KIV/IR - semestrální práce - jednoduchý search engine

A22N0060P Jiří Trefil

Úvod do problematiky

Problematika vyhledávačů je velice snadná na pochopení. Uživatel zadá nějaký dotaz (tzv. query), pro které vyhledávač má najít relevantní dokumenty. Již použití slova “relevantní” nutně implikuje, že se relevance dotazu vůči dokumentům musí počítat nějakou metrikou. U takovéto metriky je nutné (nad rámec klasických vlastností metriky), aby její hodnota pro relevantní dokumenty byla vysoká a pro irelevantní dokumenty blížící se k 0.

To je v jádru celé, co má vyhledávač dělat. V následujících kapitolách bude rozepsána má implementace jednoduchého vyhledávače s pokusem o vysvětlení, proč je tento úkol vlastně silně netriviální, ač vypadá snadně.

Analýza problému

V následující kapitole zkráceně popíšu problematiku získání dokumentové kolekce. vytváření invertované indexu a proč jej vůbec vytvářet. Jak interpretovat dokumenty a uživatelské dotazy. Samotná volba metriky pro výpočet relevance dokumentů k dotazu a speciální případ booleovského vyhledávání ve vyhledávačích.

Akvizice datové sady

Aby dával vyhledávač jakýkoliv smysl, musí mít nějaké dokumenty nad kterými může vyhledávat. Dokument v kontextu vyhledávání informací je abstraktní entita reprezentující jakýsi blok informací. Může to tedy být typicky webová stránka, nějaký pdf dokument atp.

Jelikož doména mého vyhledávače jsou články BCC News, tak jsem potřeboval vytvořit dostatečně velkou datovou sadu (v řádech desítek MB článků), nad kterou by uživatel mého vyhledávače mohl ukojit svoji zvědavost.

Pro efektivní “prolezení” pavučinou BBC News jsem použil tzv. “crawler”, který automatizuje a zefektivňuje stahování obsahu webu. Crawleru stačí pouze předat seznam XPATH výrazu, které popisují, jaký obsah má z webu stáhnout.

Uložení dokumentů na serveru

Samotné stažení dokumentů crawlerem samozřejmě nestačí, je nutné je nějak perzistentně uložit a vytvořit tím tzv. document cache (skupina (optimálně množina) dokumentů). Problém perzistentního uložení adresuje filesystém operačního systému či nějaký databázový systém (v podstatě mnoho B-stromů, které optimalizují přístup na filesystém~disk).

Předzpracování textu

Předzpracování textu není esenciální pro fungování vyhledávacího systému, ale definitivně zlepšuje jeho výkon, takže by bylo bláhové nevyužít jej. Předzpracování textu je vlastně už poměrně kvalitně zmapovaná disciplína z výzkumu v oblasti umělé inteligence - zpracování přirozené jazyka. Myšlenkou tohoto procesu je vyhodit všechna informačně nedůležitá slova (stop words), provést takzvaný stemming a rozdělit korpus na tokeny.

Vyhození stop slov - proč? Tyhle slova se vyskytují téměř všude, takže jejich informační hodnota je téměř nulová (například slova “the, and, or” nenesou moc informace, nemá proto smysl s nimi jakkoliv pracovat). Jak je vyhodit? Stáhnutím seznamu stop slov, každý jazyk má nějaký. V mém případě potřebuji seznam stop slov pro anglický jazyk.

