GYMNASIUM JANA KEPLERA

Parléřova 2/118, 169 00 Praha 6

Komparativní analýza Bayesiánského a klasického přístupu k třídění a vyhledávání informací

Maturitní práce

Autor: Matěj Dvořák

Třída: R8.A

Školní rok: 2020/2021

Předmět: Informatika

Vedoucí práce: Šimon Schierreich

Praha, 2021

GYMNASIUM JANA KEPLERA
Kabinet informatiky

ZADÁNÍ MATURITNÍ PRÁCE

Student:	**Matěj Dvořák**
Třída:	**R8.A**
Školní rok:	**2020/2021**
Platnost zadání:	**30. 9. 2021**
Vedoucí práce:	**Šimon Schierreich**
Název práce:	**Komparativní analýza Bayesiánského a klasického přístupu k třídění a vyhledávání informací**

Pokyny pro vypracování:

Cílem práce je vytvořit systém pro automatickou klasifikaci textu v anglickém jazyce. Systém bude umožňovat a) sestavení vhodného hierarchického systému a jeho ruční obohacování o texty z různých zdrojů, b) analýzu pomocí Bayesiánského klasifikátoru a určení pravděpodobnosti příslušnosti souboru do dané složky a c) jednoduchý crawler vyhledávající stránky spadající do zadané kategorie.

Doporučená literatura:

[1] SRIVASTAVA, Ashok N. a Mehran SAHAMI, ed. Text Mining: Classification, Clustering, and Applications. London: Chapman and Hall/CRC, 2009. ISBN 978-1-4200-5940-3.

[2] RASCHKA, Sebastian. Naive Bayes and Text Classification I - Introduction and Theory [online]. 2017. Dostupné z: https://arxiv.org/abs/1410.5329

URL repozitáře:

https://github.com/xdvom03/klaus

vedoucí práce *student*

V Praze dne 29. 10. 2020

Prohlášení

Prohlašuji, že jsem svou práci vypracoval samostatně a použil jsem pouze prameny a literaturu uvedené v seznamu bibliografických záznamů. Nemám žádné námitky proti zpřístupňování této práce v souladu se zákonem č. 121/2000 Sb. o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon) ve znění pozdějších předpisů.

V Praze dne 7. dubna 2021 Matěj Dvořák

Poděkování

Děkuji vedoucímu práce Šimonu Schierreichovi za poskytnutí materiálů a konzultace. Děkuji spolužákům (Šimon Brecher, Benjamín Verner) za konzultace struktury dat a algoritmů.

Abstrakt

Práce se zabývá tvorbou programu pro zpracování textu pomocí systému Naive Bayes. Tuto myšlenku zobecňuje pro hierarchický systém kategorií. Výsledný klasifikátor využívá pro tvorbu zaměřeného crawleru umožňujícího vyhledávání stránek do dané kategorie. Na závěr práce rozebírá dva příklady spuštění takového crawleru na příkladných datech.

Klíčová slova

zpracování přirozeného jazyka, naivně Bayesiánská klasifikace, webový adresář, sémantické vyhledávání, směrovaný crawler

Abstract

The project is concerned with generalising the Naive Bayes text classification algorithm to a hierarchical class system. The resulting automatic classifier is used to create a focused crawler searching for World Wide Web sites within a sought class. The work concludes with discussing two example runs of such a crawler on example data.

Keywords

natural language processing processing, naive Bayes, web directory, semantic search, focused crawler

Obsah

1. Teoretická část

Před existencí internetového vyhledávače Google byly vyhledávače často neefektivní: nacházely spam a nenacházely relevantní stránky s gramaticky jinými formami hledaných klíčových slov. Proto se na ně uživatelé nespoléhali jako na jediný způsob, jak nacházet informace na internetu. Jednou z alternativ byly webové adresáře (web directories), veřejně dostupné indexy odkazů tříděné podle tématu. Mezi takové adresáře patřilo například Yahoo! Directory nebo DMOZ.

Adresáře byly udržovány ručně na základě podnětů od uživatelů. Tento systém se s růstem počtu dokumentů na internetu ukázal neúnosným, zatímco vyhledávače šlo škálovat. Ve výsledku tak dnes DMOZ ani Yahoo! Directory oficiálně neexistují a dobrovolníky vedené Curlie (vycházející z DMOZ)[1] je těžce zastaralé, značná část odkazů už ani nefunguje. Výsledkem je dnešní stav, kde má na vyhledávání téměř monopol Google.

Vyhledávače na základě klíčových slov ale také nejsou ideální. Dobře si poradí s konkrétními požadavky (např. chybová hláška nebo název restaurace). Necháme-li stranou problematiku jejich monetizace a SEO, stále trpí omezenou schopností pracovat do hloubky se širokým dotazem. Požadavku o několika slovech odpovídá řada dokumentů, výběr z nich ale probíhá arbitrárně nebo na základě řady faktorů, jako jsou stáří odkazu, další klíčová slova, nebo informace o vyhledávajícím uživateli (např. lokace). Tyto faktory jsou těžko ovlivnitelné, prozkoumat vyhledávaný široký dotaz ze všech úhlů pohledu proto obvykle nejde. Někdy navíc uživatel nemusí správné klíčové slovo znát.

