ČVUT, FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
VYHLEDÁVÁNÍ NA WEBU A V MULTIMEDIÁLNÍCH DATABÁZÍCH
LETNÍ SEMESTR 2019/2020
ZÁVĚREČNÁ ZPRÁVA K PROJEKTU

LSI vektorový model

David Mašek a Kristýna Klesnilová

13. května 2020

OBSAH

1 Popis projektu

V tomto projektu implementujeme *LSI vektorový model* sloužící k podobnostnímu vyhledávání v databázi anglických textových dokumentů. Tuto funkcionalitu následně vizualizujeme pomocí webového interface, který uživateli umožňuje procházet databázi článků na základě doporučování nejpodobnějších článků k právě čtenému.

V experimentální části projektu jsme se dále zaměřili na:

- Určení optimálního počtu konceptů
- Porovnání vlivu LSI na kvalitu výsledků vyhledávání s ohledem na výskyt synonym a homonym
- Porovnání průchodu pomocí LSI vektorového modelu se sekvenčním průchodem databáze s ohledem na čas vykonání dotazu
- Vliv různých vnitřních parametrů na výkon algoritmu (změna počtu konceptů, změna počtu extrahovaných termů, použití lemmatizace namísto stemmingu, odstranění číslovek při preprocesingu, použití jiného vzorce pro výpočet vah termů)

Celý náš projekt je volně dostupný k vyzkoušení na: `https://bi-vwm-lsi-demo.herokuapp.com/`.

2 Způsob řešení

2.1 Preprocesing dokumentů

Jako první v naší aplikaci začínáme s preprocesingem dokumentů. Slova z jednotlivých dokumentů převedeme na malá písmena a odstraníme z nich nevýznamová slova a interpunkci. K identifikaci nevýznamových slov používáme seznam anglických nevýznamových slov. Jako parametr programu posíláme také, zda má z dokumentů odstranit i číslovky. Následně na zbylé termy aplikujeme *stemming* či *lemmatizaci*. Tím se snažíme slova, která mají stejný slovní základ, vyjádřit pouze jedním termem. Stemming to dělá pomocí algoritmu, kterým odsekává přípony a koncovky slova. Lemmatizace na to jde o něco chytřeji, podle kontextu slova se pokusí určit, o jaký slovní druh se jedná, a podle toho ho zkrátit.[1] Porovnání jejich použití v programu se dále věnujeme v experimentální části.

2.2 Výpočet vah termů

V aplikaci vytváříme matici M_w, která má v řádcích jednotlivé termy a ve sloupcích jejich váhy v jednotlivých dokumentech.

Začneme tím, že si vytvoříme matici počtu výskytů jednotlivých termů v jednotlivých dokumentech. Počet termů v této matici poté dále zredukujeme, abychom pracovali jen s těmi nejdůležitějšími. Funkci pro redukci termů posíláme následující parametry:

- *max_df* - termy nacházející se ve více % dokumentů, než udává číslo *max_df*, z matice odstraníme
- *min_df* - termy nacházející se v méně nebo stejně dokumentech, než udává číslo *min_df*, z matice odstraníme
- *max_terms* - maximální počet termů, které si v aplikaci necháme

[1] `nlp.stanford.edu/IR-book/html/htmledition/stemming-and-lemmatization-1.html`

- *keep_less_freq* - udává, zda si při výběru *max_terms* termů nechat ty nejméně či nejvíce často zastoupené v dokumentech

