FIT ČVUT, BI-VWM

Projektová dokumentace

Rozšířený Boolovský Model

Lukáš Rynt, Martin Šír
1.1.2021

Obsah

1. Popis projektu

Cílem projektu je vytvořit implementaci rozšířeného boolovského modelu ukládání dat (tj. preprocessing a indexování) spolu s možností dotazování z GUI.

Vstupem je boolovský dotaz a výstupem je seznam databázových dokumentů odpovídající dotazu v klesajícím pořadí podobnosti (relevance).

2. Způsob řešení

Aplikace při spuštění serveru nejprve načte všechny soubory a postupně je stemmuje (tedy redukuje slova na společný základ, odstraňuje duplicity a stop slova) a tokenizuje (zařazuje zredukované základy slov do pole). V další fázi se iterují jednotlivá slova (teď už vlastně termy se kterými pracujeme) a počítá se jejich frekvence v kolekci. Tato frekvence se poté mapuje na termy a jména souborů, ze kterých pochází.

2.1. Invertovaný index

V další fázi se vytváří samotný invertovaný index (v naší implementaci se jedná o mapu, kde jsou termy namapovány na pole obsahující jméno souboru, ze kterého term pochází a jeho váhu). Váha se pak počítá pomocí schématu tf-idf. Mějme kolekci dokumentů d a termy t. Pro tyto potřeby jsou nejprve spočítány hodnoty df_i (což je počet dokumentů v kolekci, ve kterých se daný term vyskytuje alespoň jednou). Poté můžeme spočítat hodnoty tf_{ij} (normalizovaná frekvence termu t_i z dokumentu d_j vztažená na celou kolekci) následovně:

$$tf_{ij} = \frac{f_{ij}}{max_i(f_{ij})}$$

Kde f_{ij} je frekvence termu t_i v dokumentu d_j a $max_i(f_{ij})$ je nejvyšší frekvence termu t_i v celé kolekci.

Následně spočítáme idf_i (inverzní dokumentovou frekvenci) a to jako:

$$idf_i = log_2\left(\frac{|d|}{df_i}\right)$$

Výslednou váhu termu potom spočítáme jako

$$w_{ij} = idf_i * tf_{ij}$$

S takto vypočtenými vahami máme všechno potřebné k vytvoření invertovaného indexu, který si server uloží ve formě JSONu do souboru pro použití při dalším spuštění. Zároveň si server drží invertovaný index v paměti a čeká na požadavky od klienta.

2.2. Zpracování dotazů

Server přijímá dotazy ve formě textových řetězců. Ty následně parsuje na jednotlivá slova a logické spojky (program zpracovává jak formát logických operátorů typu “&&” nebo “||” ale i textový formát “and”, “or”). Výsledkem je pole tokenů obsahující logické operátory a jednotlivá slova.

V další fázi je z tohoto pole tokenů vytvořen AST (Abstract Syntax Tree), kde jednotlivé nody stromu představují logický operátor nebo samotný výraz. Tento strom je tvořen rekurzivně zpracováním pole tokenů. V rámci zpracování jsou brány v potaz také priority operátorů.

2.3. Zpracování dotazů

Samotný strom se ve výsledku zpracovává rekurzivně. Při vyhodnocení listů stromu – tedy samotných výrazů – jsou vráceny hodnoty v invertovaném indexu včetně váhy a umístění souboru.

Při vyhodnocení AND operátorů se postupuje podobně jako při zatřiďování v merge sortu. Vzhledem k tomu, že výrazy v listech jsou vzestupně třízeny podle jmen souborů kde se nachází, můžeme tuto strategii využít. Postupně zvyšujeme index levého nebo pravého souboru tak, abychom byli stále na stejné úrovni. Indexy odpovídají číslu souborů. V momentě, kdy se indexy rovnají, přidáme nový záznam do výsledku. Váha toho záznamu bude počítána následujícím způsobem (w_La $w_R$jsou popořadě váhy levého a pravého výrazu, které byly rekurzivně zpracovány dříve):