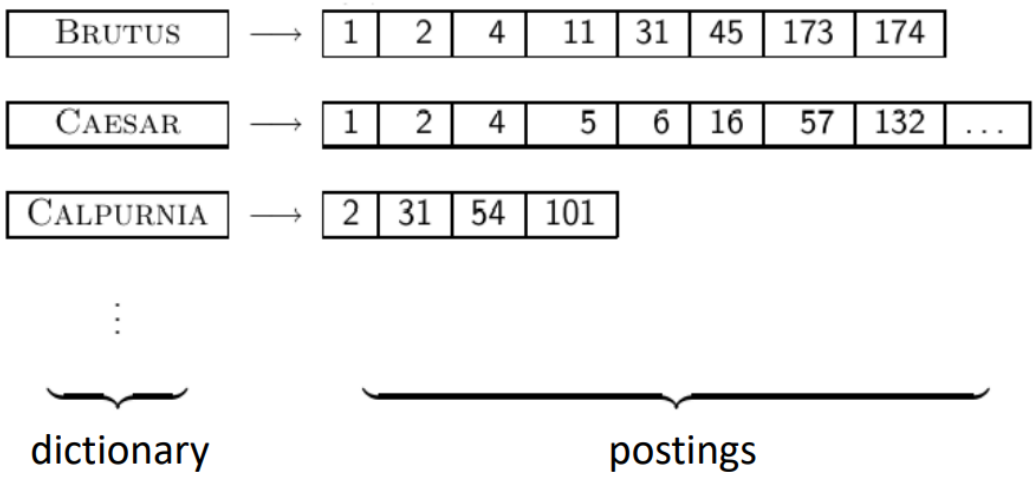
Stemming je v podstatě heuristická verze lematizace, která je výpočetně rychlá a dostatečně dobrá pro praktické účely. Tento proces zredukuje slova do jejich základního tvaru, díky tomuto nemusíme řešit různé formy (s –ing na konci, v jiném času) slova. Tento problém pro anglický jazyk vyřeší Porterův stemmer, který v sedmi fázích zredukuje slovo na jeho základní tvar (kořen).

Tokeny v korpusu jsou jednotlivá slova. Předzpracování ovšem nesmí být příliš agresivní, aby nedošlo k příliš velkému zahození informací a tedy snížení kvality vyhledávače.

Pro novinové články samozřejmě toto samotné nebude stačit, jelikož obsahují kvanta dalších tokenů, které jsou pro vyhledávač irelevantní, jako například html tagy, uri adresy, outlook reference na emaily atp, které je chtěné zahodit. Tyto výskyty jdou ale dostatečně dobře popsat pomocí regulárních výrazů (zjednodušený zápis akceptačních konečných automatů) a následně odstranit.

Vytvoření invertované indexu

Invertovaný index je alternativa k maticové reprezentaci termů a dokumentů. Takovéto matice bývají z pravidla řídké a zabírají místo v paměti neužitečnou informací. Invertovaný index je datová struktura, která se skládá z termů a pointerů (nebo referenčních proměnných) na posting listy. Pro lepší představu viz obrázek 1. Invertovaný index je esencí celého vyhledávače, protože díky němu víme, která slova se nachází ve kterých dokumentech a můžeme dle toho začít počítat relevanci dokumentů k dotazu.



Obrázek 1: Invertovaný index

Reprezentace dokumentů a dotazů pro výpočet relevance

V téxtové formě není dotazy ani dokumenty možné použít pro výpočet čehokoliv užitečného. Je tedy nutné transformovat přirozený jazyk do vhodné numerické podoby. Toto je opět disciplína, kterou se zabývá zpracování přirozeného jazyka. Vývoj v této oblasti byl poměrně dlouhý (časově), ale v podstatě existují dva přístup. N-gramové modely, které byly historicky používány a opírají se o Markovovy řetězce (ppnost jevu závisí pouze na aktuálním a předchozím jevu. V kontextu NLP je jev slovo). Tedy například bigramový model uvažuje jeden předchozí jev (slovo). Tímto způsobem by bylo možné vyjádřit relevanci dokumentu k dotazu pravděpodobností a bylo by možné tento model použít. Druhým a revolučním popisem přirozeného jazyka je tzv. word-embedding, o český překlad se ani nebudu pokoušet. Velice prostá myšlenka, která do N dimenzionální prostoru promítá vektory, každý vektor reprezentuje právě jedno slovo a každá položka vektoru reprezentuje určitý “atribut” slova.

