať slová, ktoré obsahuje. My si naimplementujeme stemmery `CzechStemmer` inšpirovaný od prof. Savoya [8] a `SlovakSimpleStemmer`. Ďalej naimplementujeme lemmatizér `MorfoProcessor`, ktorý bude pracovať s morfologickou analýzou programu MORFO [7] a bude navyše obsahovať náš experiment so slovnými spojeniami, popísanými v sekcii 2.3.

2.5.4 Vyhľadávanie relevantných dokumentov

Vyhľadávanie relevantných dokumentov má na starosti objekt `SearchEngine`. Jeho metóda `getRelevant(mixed $idExternal)` slúži na vyhľadávanie podľa externého ID dokumentu, ktoré je prvým argumentom. Druhá metóda `searchByBow (WordCollection $coll)` vyhľadáva podľa množiny slov, zadanej ako argument. Využitie nájde napríklad pri hľadaní výsledkov na základe užívateľského dotazu. `SearchEngine` má ďalej setter `setCache(boolean $cache)`, ktorým sa dá prepínať použitie cache. Má v sebe implementované aj stránkovanie, čo zaručí efektívne vyhľadávanie len potrebného počtu výsledkov. Poznamenajme, že `SearchEngine` je optimalizovaný pre databázu MySQL. Preto pri použití inej databázy by pravdepodobne bolo potrebné upraviť niektoré SQL dotazy, ktoré používa.

2.5.5 Správa cache

Správa cache je implementovaná v objekte `WeightCalculator`. Používa dve rôzne cache. Jedna je určená na predpočítavanie veľkostí vektorov, čo je popísané v sekcii 2.4 pri tabuľke `cache_vector`. Táto cache sa aktualizuje spustením metódy `updateWeights()`. Druhá cache slúži na cachovanie výsledkov hľadania podľa ID dokumentu. Obsahuje páry dokumentov spolu s ich skóre vzájomnej relevantnosti. Kvôli kvadratickej náročnosti je cache implementovaná ako *lazy* a vytváraná „on demand“ počas behu aplikácie. Vymaže sa metódou `clearCache()`, ktorá by mala byť spúšťaná po aktualizácii kolekcie.

2.6 Minimálne požiadavky

Knižnica podporuje ľubovolnú platformu, na ktorej beží PHP >= 5.3.1 a MySQL >= 4.1. Program MORFO oficiálne podporuje len platformu Linux, preto ak budeme chcieť používať `MorfoProcessor`, budeme musieť zvoliť túto platformu. Hardwarové požiadavky závisia na veľkosti kolekcie a miere trpezlivosti užívateľa. Minimun sú požiadavky MySQL (ktoré nie sú prakticky žiadne), prípadne inej použitej databázy. V prípade vzdialeného databázového serveru odpadá i táto požiadavka.

3. API

Predstavíme si API našej knižnice. Uvedieme základné vlastnosti a predvedieme ich na praktických príkladoch. Kompletné API sa nachádza na priloženom disku. Začneme načítaním knižnice dibi [6], ktorá nám uľahčí komunikáciu s databázou. Ďalej načítame našu knižnicu *RelevantText*.

```
require_once dirname(__FILE__) . "/libs/dibi/dibi.php";
require_once dirname(__FILE__) . "/libs/reltext/
RelevantText.php";
```

Bude užitočné, ak si na jednom mieste definujeme konfiguráciu pripojenia k databáze. Tu bude uložená celá ER schéma našej knižnice, ako je nakreslená v sekcii 2.4. Detailný popis konfigurácie nájdeme v dokumentácii dibi [6], no z ukážkového kódu by malo byť všetko zrejmé. Tento kód je nutný pred každou nasledujúcou ukážkou v tejto kapitole.

```
// define database config
Environment::setDbLogin(array(
    'user' => 'root',
    'password' => 'heslo',
    'driver' => 'mysql',
    'host' => 'localhost',
    'database' => 'reltext',
    'charset' => 'utf8',
    'lazy' => '1', // connect upon first query
));
```