$$w = \frac{\sqrt{(1 - w_L)^2 + (1 - w_R)^2}}{2}$$

Vyhodnocování OR operátoru probíhá podobně s tím rozdílem, že v případě, že se indexy neshodují tak je stejně přidáme do výsledku. Další věc, která se zde liší je samotné počítání váhy při shodě. Ta se počítá tímto způsobem:

$$w = \frac{\sqrt{{w_L}^2 + {w_R}^2}}{2}$$

Při vyhodnocování NOT operátoru se použije stejné vyhodnocení, jako kdyby to NOT operátor nebyl. Tedy spustí se vyhodnocení toho, co je uvnitř nodu úplně normálně. Poté co dostaneme tyto záznamy, vrátíme právě ty, které v tomto výsledku nejsou – tedy NOT operátor. Také všechny tyto záznamy budou mít váhu 1, což vyplývá z logiky věci. V podstatě tedy do výsledku propagujeme zbylé dokumenty v kolekci, které nebyly nalezeny v původním hledání výrazu.

3. Implementace

3.1. Použité technologie a knihovny

Pro implementaci projektu byl zvolen jazyk Javascript. Projekt je rozdělen na část serveru a klienta, kde klient využívá framework React.js a server Node.js. Program využívá pro svou činnost různé balíčky (knihovny) pro práci s komunikací mezi klientem a serverem, kromě těchto využívá jmenovitě balíček Natural, který se stará o zpracování stemmování/lemmatizace textu a následnou tokenizaci. Aplikace nevyužívá žádnou specializovanou databázi a přijímá soubory ve file systému počítače s uloženým textem.

4. Příklad výstupu

Mějme uživatelem zadaný vstup. Například tento:

Obrázek 1 - výraz zadaný uživatelem

Jak už bylo zmíněno, aplikace reaguje na prioritu operátorů a bere v potaz klíčová slova and/or/not. Tento výraz tedy bude parserem přeložen na:

(shakespear && (!hamlet)).

Samotný AST výrazu pak vypadá následovně:

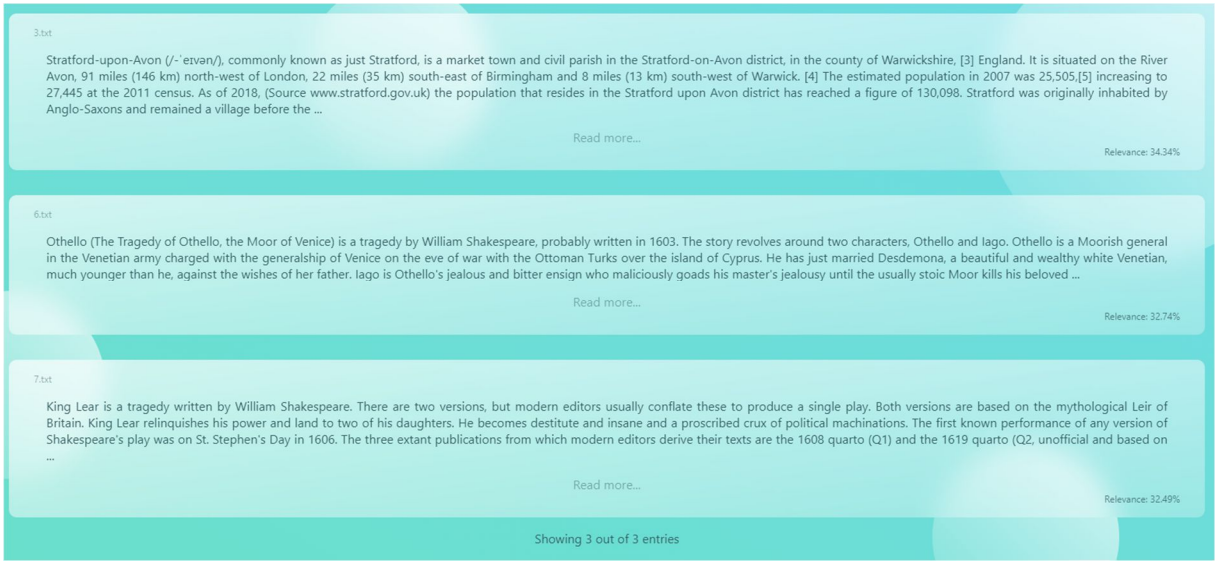
```
AndNode {
  lVal: Node { value: 'shakespear' },
  rVal: NotNode { value: Node { value: 'hamlet' }, operator: '!' },
  operator: '&&'
}
```