Pro příklad by bylo možné definovat word-embedding prostor jako N=3, tedy každé slovo budu reprezentováno jako tříprvkový vektor a každé slovo bude popsáno pomocí atributu [chlupatý, pojidatelný, nebezpečný]. Pokud by se promítly slova “kočka” a “želva” do toho prostoru, pak by je bylo nutné transformovat do vektoru například následovně, kočka = [1,0.4,0.1] a želva=[0,0.5,0]. Již zde je vidět, že tyto body v prostoru nebudou u sebe blízko, protože podobnost atributů želvy a kočky je mizivá. Je zde také možné udělat velice jednoduchou indukci – body sobě blízké si jsou sémantické podobné. Tato úvaha je esencí word embeddingu – slova sobě sémanticky blízká chceme blízko sobě.

Tato metoda by také šla využít pro reprezentaci dokumentů a dotazů a troufám si říct, že by fungovala lépe jak N-gramový model.

Nýbrž implementace neuronové sítě (respektive stažení natrénované enkodéru) je pro účel mého vyhledávače ekvivalent kanónu na vrabce a vystačím si s jednoduchým ohodnocením pomocí tf-idf.

Tf-idf penalizuje slova často se vyskutující v kolekci a bonifikuje slova, která jsou v kolekci vzácná. Tato metoda váhování také uvažuje takzvanou term-frequency (tf) a document frequency (pouze je ve vztahu použita inverzní hodnota, proto idf). Term frequency udává četnost termu v daném dokumentu, tedy

kolikrát se slovo daném dokumentu vyskytuje. Document frequency zase říká v kolika dokumentech se slovo vyskytuje. Pomocí vztahu na obrázku 2 můžeme vypočítat váhu termu pro daný dokument a vytvořit tím vektorovou reprezentaci, která je velice vhodná pro vyhledávací systém.

Druhým méně vhodným, ale levným a relativně funkčním modelem popisu dokumentů a dotazů je Bag of words model. Ten funguje velice jednodušše - jednotlivá slova nahradíme jejich četností a veškeré ostatní informace (například poziční) zahodíme.

$$w_{t,d} = (1 + \log \mathrm{tf}_{t,d}) \cdot \log \frac{N}{\mathrm{df}_t}$$

Obrázek 2: výpočet tf-idf váhy

Výpočet relevance

Po úspěšném reprezentování dokumentů a dotazů vektory je možné aplikovat znalosti z lineární algebry a vypočítat vzdálenost mezi těmito vektory. Metrik pro výpočet vzdálenosti je několik, například Euklidovská, Manhattanská či Cosinová. Euklidovská v tomto případě není vhodná, jelikož nás sice zajímá konkrétní hodnota prvků vektoru, ale není to tím rozhodujícím faktorem. Proto není vhodná Euklidovská vzdálenost, jelikož se orientuje spíše na “váhu” (magnitude). Manhattan distance je speciální případ Euklidovské, kdy se počítá se “blokovitým terénem” (odtamtud také název dle Manhattonu a členení měst na bloky) - v tomto případě je také k ničemu. Cosinová vzdálenost je zde optimální, jelikož vyjadřuje podobnost dvou vektorů úhlem, který mezi sebou svírají spíše než podobnostmi jejich hodnot. Cosinová vzdálenost (podobnost) se vypočítá podle vztahu na obrázku 3. Ve jmenovateli je L_2 norma, která provádí délkovou normalizaci – tedy chceme do výpočtu relevance zahrnout i délku jednotlivých vektorů.

$$\cos(\vec{q}, \vec{d}) = \mathrm{SIM}(\vec{q}, \vec{d}) = \frac{\vec{q} \cdot \vec{d}}{|\vec{q}||\vec{d}|} = \frac{\sum_{i=1}^{|V|} q_i d_i}{\sqrt{\sum_{i=1}^{|V|} q_i^2} \sqrt{\sum_{i=1}^{|V|} d_i^2}}$$

Obrázek 3: cosinová podobnost

Závěry z analýzy

Je nutné vytvořit robustní datovou sadu prolézáním BBC News domovské stránky a stahováním jednotlivých článků. Články je nutné perzistentně uložit - na filesystém. Obsah článků je nutné předzpracovat pro odebrání informačně nedůležitých slov, znaků a výrazů. Následně je nutné vytvořit invertovaný index, přes který se budou hledat relevantní dokumenty k uživatelskému dotazu.