3.1 Modul pre spracovanie kolekcie

Začneme modulom pre spracovanie externej kolekcie do vlastných dátových štruktúr, v ktorých bude jednoduché vyhľadávať. Obecný postup je logický:

1. pripojíme sa k databáze
2. nadefinujeme si procesory
3. nadefinujeme si externí zdroj kolekcie
4. spracujeme a uložíme dokumenty

V nasledujúcej ukážke použijeme `MorfoProcessor` pre určenie lemma slova. Ako externý zdroj kolekcie použijeme databázu „zpravy“. Tu sa nám hodí už spomínaný `DatabaseDataConverter`. Predáme mu konfiguráciu pripojenia a názov tabuľky a stĺpcov, z ktorých načíta dáta. Tie skonvertuje na `WordCollection`, ktorý predáme `MorfoProcessoru` na spracovanie a následne uložíme.

```php
// setup processor
$processor = new MorfoProcessor();

// define collection source
$db = array(
    'user' => 'root',
    'password' => 'heslo',
    'driver' => 'mysql',
    'host' => 'localhost',
    'database' => 'zpravy',
    'charset' => 'utf8',
);

$convertor = new DatabaseDataConverter();
$convertor->setDbLogin($db)
    ->setIdColumn('id')
    ->setDataColumn('text')
    ->setTable('clanky');

// process and save BOWs
while($bow = $convertor->getBow(true)) {
    $processor->process($bow);
    $bow->save();
}
```

Užívateľ si tu samozrejme môže nadefinovať ľubovolný zdroj dát s použitím iného `IDataConverteru`. Takisto môže zvoliť ľubovolný, prípadne žiadny, procesor pre spracovanie textu. Procesorry sa dokonca dajú kombinovať pomocou `ProcessorCollection` a celá skupina procesorov sa pritom správa ako jeden. Ukážeme si jednoduchý príklad.

```php
// define collection of processors
$processors = new ProcessorCollection();
$processors
    ->append(new CzechStemmer())
    ->append(new RemoveAccentsProcessor())
    ->append(new LowerCaseProcessor());

// process BOW
$bow->process($processors);
```

Tu sme použili ako procesor český stemmer, skombinovali sme ho s procesorom na odstránenie diakritiky a procesorom, ktorý zmení všetky písmena na malé. Takto spracované slová budú pri vyhľadávaní považované za rovnaké bez ohľadu na diakritiku či veľkosť písmen. Poznamenajme ešte, že poradie v ktorom pridáme procesory do kolekcie je dôležité, pretože v tomto poradí bude slovo spracovávané.

Všimnime si, ako sa líši spracovávanie pri použití `CzechStemmer`a a v predošlej ukážke `MorfoProcessor`u. Je to preto, že jeden implementuje `IWordProcessor` a druhý `ICollectionProcessor`.

```
// define processor
$wordproc = new CzechStemmer();
$colproc = new MorfoProcessor();

// process BOW
$bow->process($wordproc);
$colproc->process($bow);
```

3.2 Vyhľadávací modul

Vyhľadávanie podľa ID dokumentu je jednoduché, pretože okrem tohto externého ID nepotrebujeme žiadny iný vstup. Keďže sa predpokladá rozsiahla kolekcia, je praktické použiť stránkovanie výsledkov. To nám `SearchEngine` umožňuje. V nasledujúcej ukážke vyhľadávame 16. až 20. najrelevantnejší dokument. Výsledok v premennej `$relevant` tak bude obsahovať pole o veľkosti 5, kde každý prvok bude pole s ID nájdeného dokumentu a jeho skóre. Dokumenty vo výsledku budú samozrejme zoradené zostupne podľa skóre.

```
$engine = new SearchEngine();
$relevant = $engine->setOffset(15)
        ->setLimit(5)
        ->getRelevant($extId);
```