Obrázek 2 - AST výrazu 'shakespeare and not hamlet'

Tento strom je zpracován a výsledkem jsou již samotné soubory z kolekce a jejich relevance vůči danému výrazu. Níže přikládám odpověď serveru na zadaný dotaz a také výstup v samotné webové aplikaci.

```
[
  { file: 3, weight: 0.34335188487328683 },
  { file: 6, weight: 0.3273527956348028 },
  { file: 7, weight: 0.3248913972904206 }
]
```

Obrázek 3 - odpověď serveru na dotaz



Obrázek 4 - výsledek ve webové aplikaci

5. Experimentální sekce

Odhadovaný počet řádků jednoho souboru v kolekci je zhruba 93. Na kolekcích různých velikostí jsme porovnávali rychlost vytvoření invertovaného indexu. Data jsou přiložena v následující tabulce a grafu. Pro menší zkreslení údajů byly rychlosti měřeny na jednotlivých kolekcích několikrát.

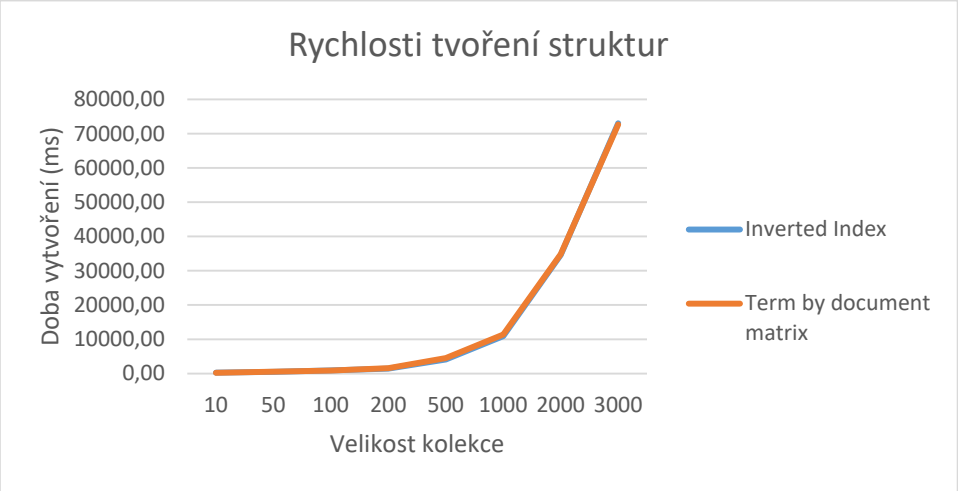
Velikost kolekce	Doba vytvoření inverted indexu (ms)				Průměrné vytvoření jednoho řádku (ms)
	1	2	3	průměr	
10	254	350	260	288,00	0,31
50	475	489	544	502,67	0,11
100	884	841	994	906,33	0,10
200	1399	1543	1423	1 455,00	0,08
500	3938	4138	4245	4 107,00	0,09
1000	10826	11083	10821	10 910,00	0,12
2000	34455	34673	34618	34 582,00	0,19
3000	72901	73978	72263	73 047,33	0,26
Průměr					0,16

Odhadovaný čas vytvoření invertovaného indexu pro jeden řádek je tedy průměrně dle uvedeného 0.16 milisekund.

Stejná tabulka je zpracována také při vytváření term-by-document matice.

Velikost kolekce	Doba vytvoření term by document matice (ms)				Průměrné vytvoření jednoho řádku (ms)
	1	2	3	průměr	
10	269	252	258	259,67	0,28
50	570	477	565	537,33	0,12
100	930	856	861	882,33	0,09
200	1531	1579	1556	1 555,33	0,08
500	4448	4319	4721	4 496,00	0,10
1000	11502	11578	11020	11 366,67	0,12
2000	35709	34336	34453	34 832,67	0,19
3000	74382	72256	71347	72 661,67	0,26
Průměr					0,15

Nakonec přikládám graf s porovnáním obou struktur.