Z této zredukované matice poté již spočteme matici M_w. Pro výpočet vah jednotlivých termů používáme metodiku *tf-idf*. Váhu termu t_i v dokumentu d_j spočítáme podle vzorce:

$$w_{ij} = tf_{ij} \cdot idf_{ij} \tag{2.1}$$

kde tf_{ij} reprezentuje normalizovanou četnost termu t_i v dokumentu d_j a spočítame ji podle vzorce[2] :

$$tf_{ij} = \frac{f_{ij}}{nt_j} \tag{2.2}$$

kde f_{ij} je četnost výskytu termu t_i v dokumentu d_j, kterou normalizujeme číslem nt_j vyjadřujícím celkový počet termů v dokumentu d_j. V přednášových slidech je použita normalizace jiná, tf_{ij} se tam počítá podle vzorce:

$$tf_{ij} = \frac{f_{ij}}{max_i\{f_{ij}\}} \tag{2.3}$$

kde $max_i\{f_{ij}\}$ vrací nejvyšší četnost termu t_i přes celou kolekci dokumentů. Tento způsob normalizace nám vrací spíše horší výsledky, jejich porovnáním se zabýváme v experimentální části. idf_{ij} reprezentuje převrácenou četnost t_i ve všech dokumentech a spočítá se podle vzorce:

$$idf_{ij} = \log_2(\frac{n}{df_i}) \tag{2.4}$$

kde n je celkový počet dokumentů a df_i reprezentuje celkový počet dokumentů obsahujících term t_i.

2.3 Implementace LSI

Jakmile máme vytvořenou matici vah termů M_w, můžeme přistoupit k samotné implementaci LSI. Princip LSI spočívá v tom, že s pomocí *singulárního rozkladu (SVD)* seskupíme tematicky podobné články do jednotlivých k konceptů. Vlivem počtu konceptů na kvalitu výsledků se dále zabýváme v experimentální sekci.

Singulární rozklad nám matici M_w rozloží následovně:

$$M_w = U \cdot S \cdot V^T \tag{2.5}$$

kde řádky matice U jsou obrazy řádků matice M_w, sloupce matice V jsou obrazy sloupců matice M_w a matice S obsahuje na diagonále *singulární hodnoty (absolutní hodnoty vlastních čísel)* matice M_w v sestupném pořadí. Z těchto matic získáme *concept-by-document* matici M_{cd} jako:

[2] `https://en.wikipedia.org/wiki/Tf%E2%80%93idf`

$$M_{cd} = S[k,k] \cdot V^T[k,:] \tag{2.6}$$

kde $S[k,k]$ značí prvních k řádků a sloupců matice S a $V^T[k,:]$ značí prvních k řádků matice V^T, kde k je počet konceptů. Nenásobíme tedy celou maticí M_{cd}, ale pouze její část podle počtu konceptů.

Matici projekce dotazu do prostoru konceptů M_q pak získáme jako:

$$M_q = U^T[k,:] \tag{2.7}$$

kde $U^T[k,:]$ značí prvních k řádků matice U^T.

2.4 Vyhodnocení dotazu

Při dotazu na nejpodobnější dokumenty k dokumentu d_j převedeme dotaz do prostoru konceptů na vektor V_c pomocí vzorce:

$$V_c = M_q \cdot M_{w_{:,j}} \tag{2.8}$$

kde $M_{w_{:,j}}$ značí j-tý sloupec matice M_w.

Vektor V_c poté pomocí *kosinové podobnosti* porovnáme se sloupcovými vektory matice M_{cd}. Indexy nejpodobnějších sloupcových vektorů matice M_{cd} pak vrátíme jako indexy nejpodobnějších dokumentů k dokumentu dotazu d_j. Spolu s indexy vrátíme i samotnou hodnotu kosinové podobnosti.

3 Implementace

Celý projekt jsme programovali v jazyce *Python*. Práci nám velmi usnadnila knihovna *NLTK*[3] nabízející rozsáhlou funkcionalitu pro práci s přirozeným jazykem. Využili jsme například *WordNetLemmatizer* pro lemmatizaci či *SnowballStemmer* pro stemming. Dále jsme v programu hojně využívali Python knihovny *pandas*[4] a *numpy*[5].