Adresáře místo toho fungují na základě tématických okruhů. Je-li adresář dobře veden, každý okruh obsahuje řadu podokruhů korespondujících k různým podtématům. Adresář tak slouží jako encyklopedie a počáteční bod pro ruční vyhledávání informací. Je tak kromě vyhledávače i pedagogickým nástrojem.

Problém s udržováním adresářů je ale stále platný. Tento projekt si klade za cíl jej částečně automatizovat - vyhledávat texty podobné již přidaným stránkám. Uživatel, manažer adresáře, by pak musel jen adresář prohlubovat a složky dále dělit.

1.1 Existující technologie

Zpracování přirozeného jazyka pomocí metody Naive Bayes má široké uplatnění, spíše je ale využíváno pro konkrétnější účely, jako klasifikace uživatelských recenzí[2]. Existuje projekt Klassify[3], umožňující obecně natrénovat klasifikátor, ale jde spíše o demonstraci a chybí základní možnosti úprav databáze, mazání/přesouvání souborů, apod. Našel jsem jeden pokus o obecnou klasifikaci[4], vychází ale z knihovnického systému, který není dobře přenositelný na internetové stránky. Ukázala se na něm ale platnost principu.

[1] https://curlie.org/
[2] https://arxiv.org/pdf/1610.09982.pdf
[3] https://github.com/fatiherikli/klassify
[4] https://www.researchgate.net/publication/4046602_Classification_of_Web_Documents_Using_a_Naive_Bayes_Method

2. Implementační část

2.1 Klasifikace

Program funguje na základě statistické metody Naive Bayes, kterou se řídí při rozhodování, kam daný dokument zařadit. Nejjednodušším případem je rozhodování mezi dvěma kategoriemi.

2.1.1 Dvě kategorie

Dokument modelujeme jako seznam slov v něm obsažených (viz část Tokenizace). U kategorií vybudujeme součtem pro všechny dokumenty v nich obsažené korpus, v němž je uveden počet výskytů každého slova. Při porovnávání dvou kategorií pracujeme s normalizovanými korpusy. U většího z korpusů podělíme výskyty všech slov stejnou konstantou tak, aby měly oba výsledné korpusy dohromady stejný počet slov. U každého slova určíme počet jeho výskytů v obou korpusech. Označme výskyt slova v kategorii A a, v kategorii B b. Pravděpodobnost pro dané slovo, že dokument jej obsahující patří do kategorie A, určíme jako:

$$P = \frac{a + k}{a + b + 2k}$$

Ve výpočtu zahrnujeme konstantu k, takzvaný "smoothing parameter", podstatnou zejména v případě, že a nebo b je velmi malé. Program využívá konzervativní Laplace smoothing, kde $k = 1$ (Lidstone smoothing, nižší hodnoty k, by v takových případech vedlo k vyšší váze takových slov). Takto určíme pravděpodobnosti pro všechna slova v dokumentu. Dále vybereme relevantní klíčová slova.

Nevyužíváme všechna slova v dokumentu, protože bychom mohli zesílit šum. Pro každou z kategorií uvažujeme jen X nejsilnějších slov proti kategorii, kde X je polovina počtu slov nad stanovenou "zajímavostí"(konkrétně slova s pravděpodobností pro jednu z kategorií pod 0.2). X má nastavenu minimální hodnotu, takže vždy uvažujeme alespoň 12 slov, ale většinou je zajímavých slov více. Smysl tohoto výběru je snaha o nalezení stejného množství důkazů proti oběma stranám a porovnání jejich celkové síly.

Skóre pro tato klíčová slova vynásobíme. Tento proces provedeme stejně pro kategorie v opačném pořadí (vyměníme a a b). Výsledné pravděpodobnosti pro obě kategorie vynásobíme společnou konstantou tak, aby se sečetly na 1 (tomuto dále v textu budeme říkat "normalizace pravděpodobnosti", NP). Tento krok je primárně řízen přehledností a snahou vytvořit čisté pravděpodobnosti, ne dvě velmi nízká skóre.

2.1.2 Zobecnění pro N kategorií

Problém zařazení do jedné z několika kategorií má několik řešení. Při volbě modelu jsem se zejména řídil požadavkem, aby obsah jedné kategorie neovlivnil rozhodování mezi ostatními kategoriemi. Proto například nevolím metodu, kde se každá kategorie srovná s celým vzorkem a vyhraje ta s

nejlepším výsledkem[1]. Každá kategorie by byla v takovém případě součástí všech rozhodování. Pokud by například jedna z kategorií byla "úplně mimo", všechna ostatní skóre by mohla vyjít poblíž jedné a rozlišovací schopnost by se ztratila. Toto je případ např. nejvyšší úrovně přiloženého rozdělení, kde by takto působily legální dokumenty[2].