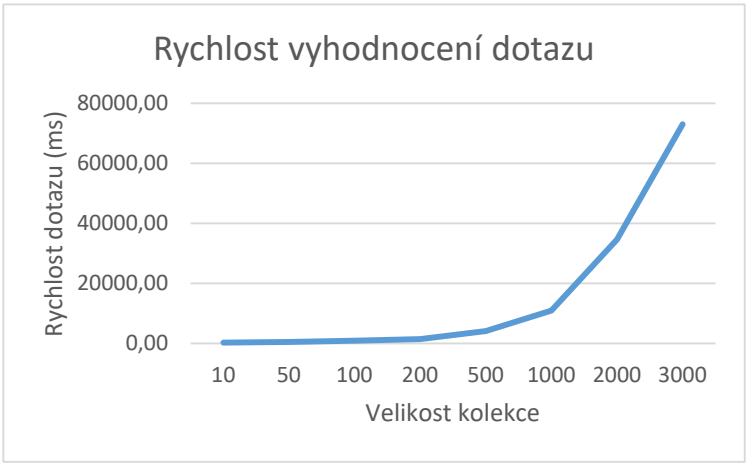


Na základě těchto měření je patrné, že při samotném vytváření nevyniká ani jedna ze struktur co se týče rychlosti. Samotné výhody jsou patrné až při dotazování.

V projektu není implementováno hledání pomocí term-by-document matice, ale je známo, že toto hledání je obecně mnohem pomalejší, vzhledem k potřebě lineárního procházení každého výrazu. Obecně se tedy nevyplatí jí vůbec používat a my jí zde máme jen pro porovnání rychlosti tvorby této struktury.

Dalším měřením na našem projektu byla rychlost vyhodnocování dotazů v závislosti na velikosti kolekce (zde byl konkrétně zvolen dotaz ‘shakespeare and not hamlet’):

Velikost kolekce	Rychlost dotazu (ms)			
	1	2	3	průměr
10	9,6	4,4	3,1	5,70
50	8,9	5,7	3,7	6,10
100	10,9	6	4,9	7,27
200	10,2	5	6,7	7,30
500	13,2	7	6,7	8,97
1000	18,9	12,6	14,8	15,43
2000	43,9	40,7	31,9	38,83
3000	84,8	77,7	79,5	80,67



6. Diskuse

Projekt slouží jen pro školní účely a nejsou v něm tedy vyladěné všechny implementační detaily, které souvisí s implementací rozšířeného boolovského modelu.

Možnost pro zlepšení je zejména u operátoru NOT, kdy například pro výraz: cat or not dog, budeme výsledek vracet ve tvaru, kdy první budou soubory, které obsahují cat a neobsahují dog, poté soubory, které neobsahují cat ani dog a poté soubory, co obsahují cat i dog – tyto by měly nejmenší váhu. Ovšem naše implementace OR operátoru, které jsou srovnány podle váhy odpovídá přirozenému chování operátoru OR. Tedy všechny

soubory, které neobsahují dog budou mít váhu 1 a budou logicky v popředí. Další možnost pro zlepšení je samotný sorting výsledku. Array.sort() v JavaScryptu na platformě Linux není stabilní – tedy prvky, které mají stejné hodnoty nezachovají stejné pořadí jako bylo původně. Na platformě Windows tento sort stabilní je – podle lokálních testů. Možnost pro zlepšení by byla právě implementace sortu, který je stabilní na všech platformách. Ovšem pro tento projekt to je zbytečné, jelikož námi požadované výsledky budou stejně srovnány na základě váhy. Na samotném pořadí stejných prvků nezáleží.

7. Závěr

Projekt měl za cíl zpracovat boolovský model vyhledávání nad full-text kolekcemi. Osobně považuji tento projekt za úspěšný, až na drobné implementační detaily.