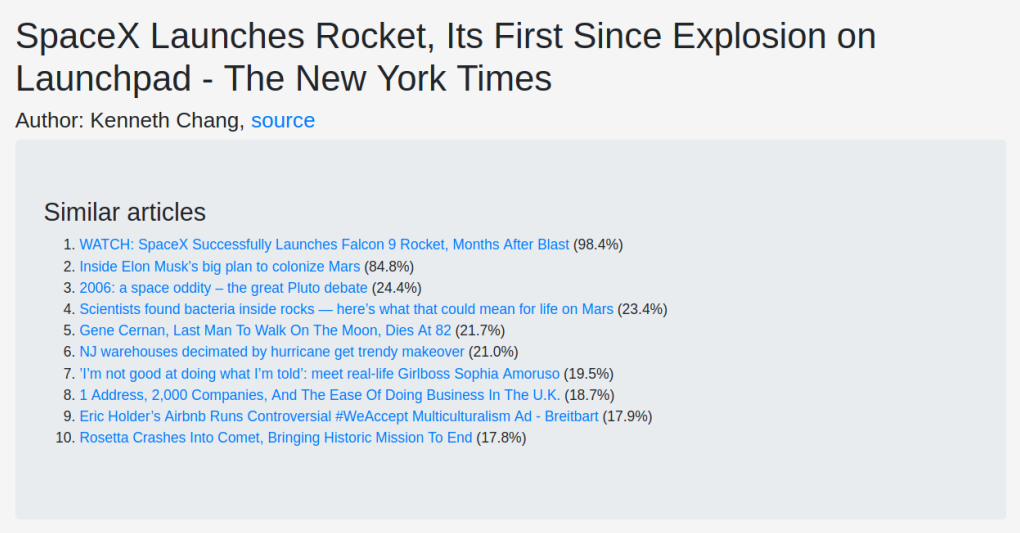
Ukládání dat v projektu řešíme přes CSV soubory, ke kterým přistupujeme přes pandas funkce. V jednom souboru máme uložený dataset nad kterým provádíme LSI. V dalších souborech pak máme uložené matice M_w, M_{cd} a M_q, abychom je mohli cachovat a přepočítavat jen při změně LSI parametrů, které máme uložené v souboru *server/lsa_config.json*.

Procházení článků vizualizujeme v prohlížeči pomocí *Flask*[6] web serveru. Jako dataset v našem projektu používáme anglicky psané novinové články stažené z *kaggle.com*[7]. Dataset nepoužíváme celý, vybrali jsme z něj pouze 996 článků.

4 Příklad výstupu

Na obrázku 4.1 je vidět konkrétní vstup a výstup naší aplikace. Zobrazí se název článku dotazu, jméno jeho autora a také samotný text článku. Naše aplikace dále uživateli nabídne seznam 10

[3] `https://www.nltk.org/`
[4] `https://pandas.pydata.org/`
[5] `https://numpy.org/`
[6] `https://flask.palletsprojects.com/en/1.1.x/`
[7] `https://www.kaggle.com/snapcrack/all-the-news`

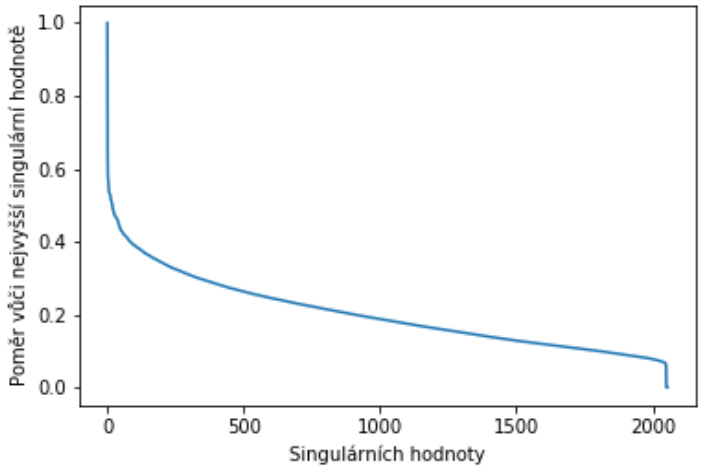


Obrázek 4.1: Příklad výstupu aplikace

nejpodobnějších článků i s určenou kosinovou převedenou na procenta. Je vidět, že aplikace vrací víceméně přesně to, co bychom čekali. Jako nejpodobnější vrátila téměř shodný článek taktéž informující, že firma SpaceX vypustila do vesmíru raketu. Další navrhované články se také týkají firmy SpaceX, vesmíru nebo businessu.