Namísto toho začínám nalezením pravděpodobností pro každou dvojici kategorií. Dále vycházím z předpokladu, že správná kategorie by měla porazit všechny ostatní. Jakákoli kategorie tedy může být sama o sobě kontrolní pro jakoukoli jinou. U každé kategorie získáme výslednou pravděpodobnost příslušnosti vynásobením pravděpodobností proti všem ostatním kategoriím. Pro prezentaci provedeme NP všech výsledků.

Obrázek 2.1: Vysvětlení klasifikace

	articles	fiction	legal	news	non-english	utility	
articles		0.0	-237.4	0.0	0.0	-0.065	-237.465
fiction	-321.719		-719.083	-156.049	0.0	-257.189	-1454.04
legal	0.0	0.0		0.0	0.0	0.0	0.0
news	-44.469	0.0	-404.777		0.0	-65.507	-514.753
non-english	-735.722	-480.616	-1171.384	-607.362		-705.945	-3701.029
utility	-2.769	0.0	-222.496	0.0	0.0		-225.265

Na této ukázce je vidět klasifikace Wikipedia Terms of Use. Čísla v tabulce jsou logaritmy jednotlivých pravděpodobností, čísla vpravo od tabulky jejich součty. Jak je vidět, kategorie *legal* porazila všechny ostatní kategorie a jejich vzájemná skóre už nejsou podstatná.

Další výhodou tohoto systému je fakt, že výstupem je stále pravděpodobnost, nikoli pouze pořadí.

Výstup vícekategorické klasifikace

Pravděpodobnost klasifikace do dané kategorie vrací funkce (prob). Na vyšší úrovni je funkce (place), jež najde nejpravděpodobnější konečné umístění stránky. To získáme klasifikaci úroveň po úrovni. Na každé úrovni sledujeme nejlepší možnost, pokud jí přísluší pravděpodobnost překračující danou mezní hodnotu. Zde ale nevyužíváme NP. Logika je taková, že pokud je NP podstatný efekt, znamená to, že všechny kategorie mají malé skóre, což znamená, že proti všem kategoriím existují důkazy. To může nastat v případě, že je klasifikace založená na mylném předpokladu: stránka pravděpodobně nepatří do žádné z možností. To může mít tři hlavní důvody: Buď nepatří do kategorie, v rámci níž klasifikujeme (tj. někde předtím nastala chyba v klasifikaci), nebo daná podkategorie neexistuje, nebo nepatří do podkategorie, ale jen do celé kategorie.

[1] Tento předpoklad s vznešeným názvem *Maximum-likelihood estimate* je ve většině zdrojů, např. v https://arxiv.org/pdf/1410.5329.pdf, nebo v https://web.archive.org/web/20200708084909/https://web.stanford.edu/class/cs124/lec/naivebayes.pdf (slide 55). Pokud vím, mou alternativu nikdo jiný zatím netestoval, možná také proto, že její časová náročnost roste s druhou mocninou počtu kategorií, takže pro více kategorií prakticky vynucuje hierarchický systém.

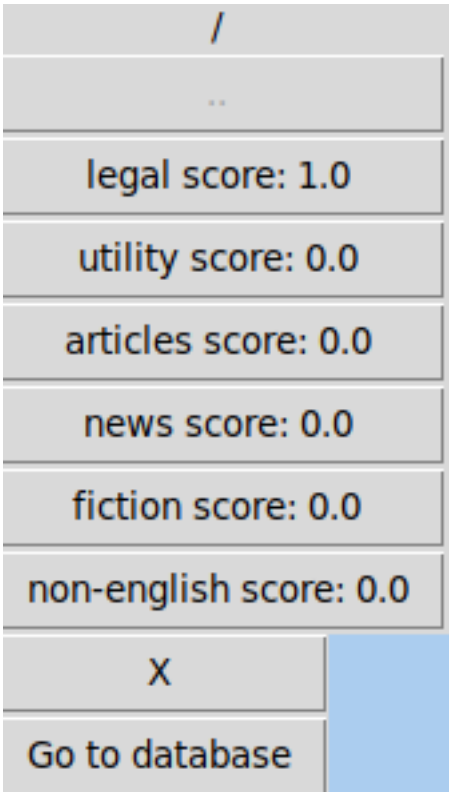
[2] Jednou z prvních kategorií, kterou jsem testoval, jsou dokumenty typu "Terms of Use"a "Privacy Policy", pro jejich snadnou odlišitelnost (a velký výskyt, pro crawler jsou užitečné, aby se jich dokázal vyvarovat)

Logartimický formát čísel

Ježto se s dostatkem slov můžeme dostat na pravděpodobnosti velice blízké nule, hrozí ztráta přesnosti, v horším případě zaokrouhlení na nulu. Proto všude namísto s pravděpodobnostmi počítáme s jejich přirozenými logaritmy. Ty také zobrazujeme ve vysvětlovacím okně (viz manuál).

2.1.3 Hierarchický systém

Obrázek 2.2: Možnost výběru další úrovně



Kategorie může obsahovat podkategorie. Soubory mohou obsahovat všechny kategorie, nejen ty na poslední úrovni. Při rozhodování se berou v úvahu všechny dokumenty v kategorii a všech úrovních jejích podkategorií. Klasifikace pak může být vícekroková. Na obrázku je vidět okno při vysvětlení klasifikace, kde je možné kliknutím na příslušné tlačítko pokračovat v klasifikaci v libovolné podkategorii. Další části programu využívají funkci place, která pro daný text najde nejpravděpodobnější umístění (v každém kroku se dá cestou s nejvyšší pravděpodobností).