Implementace

V následující kapitole stručně popíši implementace systému a mé řešení problémů, které vznikly po cestě.

Semestrální práce je rozdělena na 2 disjunktní aplikace. Backend (serverová část) je implementována v jazyce Java a v hojně rozšířeném frameworku Spring boot. Frontend (klientská část) je implementována ve frameworku React, tedy v javascriptu. Aplikace se spouští nezávisle na sobě - klientská aplikace se nepodílí na logice vyhledávání, pouze posílá requesty na server. Backend je navržen a implementován jako RESTful aplikace, tedy nedrží si žádný vnitřní stav dle kterého řídí logiku (s vyjímkou stavu crawleru, aby nebylo možné server levně DDOSnout). Aplikace je strukturová dle MVC architektury, a to jak na klientské, tak na serverové části. Serverová část má 3 hlavní vrstvy. Controller vrstva (zde přijímá HTTP requesty), service vrstvu (služby, které se provolávají navzájem či z controllerů a vykonávají tu "těžkou práci" a předávají informaci o výsledku). Model vrstva obvykle obsluhuje komunikaci s databází, ale jelikož jsem databázi v semestrální práci nepotřeboval, tak jsem ji neimplementoval. Vrstva samotná tedy nemá žádnou logiku, pouze definuje entity a dto (data transfer object – design pattern pro předávání dat nějakým formálním způsobem - v Javě to jsou třídy).

Implementace crawleru

Ze zadání semestrální práce vychází, že crawler nutně implementován být nemusí, ovšem problematika je zajímavá, proto jsem jej implementoval. Při testování aplikace mi sloužil pro "občerstvování" document cache novými články, neexistuje však způsob, jak jej zavolat přímo z klientské aplikace.

Crawler samotný je implementován pomocí farmer:slave modelu. Existuje jeden farmář, který má N otroků. Farmář zadává otrokům práci a ti ji vykonávají, když není práce, uspí se. V kontextu semestrální práce je farmářem třída Crawler, která v sobě obsahuje N (parametr konstruktoru) otroků, kteří běží paralelně (dědí od třídy Thread). Ti stahují články, předávají je na zaindexování a zároveň je zapíši na filesystém ve formátu json.

Document cache

Document cache je v mém případě reprezentována jako adresář na filesystému, který v osobě obsahuje N json souborů s pevným formátem:

- "title":"<titulek článku>"
- "author" : "<autoři článku">
- "content":"<celý obsah článku vyjma titulku a autorů>"

Dokumenty v tomto úložiště jsou následně používany pro zaslání krátké "výstřižku" textu článku na klientskou stranu.

Předzpracování BBC článků

Samotné předzpracování dat nebylo vůbec triviální. Regulárními výrazy bylo nutné pokrýt ještě odebírání číslic, všemožných závorek, dat (kalendářních), emailů, url adres a hlavně html tagů, které byly v textu občas vnořené. Všechno vyjma html tagů lze poměrně dobře popsat regulárními výrazy. HTML tagy je možné pouze dostatečně dobře aproximovat, protože neexistuje jeden konkrétní vzor "pattern", kterými se tagy řídí. Ve výsledku se mi ale podařilo dostatečně dobře pokrýt většinu tagů a odstranit tak zbytečné informace z textu.

Samotné neodstanění tagů, url, …, by nebyl problém pro vyhledávání, fungovalo by to i tak, ovšem s nižší přesností a docházelo by k indexování nesmyslných věcí (jako například číslic či emailů), které nejspíš nikdo nebude vyhledávat při brouzdání BBC články. Dobrým preprocessing se tedy nejenom ušetřila paměť potřebná pro invertovaný index, ale také se zvýšila kvalita vyhledávání respektive výpočtu relevance.