Teraz si ukážeme vyhľadávanie podľa užívateľom zadaného dotazu. Bude to trochu zložitejšie, pretože užívateľom zadaný reťazec musíme spracovať. Výrazne nám to však uľahčia pripravené nástroje. Najprv vytvoríme `WordCollection` z reťazca zadaného užívateľom pomocou `StringDataConverter`u. Ďalej si nadefinujeme procesory. Tu je dôležité nadefinovať ich rovnako ako pri vytváraní kolekcie. Inak by sa rovnaké slová mohli spracovať rôzne a neboli by považované za identické. Je teda vhodné mať procesory nadefinované niekde bokom na jednom mieste a potom ich includovať kde je potreba. V ďalšom kroku spracujeme kolekciu slov. Teraz máme všetko pripravené a môžeme hľadať pomocou metódy `searchByBow()`, kde dáme ako argument spracovanú kolekciu slov od užívateľa. Rovnako ako v predošlom príklade môžeme využiť stránkovanie. Pri použití limitu na stránkovanie by sme ľahko mohli stratiť prehľad o počte celkovo nájdených relevantných dokumentov. K tomu slúži metóda `totalBows()`, ktorá nám tento počet vráti.

```
// build word collection
$convertor = new StringDataConverter($search);

// build processors
$processor = new MorfoProcessor();

// process collection
$bow = $convertor->getBow();
$processor->process($bow);

// search
$engine = new SearchEngine();
$result = $engine
    ->setOffset($page)
    ->setLimit($max)
    ->searchByBow($bow);

// total number of results
$total = $engine->totalBows();
```

3.3 Aktualizácia kolekcie

Na webových portáloch s rozsiahlou kolekciou dokumentov sa pravidelne pridávajú nové dokumenty, prípadne aktualizujú staré. Nato musíme myslieť a preto si ukážeme, ako aktualizovať databázu našej knižnice tak, aby odpovedala stavu v externom úložišti. Nové dokumenty je potrebné najprv spracovať procesormi. Nato nám slúži rovnaký kód ako pri vytváraní novej kolekcie v sekcii 3.1. Všimnime si v ňom metódu `getBows`, kde je nepovinný argument typu `boolean`.Pomocou neho sa nastavuje, či sa majú naimportovať všetky dokumenty, alebo len nové. Pri aktualizácii teda musí byť nastavený na `true` ako je tomu v ukážke.

Uvedomme si, že metóda tf-idf nie je príliš vhodná pri často aktualizovanej kolekcii. Pridaním nových dokumentov sa totiž zmenia aj váhy slov v starých dokumentoch. Z tohto dôvodu by sme mali všetky váhy prepočítať, čo je pomerne náročná operácia. Avšak drvivá väčšina aktualizácií je výrazne menšinová. Oproti celkovému počtu dokumentov zvyčajne pridávame len zlomkový počet nových. Je nepravdepodobné, že týchto pár nových dokumentov výrazne zmení váhy slov, a preto nie je nutné po každej aktualizácii prepočítavať všetko. Nasledovná ukážka kódu len doplní chýbajúce váhy.

```
$wc = new WeightCalculator();
$wc->completeWeights();
```

Po veľkej aktualizácii, je ale vhodné prepočítať všetky váhy. Malo by sa tak robiť aj po niekoľkých malých aktualizáciách, ktoré spolu pridávajú do kolekcie podstatný počet nových dokumentov. Takisto je vhodné prepočítať všetko

pri zmene starých dokumentov, pretože s nimi predošlá ukážka nijak nemanipuluje. Následne po prepočítaných váhach je vhodné vymazať cache výsledkov hľadania, aby sme nepoužívali staré výsledky. Nasleduje ukážka.

```
$wc = new WeightCalculator();
$wc->updateWeights()
    ->clearCache();
```