2.1.4 Benchmark klasifikace

Tento systém jsem testoval na často užívaném datasetu 20 Newsgroups[3]. Hierarchie jsem sestavil dle doporučení autora datasetu[4]. Výsledná přesnost byla 6093 z 7528 dokumentů klasifikovaných správně, tj. zhruba 81%, což je srovnatelné s alternativními technologiemi. Naive Bayes s dodatečnými "vylepšeními"jako stemming (zjednodušení na základní gramatické tvary slov) a odstranění

[3]https://ana.cachopo.org/datasets-for-single-label-text-categorization
[4]http://qwone.com/ jason/20Newsgroups/

stop words (velmi častých slov bez významového obsahu) získal (dle prvního zdroje) výsledek 6100 z 7528, z čehož usuzuji, že efekt těchto technlogií je zanedbatelný v rámci chyby měření. K podobnému výsledku dochází i většina literatury, např. zde[5] dochází autoři k výsledku, že v angličtině různé lemmatizační algoritmy vedou k menšímu korpusu, ale vždy za cenu poklesu efektivity. Na škále, na které pracuje tento projekt, nemá zmenšování korpusu prioritu.

2.2 Crawler

Crawler vychází z předpokladu, že stránky budou spíše odkazovat na stránky s podobným tématem. Snaží se tedy stále sledovat co nejlépe odpovídající stránky a odkazy z nich. Pro specifické požadavky na několikáté úrovni klasifikace to činí tak, že (obvykle) najde nejprve obecnější požadavek a až z něj požadavek konkrétní. Například pro /news/sports/ se pravděpodobně nejprve dostane crawler na stránku novin a až poté najde sportovní sekci. Tomuto algoritmu by měl být přizpůsoben hierarchický systém.

2.2.1 robots.txt

Bot splňuje robots.txt podle původní specifikace[6]. Je identifikován v hlavičce požadavku na stránku, kde je také uvedena adresa projektu a má emailová adresa pro případ, že by někde způsoboval problémy. Na stránky neposílá přehnaně mnoho požadavků - teoretické maximum během jednoho crawlu je zhruba 20*(počet stránek na doménu) a vzhledem ke klasifikaci je posílá postupně a se zdržením.

2.2.2 Skóre

Crawler má za cíl najít stránky v dané kategorii, vychází tedy ze skóre určujícího, jak moc se daná stránka cílové kategorii přiblížila. Skóre vychází z nejpravděpodobnějšího umístění kategorie. Je součtem počtu úrovní (od počáteční), ve kterých se shoduje cílové a skutečné umístění, a pravděpodobnosti pro první neshodující se kategorii. Například: hledá-li crawler /news/sports/tennis/ a najde /news/news-reports/, je skóre 1 plus pravděpodobnost, že namísto do /news/news-reports/ patří stránka do /news/sports/.

2.2.3 Algoritmus crawleru

Vstupem je počáteční URL adresa a počet kroků. Algoritmus crawleru:

1. Ze zatím prošlých URL vybereme to s nejlepším skóre, které jsme zatím takto nevybrali (tj. po vybrání odstraníme URL z možností výběru). 2. Najdeme všechny odkazy vedoucí z tohoto URL.

[5]Toman, Michal & Tesar, Roman & Jezek, Karel. (2006). Influence of Word Normalization on Text Classification.

[6]http://www.robotstxt.org/orig.html

Odstraníme duplikátní odkazy[7]. 3. Vyloučíme všechny nefunkční odkazy[8] a všechny odkazy vedoucí na již nám známé domény. 4. Stáhneme všechny zbylé cílové stránky, oskórujeme. Vybereme z nich tu s nejlepším skóre. Skóre měříme v náhodném pořadí, je-li odkazů moc, můžeme po daném počtu přestat. 5. Na nové doméně několikrát (počet je nastavitelný) sledujeme odkazy zůstávající uvnitř domény, vždy opět na ten s nejlepším skóre. Všechna URL po cestě ukládáme do seznamu pro bod 1. 6. Opakujeme kroky 1-5 tolikrát, kolik domén je specifikováno projít.

Pokud v kroku 4 nemáme žádné funkční odkazy, doménu pouze odstraníme a zkusíme jinou. Pokud dojdou domény, systém zavolá chybu, která se uživateli zobrazí [!]. Pokud nenajdeme funkční odkazy v kroku 5, zkoušíme to znovu, ale neplatný pokus počítáme jako pokus (abychom se mohli dostat pryč z domény, která neodkazuje na sebe samu).

Při výpočtu skóre automaticky uložíme nalezené umístění URL do dat, která periodicky ukládáme do souboru /DATA/discovered. Tento soubor pak můžeme procházet [tady doplnit až bude funkční GUI].