Implementace invertovaného indexu

Invertovaný index je jádrem celého vyhledávače. Rozdíl mezi invertovaným indexem a reprezentací pomocí matice je v náročnosti časové i paměťové. Operace nad maticí jsou samozřejmě rychlejší, nýbrž její paměťová náročnost O(počet_termů*počet_dokumentů) je neúnosná. Paměťová náročnost invertovaného indexu je O(počet_termů+délka_posting_listu). V nejhorším případě, tedy že se všechny termy vyskytují ve všem dokumentech se dostaneme na paměťovou náročnost matice, v očekávaném případě bude délka_posting_listu << počet_termů.

Jak již bylo v analýze řečeno, posting list datová struktura, která mapuje posting listy na termy. Posting list obsahuje posting items, které v sobě nutně nesou **id** dokumentu a případně další metadata (váha, frekvence, poziční informace, …). V mé implementace posting listu každý posting item obsahuje **id** a **váhu**.

Invertovaný index se vytváří při spuštění serveru. Dojde k přečtení všech dokumentů v document cache a jednotlivé články se proženou preprocesorem a následně jsou zaindexovány. Celý proces netrvá příliš dlouho, ale samozřejmě zde **silně** závisí na vytíženosti sběrnice a CPU. Jelikož vytváření invertovaného indexu je náročná operace na IO (čteme z disku), tak je vlákno různě plánováno jádrem OS. Klasický průběh tedy je zavolání syscallu OPEN, který ale je blokující a vlákno je tímto tedy vyhozeno z procesoru (zde nastává zpomalení), následně je vlákno vhozeno do fronty ostatních vláken čekajících na konkrétní IO operaci (zde čtení z disku, opět další zpomalení). Samotná velikost json souboru s článkem je v řádech jednotek kB.

Pro moji implementaci vyhledávače je document cache cca 40 MB velká, takže spuštění serveru včetně vytváření indexu trvá cca 40 vteřin - pokud je hardware napojen na síti a není na žádném spořícím režimu (zajímavé sledování bylo vypojit notebook ze sítě, kde se jádro windowsu agresivně přepne do úsporného režimu (ač je výkon nastaven na maximální) a samotné IO operace trvají věky (vytváření invertovaného indexu > 70 vteřin)).

V případě větší (stovky MB i více) by nejspíš bylo vhodné agregovat dokumentu do větších celků, aby se zredukoval počet volání OPEN. Tento problém jsem bohužel neměl čas adresovat při implementaci semestrální práce z časových důvodů.

Samotný invertovaný index je **hierarchický**, v mém případě má **2** vrstvy. První vrsta je “titulková” a druhá vrstva je “obsahována”. V následující podkapitole o vyhledávání popíši logiku zpracování dotazů.

Implementace vyhledávání/vyhledávače

Samotný vyhledávač (v mém případě obalovací třída SearchEngine nad invertovaným indexem) pouze přebírá uživatelský dotaz a vyhledá pro něj relevantní dokumenty v invertovaném indexu. Následně si výsledek vyhledávání zacachuje (do operační paměti), pokud by nějaký jiný uživatel měl stejný dotaz, tak už znovu nemusí pracně počítat všechny relevantní dokumenty, ale vrátíme již známý výsledek.

Tento přístup cachování v paměti je velice rychlý, má však velký problém se škálováním - pro více uživatelů by byl brzy neudržitelný, bylo by zde vhodné opět nasadit databázi (například Redis, která je in-memory a ztratí data po ztrátě napájení, ideální cache).

Paralelně vedle sebe na serveru existují 3 vyhledávače, booleovský, bag-of-words model a tf-idf model. Booleovský vyhledávač je velice ochuzený a poskytuje pouze AND spojení mezi slovy.

Implementace plně funkčního booleovské vyhledávače by nutně musela obsahovat nějakou definici vlastní formální bez kontextové gramatiky, která by umožnila transformovat uživatelský dotaz do derivačního a posléze abstraktního syntaktického stromu. To hlavně z důvodu vynucení priority závorek (ve stromě by byly termy v závorkách “hlouběji”). Tento problém by šel adresovat například pomocí knihovny Lucene, která toto parsování umí, případně nadefinovat si vlastní gramatiku a buďto si implementovat vlastní zásobníkový LR automat (extérmně složité a pracné) nebo využít nějakou knihovnu, která jej již implementuje (yacc/bison) nebo případně alternativu rekurzivního sestupu (antlr). Bohužel je toto další oblast, která mi propadla pod rukama kvůli časovému presu.