4. Výsledky

Vyhodnocovať algoritmus na hľadanie relevantného dokumentu je veľmi problematické kvôli zložitosti ľudského jazyka. Skutočný význam textu môže byť skrytý aj „medzi riadkami“ a dá sa zachytiť len inteligenciou čitateľa. Vyhodnocovať automatizované vyhľadávanie objektívne je až nemožné, pretože nikdy nepreveríme všetky možné kombinácie slov jazyka, ktoré dávajú zmysel. Preto sa úspešnosť automatizovaného vyhľadávania často meria subjektívne na náhodne vybraných dotazoch v obmedzenom počte a vyhodnocuje sa manuálne. My sa predsalen pokúsime utvoriť nejaké závery s použitím rozsiahlej kolekcie článkov v českom jazyku so širokým záberom tém.

4.1 Testovacie dáta

Chceme sa zamerať na testovanie nášho algoritmu na textoch v českom jazyku. Existuje niekoľko pomerne rozsiahlych českých korpusov, ktoré by sme mohli využiť. Problém je, že nie sú voľne dostupné. Avšak tu nám prídu vhod zdroje z českého internetu. Na priloženom disku sa nachádza program napísaný v Jave pre potreby tejto práce, ktorý dokáže prechádzať spravodajský portál idnes.cz a sťahovať z neho články. Pomocou tohto programu sa mi podarilo získať kolekciu o veľkosti 23 922 článkov so širokým záberom tém. Niektoré články boli príliš krátke, preto som použil len články s dĺžkou minimálne 1000 znakov. Máme tak k dispozícii slušnú kolekciu na testovanie.

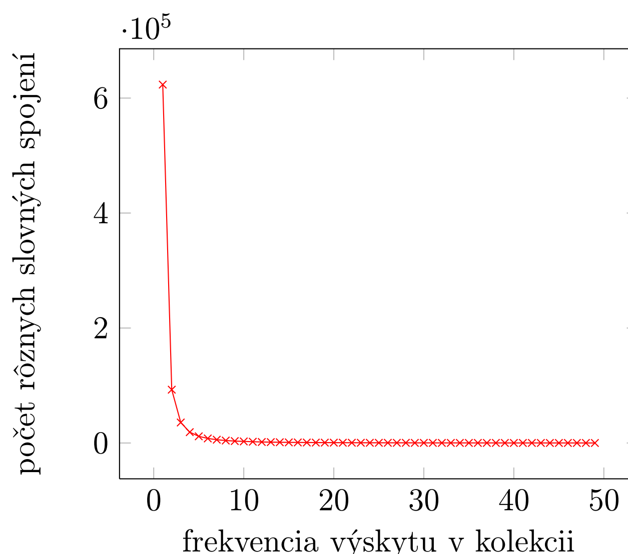
4.2 Metodika merania

Budeme sa snažiť vyvodiť závery o úspešnosti nášho experimentu so slovnými spojeniami, ktorý je popísaný v sekcii 2.3. Experiment pomocou morfologickej analýzy nájde slovné spojenia a tie by mali do vyhľadávania pridať kontext. Ako merať úspešnosť? Urobíme základný predpoklad, že vždy keď slovné spojenia ovplyvnia výsledok hľadania, ovplyvnia ho pozitívne. To je zrejmé, pretože prítomnosť spojenia v dotaze aj v hľadanom článku logicky zvýši ich vzájomnú relevantnosť. Naopak, ak sa spojenie v článku nenachádza, nijak neovplyvní jeho skóre relevantnosti. Po uvedomení si tejto súvislosti je už problém vyhodnotenia úspešnosti jednoduchý. Stačí výsledky hľadania bez použitia spojení porovnať s výsledkami hľadania s použitím spojení. Dostaneme tak počet hľadaní, v ktorých nám slovné spojenia vylepšili výsledok.