2.3 Zdrojová data

2.3.1 Tokenizace

Většina dokumentů je získána stažením HTML z internetu. V takovém případě nejprve odstraníme všechen text v tazích <script> nebo <style> a následně všechny HTML tagy. Dále text zbavíme diakritiky a všech ne-alfanumerických znaků. Všechen odstraněný text nahrazujeme mezerami. Výsledek rozdělíme na tokeny podle mezer.

2.3.2 Opakované tokeny

Během vývoje jsem zkoušel dva extrémní způsoby řešení opakování tokenů. Buď' opakování nepovažujeme za problém a tokeny počítáme vícekrát, nebo počítáme každý token pouze jednou. V prvním případě může hromadné opakování tokenu na jedné stránce pokřivit vnímání tokenu (což se například stalo s malými čísly, která disproporčně naznačovala matematiku, když přitom může jít např. o nadpisy kapitol), ve druhém případě nefungují častá slova ve dlouhých textech (jako je například kategorie fiction). Proto text rozdělujeme na úseky o dané délce (několika stovek slov), ve kterých tokeny neopakujeme, mezi nimi ale ano. [až bude vyřešené #89, tak detaily o vysvětlovači]

2.3.3 Boilerplate

Na doméně se často nachází text nesouvisející s konkrétní stránkou, určený např. pro navigaci na stránce, nebo pro splnění legálních požadavků (disclaimer psaný malým písmem ve spodní části

[7] Za duplikátní považujeme odkazy lišící se jen absencí www, rozdílem http/s, velikostí písmen v doméně, fragmentem (částí za # v odkazu). Absence www může znamenat jinou stránku, v praxi to tak ale v převážné většině případů není.

[8] Odkazy na nefunkční nebo nedostupné stránky, příliš krátké stránky, ale také stránky zakazující funkci crawleru souborem robots.txt

stránky). Takový text se snažíme u trénovacích dat odfiltrovat, aby neovlivňoval povahu kategorie. Proto u každého dokumentu poté, co ho stáhneme, během výstavby dat nejprve vytvoříme jeho verzi s co největší částí takového textu odstraněnou. Pro každou doménu v systému, od které máme alespoň dva soubory, nalezneme překryv všech jejich textů. Překryvem je jakýkoli úsek o daném minimálním počtu slov, který se vyskytuje ve všech souborech. Takové úseky pak ze souborů odstraníme, čímž vzniknou texty očištěné, které ukládáme ve složce DATA/core/.

2.4 Uživatelské prostředí

Grafické prostředí je limitováno schopnostmi knihovny LTK, v rámci nichž funguje analogicky k procházení souborového systému. Ježto nemusí uživateli zobrazovat dokumenty ani velké množství kategorií najednou, je na obrazovce hodně volného místa, takže jsou všechny možnosti zobrazeny najednou namísto ukrytí do submenu. [až bude opravené #94, tak screenshot upravovátoru databáze].

Hlavní netradiční částí UI je systém přesouvání souborů a kategorií. Ježto v knihovně není jednoduché vytvořit click-and-drag, systém běžně používaný pro přesouvání, inspiroval jsem se na C2 Wiki[9] a využil systém interně nazývaný Bucket: soubory nebo složky může uživatel zkopírovat do části Bucket, ve které zůstávají při pohybu skrz souborový systém, a ze které mohou být přesunuty do jiného místa. Umožňuje to jednodušší přesuny mezi vzdálenými kategoriemi. Podobně přesouváme i celé kategorie. Tlačítka s módy umožňují výběr mezi přesunem, smazáním, nebo jen odstraněním z části Bucket. Pro odlišení od tlačítek provádějících nějakou akci je na nich text psán velkými písmeny.

2.5 Přiložená klasifikace

Souběžně s vývojem prostředí jsem vyvíjel i vlastní soubor trénovacích dat kvůli testování systému. Mimo jiné jsem díky němu zjistil, že užitečnější systém uvažování, než ontologie (ukládání textů podle toho, co *jsou*), je ukládání podle principu vyhledávacího (požadavek, pro který by nalezení daného dokumentu dávalo smysl). Tím se vyhneme často filozofickým otázkám o tom, co je *doopravdy* např. vzdělávání.

2.6 Design

2.6.1 Použité technologie

Jazyk

Program jsem se rozhodl vytvořit v jazyce Common Lisp. Roli při výběru hrály: kombinace funkcionálního a imperativního paradigmatu, rychlost výpočtů, jednoduchá prefixová syntaxe vhodná pro zápis matematických vzorců na více řádků (což usnadňuje práci s výpočty), předchozí zkušenost

[9] http://wiki.c2.com/?HandVsPointer

se souvisejícími problémy v tomto jazyce. Klasifikace je naprogramována převážně funkcionálně, crawler převážně imperativně. Uživatelské rozhraní je řízeno na bázi událostí (stisknutí tlačítek, zavření okna).

Knihovny

LTK[10]: Grafické rozhraní. Jde o port Tcl/Tk pro Common Lisp a koncepčně nejjednodušší řešení, ale některé funkce v knihovně chybí (zejména volba barvy tlačítka a okno pro výběr souboru - které jsem ale vytvořil vlastní a snad funguje).