Princip vektorových modelů bag-of-words a tf-idf byl vysvětlen již v analýze, nebudu tady tedy zabíhat do detailů, implementace ve zdrojovém kódu je snad čitelná a adekvátně okomentovaná.

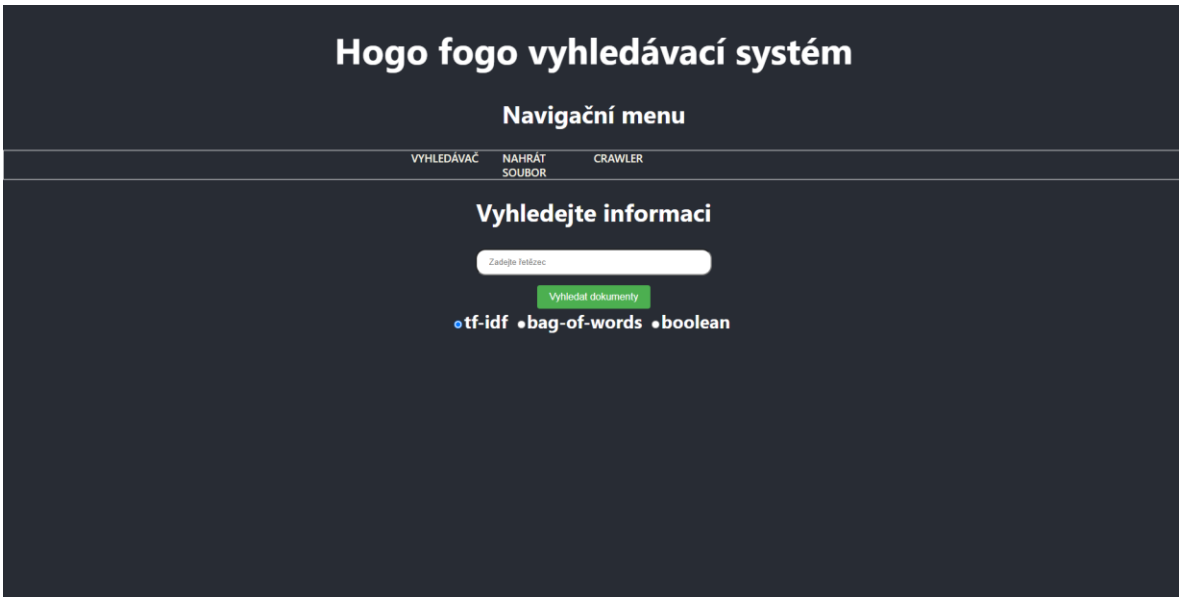
Samotný invertovaný index je **hierarchický**. Jeho dvě úrovně jsou “titulková” a “obsahová”, jak již bylo řečeno. Při vyhledávání dokumentů je tedy nejdříve prohledán index titulkový, poslézene je prohledán index obsahový. Relevance dokumentu má samozřejmě nějaký threshold po kterém dokument již není považován za relevantní, v mé implementace pro tf-idf model je tento threshold hodnota **0.1**. Na klientskou stranu jsou poslány dokumenty seřazeny dle relevance, nejdříve jsou prezentovány relevantní dle titulku, následně až dle obsahu.

Tímto je završena implementace té relevantní části, kterou nám má předmět IR přidat. Implementaci klientské aplikace velice stroze popíši, protože není tak zajímavá ani relevantní.