Postup bude nasledovný. Zoberieme si 50 000 náhodných slovných spojení z kolekcie a použijeme ich v dvoch druhoch dotazov. Prvý bude hľadať každé slovo spojenia osobitne. Druhý dotaz bude hľadať tieto slová osobitne a navyše bude hľadať aj celé spojenie. Pre jednoduchosť vyberieme pre každý dotaz jeden dokument s najlepším skóre relevantnosti. Zakaždým, keď dostaneme pre dotazy rôzne dokumenty, budeme to považovať za výrazné zlepšenie výsledku vďaka experimentu so slovnými spojeniami.

4.3 Vyhodnotenie a diskusia

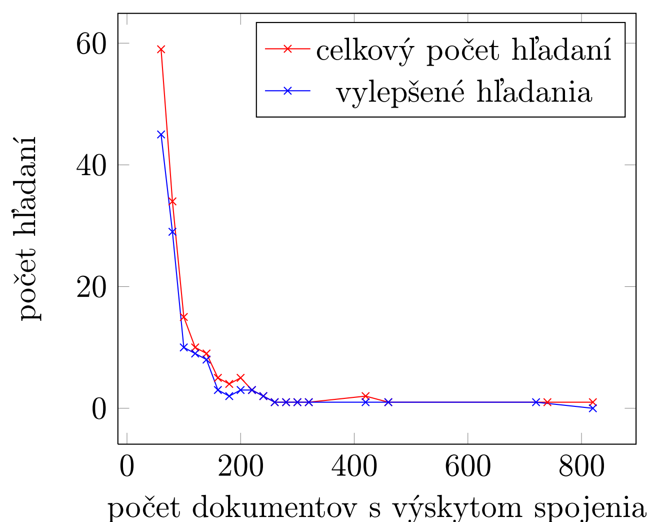
Výsledky ukázali, že na vzorke 50 000 náhodne vyhľadávaných spojení náš experiment vylepšil výsledok pri 38 175 dotazoch. Dosiahli sme tak zlepšenie až u 76.35% dotazoch. Môže sa to zdať až podozrivo pozitívny výsledok, no v skutočnosti sme použili dotazy, u ktorých sme predpokladali potencionálne zlepšenie. Výsledok samozrejme takisto závisí od konkrétnej kolekcie ako aj výberu vzorky spojení.



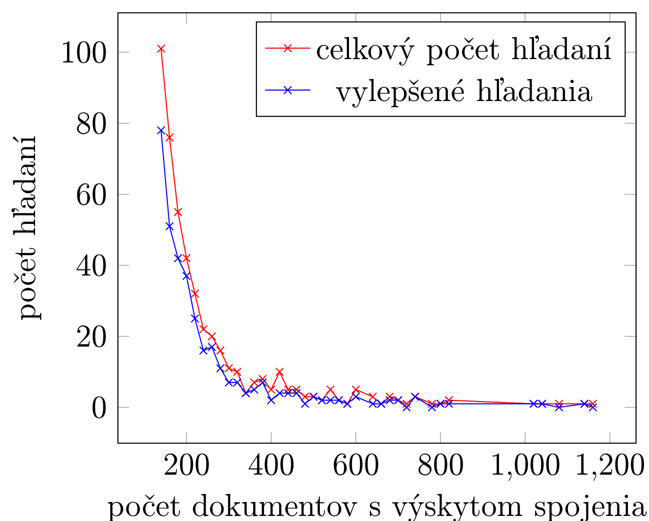
Obr. 4.1: Distribúcia slovných spojení podľa frekvencie výskytu v kolekcii (graf je pre prehľadnosť orezaný zprava)

Keď si preštudujeme graf 4.1 všimneme si, že frekvencia výskytu slovných spojení hyperbolicky klesá. Napríklad až 75.3% slovných spojení sa vyskytuje len v jednom dokumente. Mohli by sme namietať, že slovné spojenia majú obecne extrémne nízku frekvenciu v rámci kolekcie. Preto dostanú vysokú váhu a „zatienia“ ostatné slová. Poďme sa teda pozrieť ako je úspešnosť distribuovaná medzi rôznymi frekvenciami. Rozdistribuovanie náhodne vybranej kolekcie s celkovou úspešnosťou 76.35% je znázornené na obrázku 4.2. Z grafu je zrejmé, že úspešnosť je vysoká pri nízkych ako aj pri vysokých frekvenciách.