Drakma[11]: Stahování HTML. Nejprve jsem používal knihovnu Dexador[12], rozšíření Drakma, ale ukázalo se, že trpí chybou, která způsobuje únik paměti. Tuto chybu jsem nahlásil[13], ale knihovna není udržovaná a nepřišla žádná reakce.

QURI[14] umožňuje jednak úpravy znaků v názvech (např. example.com/B%C3%A1g%C5%99i -> example.com/Bágři), jednak práci s odkazy - najde cíl odkazu i pro řadu speciálních případů (kromě pasti, kterou je dvojí lomítko pro relativní protokol) a dokáže obecně najít klíčovou část domény pro odkazy (mimo jiné pozná, že .co.uk je jen jedna koncovka).

Plump[15] dokáže rozklíčovat escaped znaky v HTML (’ -> ’).

Trivial-timeout[16] zastaví načítání nereagující stránky.

Cl-strings[17] obsahuje řadu utilit pro práci s textem.

Alexandria[18] umožňuje kopírování struktury hash table.

2.6.2 Ukládání dat

Program má tři druhy dat: Korpusy, konfigurační data ke složkám a zdrojové dokumenty. Zdrojové dokumenty jsou z nich zdaleka největší, proto je každý dokument uložen v samostatném souboru. Při vložení souboru se vytvoří jeho tři verze - hrubý text tak, jak byl vložen, text očištěný dle tokenizace a text s odstraněnými duplikátními prvky (viz Boilerplate). Tyto zpracované verze textu by bylo lze vytvořit znovu, např. v případě změny povolených znaků.

Ostatní data nepřesahují pro typické používání jednotky MB, jsou tedy uloženy pro všechny kategorie dohromady v jednom souboru. To má výhodu jednoduchého exportu, viz manuál.

[10] http://www.peter-herth.de/ltk/index.html
[11] https://github.com/edicl/drakma
[12] https://github.com/fukamachi/dexador
[13] https://github.com/fukamachi/dexador/issues/97
[14] https://github.com/fukamachi/quri
[15] https://github.com/Shinmera/plump
[16] https://common-lisp.net/project/trivial-timeout/
[17] https://github.com/diogoalexandrefranco/cl-strings
[18] https://gitlab.common-lisp.net/alexandria/alexandria

2.6.3 Databáze?

Data program při spuštění načte ze souborů a opět je do nich uloží, pokud uživatel uloží změny. Jinak má celý korpus v pracovní paměti, což zrychluje klasifikaci.

V současném stavu má korpus 8.1 MB. Obsahuje kolem 100000 různých slov a 200 kategorií. Počet různých slov už řádově nevzroste, např. Oxford English Dictionary obsahuje kolem 600000 slov[19], ale řada z nich je reálně nevyužívaná. Počet kategorií může růst s vývojem. Spíše ale porostou počty slov, což na velikost dat prakticky nebude mít vliv. Na problémy by mohl tento systém narazit při řádově větším počtu kategorií.

2.6.4 Rychlost

Úpravy databáze

Program se snažíme zrychlit zejména dvěma technikami: prací v paměti, kdy omezujeme práci se soubory na načtení a uložení, a omezením redundantní práce. U časově náročných činností, tedy odstranění boilerplate a výstavba korpusu, ukládáme vstupní data, ze kterých vznikla předchozí verze takto zpracovaných dat. Pokud zjistíme, že uživatel nezpůsobil žádné změny, necháme data tak, jak jsou. Tím udržujeme prostředí pro úpravu databáze použitelným.

Klasifikace

Většinu doby běhu klasifikace stráví program požadavky na vyhledání v hash table s počty slov v jednotlivých korpusech. Nic výrazně rychlejšího, než hash table, už ale nevymyslíme a všechny požadavky jsou nutné pro výpočet skóre. Teoreticky by se dal program zrychlit předpočítáním všech skóre všech slov, ale prakticky by to nepřiměřeně zpomalilo tvorbu dat.

Crawler

Crawler na rozdíl od práce s databází data stránek neukládá. S HTML pracuje v paměti a zahodí jej, jakmile už není potřeba. Dále má cache všech pravidel robots.txt, na která cestou narazil (ne souborů robots.txt, ale už zpracovaných pravidel), aby je nemusel stahovat a zpracovávat stále dokola. Ve výsledku tvoří většinu času běhu crawleru stahování dat ze stránek, což na straně programu vylepšit nemůžeme.

2.6.5 Slepé uličky

Nejdynamičtější byl vývoj uživatelského prostředí, které nejdřív mělo řadu zbytečných funkcí, jako historie vložených URL, a do databáze se musel uživatel prokliknout s jedním URL pro jeho zařazení (takže přidávání víc URL vyžadovalo pokaždé znovu najít cílovou kategorii).

[19] https://www.oed.com/

Skórování mezi více kategoriemi nejdřív fungovalo na bázi algoritmu podobného PageRanku, kde se váha každé kategorie jako důkazu proti ostatním kategoriím určovala na základě výsledného skóre dané kategorie. Do věci to vnášelo složité další výpočty, které neměly dobré logické opodstatnění (na rozdíl od samotného PageRanku, který koresponduje s určitým modelem chování uživatele).