Klientská aplikace

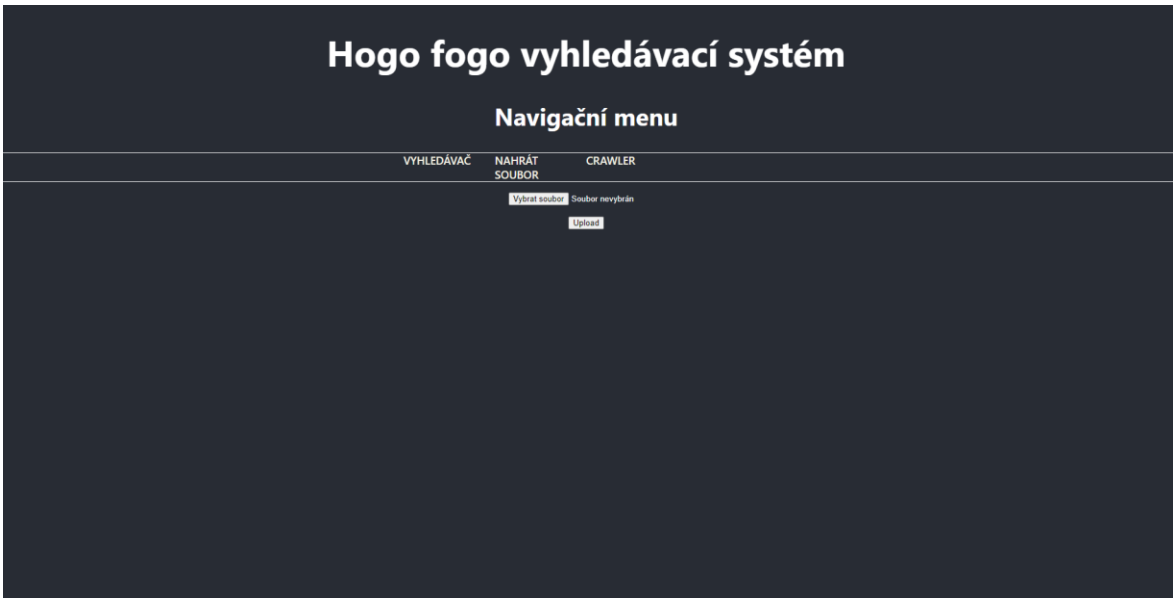
Jak již bylo řečeno, aplikace je implementována pomocí knihovny Reactjs, kterou vyvinul Facebook / META. Framework je komponentově orientovaný a následuje deklarativní paradigma. Velkou výhodou knihovny je client-side rendering, tedy všechny složité vykreslovací výpočtu oddře klientský stroj a nemusí se tím zatěžovat server. Tento fakt ale také nutně říká, že pokud bude ta aplikace příliš výpočetně “těžká”, tak se uživatel vůbec nemusí na náš vyhledávač dostat a nebo to bude trvat příliš dlouho a radši uteče jinam.

Aplikace samotná obsahuje tří hlavní stránky, které umožní vyhledávat, nahrát dokument a případně stáhnout bbc článek z webu. Viz obrázek 4, 5, 6.