Avšak spojenia boli vyberané náhodne a vysoké frekvencie na obrázku 4.2 nie sú reprezentované dostatočne na utvorenie záverov. Preto sa poďme ešte pozrieť na distribúciu úspešnosti u 50 000 najviac frekventovaných slovných spojení, čo vidíme na obrázku 4.3. Vidíme tu zhruba 60-80% zlepšenie u každej frekvencii. Vďaka tomu môžeme uzavrieť, že náš experiment so slovnými spojeniami výrazne zlepšuje výsledky bez ohľadu na frekvenciu slovného spojenia v kolekcii.



Obr. 4.2: Výsledky hľadania ovplyvnené experimentom so slovnými spojeniami pri výbere náhodných spojení (graf je pre prehľadnosť zľava orezaný a počet dokumentov s výskytom spojenia je kumulovaný po 20)



Obr. 4.3: Výsledky hľadania ovplyvnené experimentom so slovnými spojeniami pri výbere najfrekventovanejších spojení (graf je pre prehľadnosť orezaný a počet dokumentov s výskytom spojenia je kumulovaný po 20)

Záver

Podarilo sa mi navrhnúť a naimplementovať knižnicu *RelevantText*, ktorá je univerzálnym nástrojom pre spracovanie rozsiahlych kolekcií textových dokumnetov. V návrhu som vsadil na silne objektový model, čo sa neskôr ukázalo ako výhodná voľba. Umožnilo to efektne oddeliť jednotlivé logické celky knižnice. Každá časť tak rieši svoj problém a je jednoducho rozšíriteľná. To mi umožnilo naimplementovať rôzne nástroje s rovnakým cieľom, z ktorých počas vývoja niektoré nahradili svojich predchodcov.

Prácu som začal na reprezentácii dokumentov pomocou vektorového modelu. Po naimplementovaní som si všimol, že váhy slov neodrážajú ich dôležitosť. Pri študovaní odborných článkov ma zaujala metóda tf-idf, ktorá je pekným zobecnením dôležitosti slova v rámci konkrétnej kolekcie. Rozhodol som sa ju použiť, čo sa pozitívne odrazilo na výsledkoch.

Ďalej som chcel vyhľadávanie textu prispôsobiť konkrétnemu jazyku, čo má veľký zmysel pri jazykoch s bohatou morfológiou. Inšpiroval som sa existujúcim stemmerom pre český jazyk, ktorý som prepísal do PHP. Všimol som si, že orezávanie koncoviek a predpon je príliš primitívny nástroj. Objavil som skvelý program MORFO na morfologickú analýzu češtiny, ktorý som sa hneď rozhodol použiť. Vďaka nemu som dokázal určiť lemma slova s vysokou úspešnosťou, čo plne nahradilo stemmer.

Uvedomil som si, že vektorový model nezohľadnuje vzájomnú vzdialenosť slov v texte dokumentu. Rozmýšľal som ako to zmeniť. Neskôr prišla myšlienka pridávať do vektoru ďalšie rozmery, ktoré budú reprezentovať slovné spojenie. Tu mi poslúžili morfologické značky, ktoré som získaval pri analýze slova programom MORFO. Vedel som určiť, či slová môžu utvoriť gramaticky správne spojenie. V testoch sa to ukázalo ako správna voľba, čo som prezentoval grafom 4.2.

Potreboval som testovaciu kolekciu českých dokumentov, no ako som zistil, tie nie sú ako celok voľne dostupné. Rozhodol som sa preto napísať si vlastný crawler, ktorý mi z českého spravodajského portálu stiahol dostatočne veľkú kolekciu. To mi umožnilo lepšie preveriť pozitívne výsledky mojej práce.