Hodně změn nastalo v souborovém systému. Před současným uložením do jednoho souboru jsem metaforu souborového systému v klasifikaci zakládal na skutečném souborovém systému, takže kategoriím odpovídaly složky. To bylo časem zbytečně nepřehledné.

Zbytečně dlouho jsem razil myšlenku informační konzole v GUI, ve které by šlo vidět většinu činností programu. Oproti běžné textové konzoli neměla žádné výhody (chtěl jsem ji rozdělit podle skupin oznámení, ale nikdy jsem to neudělal) a uživatelské prostředí značně komplikovala.

Prvotní inspirace programu pochází z Bayesiánského spamového filtru, který vyvinul Paul Graham[20]. Chtěl jsem jeho myšlenku zobecnit. Bohužel, detekce spamu je jako klasifikace textu značně specifický problém. Graham se v původním filtru dopustil řady zjednodušení, která fungovala konkrétně pro spam (nebo alespoň neškodila, většina spamu je odlišitelná snadno, tedy i s částečně chybným systémem). Musel jsem jednu po druhé vložit řadu oprav: kontrolu pro počet dokumentů v kategorii, kontrolu pro celkový počet slov v kategorii, smoothing, balancovaný výběr slov. Některé z těchto chyb se vzájemně vyrušují, takže se program prakticky od září až do prosince, kdy padla ta poslední, choval většinou absurdně.

Crawler měl řadu verzí, které uvažovaly pouze omezenou frontu možností, řazenou podle skóre, s různými variacemi. Výsledkem typicky bylo, že všechny možnosti měly nulové skóre a crawler se náhodně motal, protože když něco náhodou našel, nedal do fronty dost různých odkazů, takže mu brzy zase došly. Samotné skóre také mělo řadu pochybných variant, například prostě pravděpodobnost, že patří stránka přesně do dané kategorie (bez bodů za částečný překryv), což prakticky znamenalo, že pokud nenašel crawler přesně to, co hledal, všechny možnosti byly blízko nuly a rozhodoval spíše počet slov, podle kterých se klasifikace mohla řídit.

2.6.6 Možná vylepšení

Technická vylepšení

U stránek pouze stahujeme HTML, což se může někdy značně lišit od toho, co by viděl uživatel. Nejdestruktivnější případy (kdy celou stránku řídí Javascript a není na ní vůbec statický text) dokážeme odchytit varováním o krátkém textu. Lepší by ale bylo využít například program Selenium[21] pro napodobení webového prohlížeče. Na přiloženém korpusu se to projevuje mimo jiné nesprávným zpracováním stránek, na nichž je LaTeX. Dále program nevyužívá některých částí HTML standardu, zejména tagu <base> v hlavičce, což někde může způsobit, že crawler nedokáže sledovat odkazy.

Kategorizaci by mohla vylepšit opatrná verze technologie stemming.

[20] http://paulgraham.com/spam.html
[21] https://www.selenium.dev/

Další schopnosti

U crawleru by se hodilo zobrazit více statistik o jeho cestě, možnost např. stromového zobrazení, ze kterých stránek se kam dostal.

3. Technická dokumentace

3.1 Požadavky spuštění

Pro GUI je nutný grafický systém Wish z Tk. Instalovat lze např. pomocí APT: "sudo apt install tk". Pro kompilaci ze zdroje je také potřeba stejně instalovat kompilátor SBCL.

Na systému Windows je nutno instalovat Tcl/Tk, např. IronTcl[1], a před spuštěním (main) zavolat (setf ltk:*wish-pathname* "cesta

k

souboru

wish") - pozor na potřebu dvou zpětných lomítek (escaped backslash).

3.2 Release

Nejjednodušší cestou pro spuštění programu je rozbalit jeden ze souborů ve složce /releases/ (značeny jsou dnem vytvoření) a spustit spustitelný soubor v rozbalené složce. Přiložená data jsou vedena jako importy, oproti kompilaci ze souboru tedy není nutno nic stahovat.

Release jsem testoval na nově instalovaném systému Ubuntu 20.04 LTS ve Virtual Machine, kde ho šlo spustit a používat jen s požadavky výše uvedenými.

Bohužel se mi nepodařilo vytvořit funkční release pro systém Windows, ale kompilace ze zdrojového kódu alespoň na Windows 7 funguje.

3.3 Kompilace ze zdrojového kódu

Kvůli knihovnám je nutné nainstalovat manager knihoven pro Common Lisp, Quicklisp, tak, aby existoval soubor /portacle/all/quicklisp/setup.lisp. Doporučuji stáhnout systém Portacle[2] a sloučit složky portacle. Spusťte buďto instalované Portacle, nebo SBCL ve složce /portacle/. Pak stačí zkompilovat soubor portacle/load.lisp a do REPL[3] zadat (main). Po chvíli načítání se objeví okno grafického rozhraní.