5 Experimentální sekce

5.1 Určení optimálního počtu konceptů



Obrázek 5.1: Důležitost konceptů

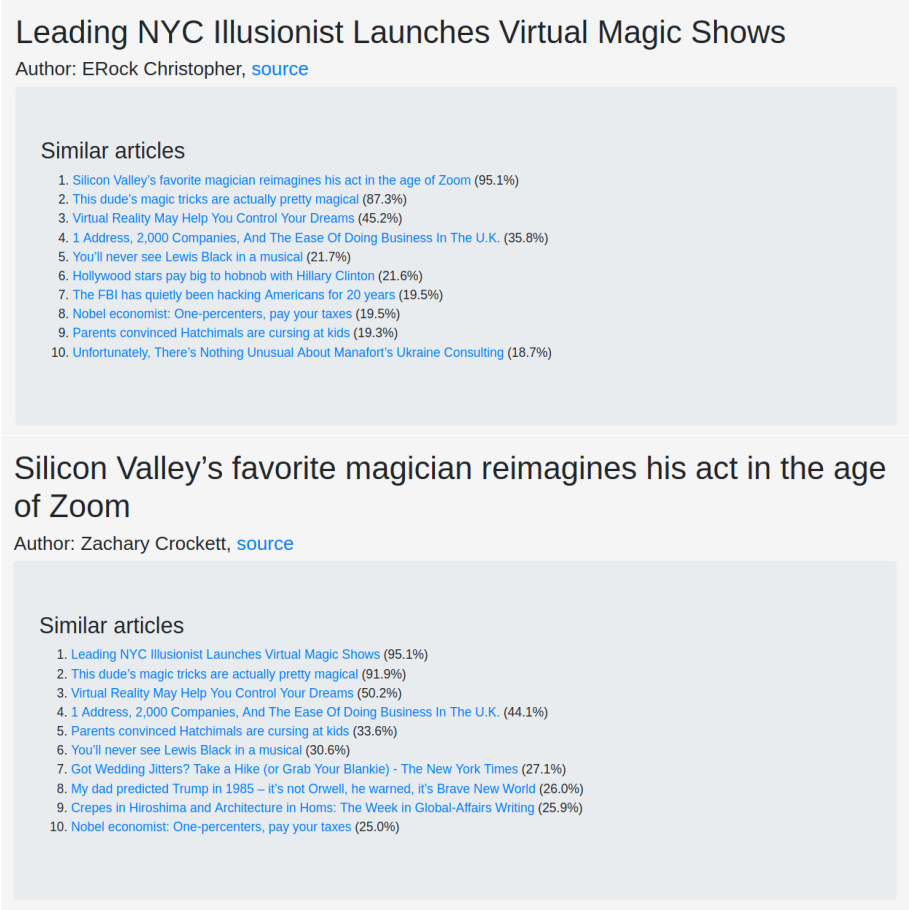
Vezmeme si singulární hodnoty, které nám vrátil singulární rozklad v rovnici 2.5. Vizualizujeme-li si v grafu 5.1, jak klesá poměr nejvyšší singulární hodnoty vůči zbylým singulárním hodnotám, vidíme z toho také, jak klesá důležitost konceptů v datasetu. Počet konceptů k v naší aplikaci tedy podle tohoto grafu určíme jako 200.

5.2 Porovnání vlivu LSI na kvalitu výsledků vyhledávání s ohledem na výskyt synonym a homonym

Pro zjištění kvality výsledků vyhledávání s ohledem na výskyt synonym a homonym jsme do datasetu přidali 4 články.