Zoznam použitej literatúry

[1] Segalovich, Ilya. *A fast morphological algorithm with unknown word guessing induced by a dictionary for a web search engine.* Proceedings of MLMTA, Las Vegas, 2003.

[2] Uyar, Ahmet. *Google stemming mechanisms.* In: Journal of Information Science. 2009, č. 5, s. 499–514. ISSN: 0165-5515, 2009.

[3] Porter, Martin. *An algorithm for suffix stripping.* Program 14(3), 1980. 130–137.

[4] Hlaváčová, Jaroslava a Kolovratník, David. *Morfologie češtiny znovu a lépe.* Seňa, Slovensko: PONT s.r.o., 2008. ISBN 978-80-969184-8-5.

[5] Fautsch, Leslie a Savoy Jacques. *Algorithmic Stemmers or Morphological Analysis? An Evaluation.* In: Journal of the American Society for Information Science and Technology. 2009, č. 12, s. 1616–1624. ISSN: 1532-2890.

[6] Grudl, David. *dibi: tiny 'n' smart database layer* [online]. 2012 [cit. 2012-07-02]. Dostupné z: `http://dibiphp.com/`

[7] Kolovratník, David a Přikryl Leoš. *System MORFO for morphological analysis of Czech* [online]. 2012 [cit. 2012-07-02]. Dostupné z: `http://ufal.mff.cuni.cz/morfo/`

[8] Savoy, Jacques. *IR Multilingual Resources at UniNE* [online]. 2012 [cit. 2012-07-02]. Dostupné z: `http://members.unine.ch/jacques.savoy/clef/index.html`

Zoznam použitých termínov

API Application Programming Interface, špecifikácia rozhrania používaného softwarovými komponentami na vzájomnú komunikáciu. 17

hyperoktant d-hyperoktant je jedna z 2^d sekcií d-rozmerného súradnicového priestoru rozdelených podľa d osí tohto priestoru, každá sekcia je jednoznačne definovaná jednou z 2^d rôznych kombinácií znamienok $(\pm, \pm, \ldots, \pm)$, 1-hyperoktant je známy ako priamka, 2-hyperoktant ako kvadrant, 3-hyperoktant ako oktant. 7

inzerent zadávateľ reklamy, v našom prípade fyzická osoba zadávajúca reklamu do reklamného systému. 4

korpus obvykle rozsiahly súbor textov v určitom jazyku, ktoré sú v rôznej miere opatrené metajazykovými značkami a slovami zaradenými do gramatických kategórií. 22

lemmatizér nástroj na určenie základného tvaru slova, k čomu využíva morfologickú analýzu. 8, 11, 16

multimnožina množina, ktorej prvky sa môžu neobmedzene veľa krát opakovať. 12

ORM object-relational mapping – nástroj alebo technika mapovania objektov na jednoduché skalárne hodnoty, ktoré môžu byť trvalo uložené (perzistované) v databáze. 15

stemmer nástroj na určenie základného tvaru slova, ktorý funguje na princípe orezávania suffixov alebo prefixov podľa pevných pravidiel špecifických pre konkrétny jazyk. 2, 8, 16, 25

stopwords slová, ktoré nevypovedajú o význame vety, nie sú dôležité pre kontext. 2–4, 11

UML Unified Modeling Language, štandardizovaný obecný modelovací jazyk, používaný pri objektovo orientovanom vývoji softvéru. 15

Prílohy

Priložený DVD disk obsahuje:

1. knižnicu RelevantText so zdrojovými kódmi
2. API dokumentáciu ku knižnici RelevantText
3. knižnicu dibi vo verzii 1.2
4. demo ukážky portálov s implementáciou knižnice RelevantText
5. tento text práce vo formáte PDF