[1] https://www.irontcl.com/
[2] https://portacle.github.io
[3] Read Eval Print Loop, konzole pro Common Lisp

3.4 Navigace

Po spuštění se uživatel dostane do klasifikačního okna, kde může zjistit klasifikaci vloženého dokumentu, případně si ji nechat vysvětlit (zobrazit pravděpodobnostní tabulky a klíčová slova).

Druhé prostředí umožňuje úpravy databáze trénovacích dat. Uživatel se může pohybovat mezi jednotlivými kategoriemi, zobrazit v nich obsažené soubory, komentáře k hierarchickému systému, procházet korpus. Může soubory přidávat (buď pomocí URL odkazu, nebo vepsáním, nebo vložením souboru). Může také přesouvat, přejmenovávat, a mazat kategorie. Vše je uloženo najednou tlačítkem "uložit změny"(Save Changes), při zavření okna vyskočí možnost uložit případné neuložené změny.

Třetí prostředí umožňuje spuštění crawleru a prezentaci jeho výsledků (až bude #23).

Mezi prostředími lze vždy přejít tlačítkem ve spodní části okna [až bude uklizené tlačítko, kam patří].

3.5 Klasifikační GUI

Závěr

Zadáním práce bylo vytvořit jednotný systém umožňující tři základní funkce:

3.6 Tvorba hierarchického systému

Soubory lze vkládat z různých zdrojů, ačkoliv v praxi jsem manuální vkládání neuplatnil, protože není vhodné pro věrný popis internetových stránek.

3.7 Klasifikace daného textu

Systém umožňuje jak přímo zjistit umístění textu, tak získat podrobná data o důvodech, z jakých byl text umístěn, kam byl.

3.8 Crawler

Myšlenku crawleru jsem v zadání úmyslně zlehčil ("jednoduchý crawler"), protože jsem neměl předem představu o tom, jak dobře může crawler dopadnout. Formálně vzato systém crawleru existuje, zajímavější je ale otázka, jestli je efektivní v praktickém účelu.

3.8.1 Test crawleru

Abych to zjistil, nechal jsem crawler běžet na základě stejných dat za stejných podmínek ze stejné počáteční stránky s dvěma různými cíli. Počáteční data jsou v release "5-4-2021", počáteční stránkou je "https://en.wikipedia.org", crawler hledá 160 domén, 4 stránky na doménu.

První crawl

Skóre stránek, na kterých cestou byl, je rozloženo takto:

Crawl trval 6 hodin a 50 minut. 17 nalezených stránek bylo duplikátních, jen s rozdílným query v adrese, což obecně může znamenat zcela jiný obsah, ačkoliv zde šlo o irelevantní požadavky.

Počet správných úrovní	0	1	2	3	4	5
Počet stránek	130	20	225	137	33	10

Tabulka 3.1: Skóre s cílem /articles/stem/nature/physics/thermodynamics/

Počet správných úrovní	0	1	2	3	4	5
Počet stránek	178	92	52	11	4	245 .

Tabulka 3.2: Skóre s cílem /articles/life/recreation/art/fiction/

V tabulce je vidět, že se crawler většinu času dokázal udržet kolem /articles/stem/nature/, ale pak měl problémy. Z toho vyplývá, že rozdělení nature je špatně navrženo. S tím nelze nesouhlasit, protože kategorie nature obsahuje pouze podkategorie space a physics. Chybí jí vlastní povaha, jde jen o nádobu pro dvě jiné kategorie. Pro další využití jde o signál, že tato část potřebuje jiné řešení, které se také v posledním release nachází.

Rozložení všech cestou ohodnocených stránek:

V příloze A je seznam všech stránek a jejich skóre, aby si mohl čtenář dokumentace udělat představu o výsledcích. Crawl trval 6 hodin a 50 minut.

Druhý crawl

Crawl trval 9 hodin.

38 stránek bylo duplikátních. Tento crawl byl mnohem úspěšnější, až 42 % prošlých stránek bylo podle crawleru relevantní. Je to také proto, že jde o mnohem širší kategorii a zároveň o kategorii šířeji zastoupenou v osobních stránkách, kterých je na internetu nevyčerpatelná zásoba. Naznačuje to, že by se měla tato kategorie prohloubit a zaplnit různými podkategoriemi, které se zde nabízejí podle žánrů[4] nebo podle typu obsahu (recenze, fanpage, diskuze o spisovatelství...). Všechny výsledky jsou v příloze B.

3.9 Shrnutí

Nejobecnějším ponaučením z práce na projektu je obrovská užitečnost databáze issues, do které jsem ukládal všechny chyby a všechna plánovaná vylepšení programu. Github umožňuje issues uzavřít a kdykoli se k nim vrátit, což je užitečné, pokud se chyba vrátí v jiném kontextu. Seznam také pomohl při tvorbě dokumentace.

[4] I přesto, že trénovací data byla pouze literární, sem spadla řada filmů. To vzhledem k obecnější kategorii fikce nevadí, ale je to zajímavé zobecnění.

Seznam obrázků

Seznam tabulek

4. Nalezené stránky, crawl 1

text

5. Nalezené stránky, crawl 2

text