5.2.1 Synonyma

Jak si náš model poradí se synonymy jsme testovali na článcích *"Silicon Valley's favorite magician reimagines his act in the age of Zoom"* a *"Leading NYC Illusionist Launches Virtual Magic Shows"*. Testovali jsme tedy anglická synonyma *illusionist* a *magician*.



Obrázek 5.2: Synonyma

Na obrázku 5.2 je vidět, že náš model synonyma v článcích identifikoval správně. V prvním článku se sice ani jednou přímo nevyskytuje slovo *magician* a v druhém ani jednou slovo *illusionist*, ale v obou se vyskytují slova se slovním základem *magic* nebo například slova *show* a *audience* a model tak články správně identifikoval jako podobné.

5.2.2 Homonyma

K testování jak si náš článek poradí s homonymy jsme použili články *"Little Richard, rock 'n' roll pioneer, has died at 87"* a *"Scientists found bacteria inside rocks — here's what that could mean for life on Mars"*, které oba obsahují slovo *rock* ovšem jednou ve významu skály a jednou ve významu rockového muzikanta.

Obrázek 5.3: Homonyma

Na obrázku 5.3 je vidět, že u prvního článku, týkajícího se vlivu objevu bakterie v podmořských skalách na možnost života na Marsu, si model poradil opravdu dobře a vrátil nám články týkající se vesmíru nebo přírody.

U druhého článku týkajícího se smrti rockového hudebníka nám model také velmi správně správně jako nejpodobnější vrátil články týkající se dalších mrtvých hudebníků. Dále vrátil další relevantní články o hudbě a celebritách.

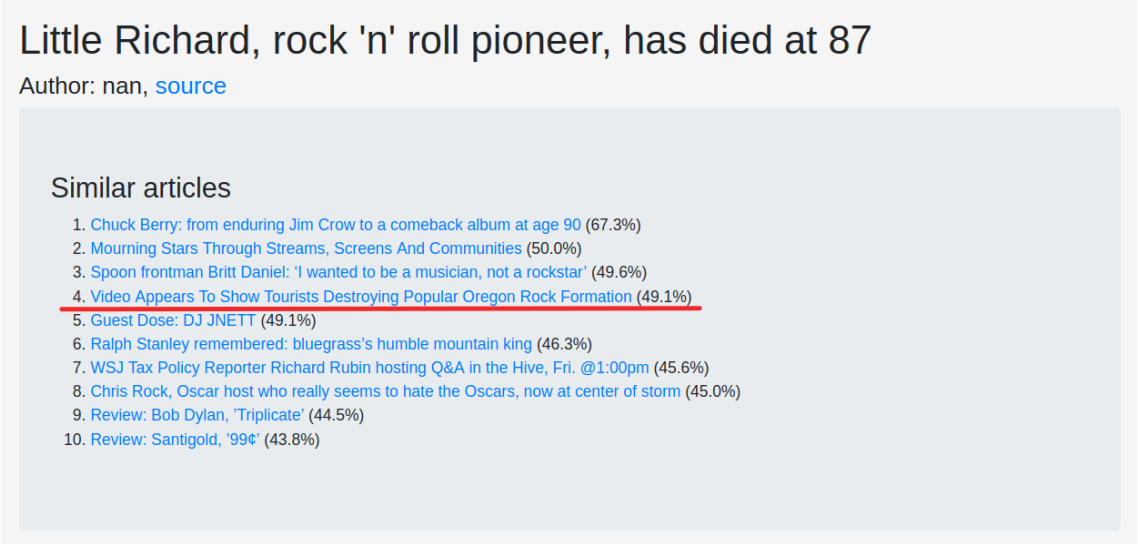
5.3 Porovnání průchodu pomocí LSI vektorového modelu se sekvenčním průchodem databáze s ohledem na čas vykonání dotazu

Pro odsimulování sekvenčního průchodu databází nastavíme počet konceptů stejný jako počet dokumentů databáze.