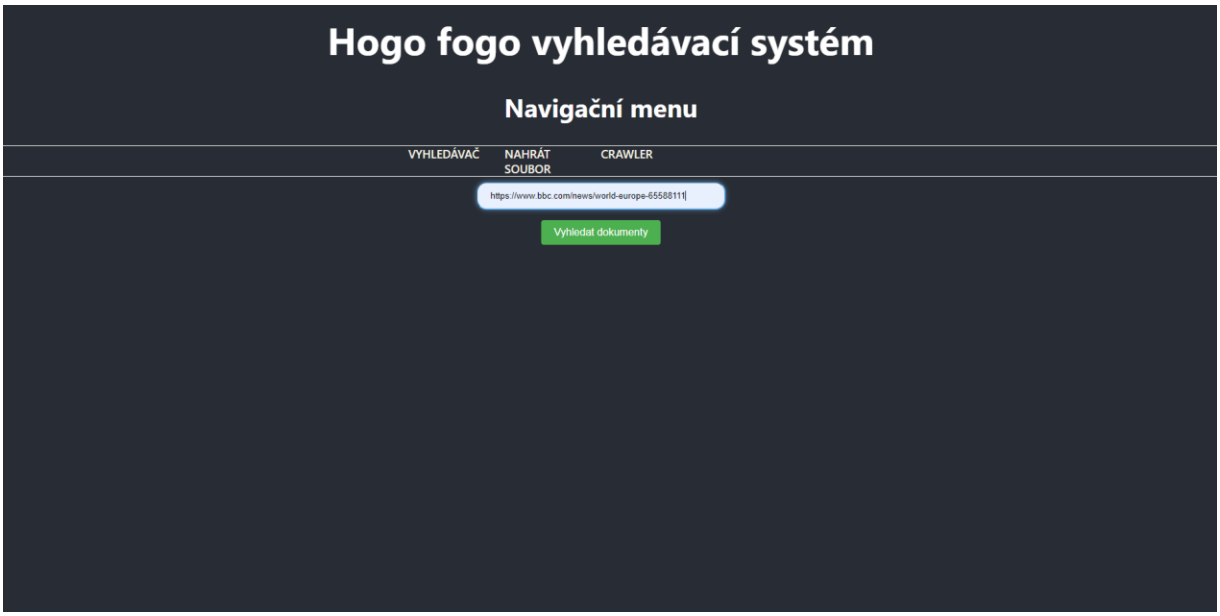
Obrázek 4: vyhledávač

Pod samotným zadání dotazu si uživatel může zvolit model, který chce použít pro vyhledávání. Defaultní je tf-idf.



Obrázek 5: Nahrání souboru

Uživatel si zde může nahrát soubor ve formátu json (viz analýza problému), který je posléze zaindexován.



Obrázek 6: Stažení článku

Zde uživatel může zadat url adresu na bbc news article, který chce stáhnout. Článek je následně stažen a zaindexován.

Bonusové body implementace

- pozdější doindexování dat - přidání nových dat do existujícího indexu
- ošetření HTML tagů
- vylepšení vyhledávání
  - Implementace hierarchického indexu dle mého názoru zlepšuje kvalitu vyhledávání
- více *scoring* modelů
  - Implementován nad rámec ještě bag-of-words model
- indexování webového obsahu - zadám web, program stáhne data a rovnou je zaindexuje do existujícího indexu,
- další předzpracování normalizace
  - Obecně s předzpracováním textu článku bylo relativně hodně práce, jelikož autoři překvapivě neumí úplně dobře psát respektive jejich text asi neprochází moc velkou kontrolou. Například vyhazování URL adres bylo složité, jelikož ne vždy byla za tečkou v textu mezera, například: Coronation of King **Charles.Late** queen …. kde vzniklá problém s Charles.Late, jelikož regulární výraz toto validně může označit za url adresu a z textu vyhodit, což není chtěné. Vypilovat tedy regex pro tyto případy bylo mírně únavné, ale poučné.
- GUI/webové rozhraní

“Nad rámec” byla implementace celé aplikace jakožto serveru pro obsluhu N klientů (což nejsem jediný, který to dělal), kde člověk musí řešit ještě problémy pararelního programování. Například když N různých vláken manipuluje s jednou instancí vyhledávače, tj. správné zamykání při zapisování do indexu (například při uploadu souboru případě stažení článku).

Za což tedy body neočekávám, nýbrž mě to nutilo trochu jinak navrhnout architekturu aplikace, aby bylo možné efektivně pracovat s vyhledávač z vícero zařízení a zároveň nezanedbat nějakou kritickou sekci.

Obecně implementace této semestrální práce byla velice poučná a zajímavá, osobně si myslím, že kdybych to implementoval jako klasickou desktopovou aplikaci v Javě, tak s tím mám méně práce, ale nebylo by tak zajímavé.

Spuštění aplikací

Jelikož semestrální práce je implementována jako 2 disjunktní aplikace, tak je samotné spuštění nestandardní. Je nutné spustit klientskou aplikaci v Reactu (**je nutné mít nainstalovaný nodejs**) a následně Java Spring boot aplikaci.

Do kořenového adresáře jsem umístil skript **run_server.cmd a run_client.cmd**, který by měl spustit obě aplikace v případě, že je prostředí připraveno, tedy primárně nainstalován node.js (interpret javascriptu na desktop).

Pozor, pokud klient nebude spuštěn na portu 3000, server s ním bude odmítat komunikaci. Toto je úmyslně nastaveno v rámci bezpečnosti.