Na obrázku 5.4 je vidět, že při sekvenčním průchodu databází se čas vykonání dotazu zpomalí asi o 0.05 sekund. Při testování v prohlížeči je však rozdíl v rychlosti načítání stránky zobrazující obsah článku a 10 jemu nejpodobnějších článku mnohem výraznější, kolem 4 sekund.

```
In [21]: df_concept_by_doc, df_query_projection = transform_to_concept_space(df_tf_idf, k=0, customSVD=False)

         start = time.time()
         best_match = get_n_nearest(df_tf_idf,df_concept_by_doc, df_query_projection, 999, n=10)
         end = time.time()
         time_seq = end - start
         print("Čas vykonání dotazu při sekvenčním průchodu databází: " + str(time_seq) + " sekund")

         Čas vykonání dotazu při sekvenčním průchodu databází: 0.19510102272033691 sekund

In [22]: df_concept_by_doc, df_query_projection = transform_to_concept_space(df_tf_idf, k=200, customSVD=False)

         start = time.time()
         best_match = get_n_nearest(df_tf_idf,df_concept_by_doc, df_query_projection, 999, n=10)
         end = time.time()
         time_norm = end - start
         print("Čas vykonání dotazu při průchodu databází pomocí LSI vektorového modelu: " + str(time_norm) + " sekund")

         Čas vykonání dotazu při průchodu databází pomocí LSI vektorového modelu: 0.14609956741333008 sekund

In [23]: print("Rozdíl: " + str(time_seq - time_norm))

         Rozdíl: 0.049001455307006836
```

Obrázek 5.4: Porovnání času vykonání dotazu při sekvenčním průchodu databáze



Obrázek 5.5: Špatné určení článku obsahujícího homonymum jako podobného

5.4 Vliv různých vnitřních parametrů na výkon algoritmu (změna počtu konceptů, změna počtu extrahovaných termů, použití lemmatizace namísto stemmingu, odstranění číslovek při preprocesingu, použití jiného vzorce na výpočet vah termů)

5.4.1 Změna počtu konceptů

Při snížení počtu konceptů na 100 vrací model stále relevantní výsledky a také určování homonym a synonym funguje správně. Na obrázku 5.5 je vidět, že při zvýšení počtu konceptů na 300 už model špatně určí jeden článek obsahující homonymum jako podobný. Tato chyba je se zvyšujícím se počtem konceptů čím dál častější.

5.4.2 Změna počtu extrahovaných termů

V naší aplikaci ve funkci na redukci termů (o které píšeme v části 2.2) odstraňujeme pomocí parametru *min_df* pouze termy vyskytující se pouze v jednom dokumentu. Parametr *max_df* necháváme defultně nastavený na 1, po preprocesingu nemáme v našem datasetu termy vysky-

tující se v hodně dokumentech. Parametr *max_terms* také necháváme defaultně nastavený na 0 (tedy bez omezení na konkrétní počet termů). Upřednostňujeme nechat si v datasetu všechny důležité termy a mít přesnější výsledky před rychlejší dobou běhu.

5.4.3 Použití lemmatizace namísto stemmingu

```
In [72]: start = time.time()
         df_words_stemmed = preprocess_docs(df_data, use_lemmatizer=False, remove_numbers=True)
         end = time.time()

         print("Čas preprocesingu s použitím stemmingu: " + str(end-start) + " sekund")

         Čas preprocesingu s použitím stemmingu: 7.0390825271606445 sekund

In [73]: start = time.time()
         df_words_lemmatized = preprocess_docs(df_data, use_lemmatizer=True, remove_numbers=True)
         end = time.time()

         print("Čas preprocesingu s použitím lemmatizace: " + str(end-start) + " sekund")

         Čas preprocesingu s použitím lemmatizace: 29.658216953277588 sekund
```

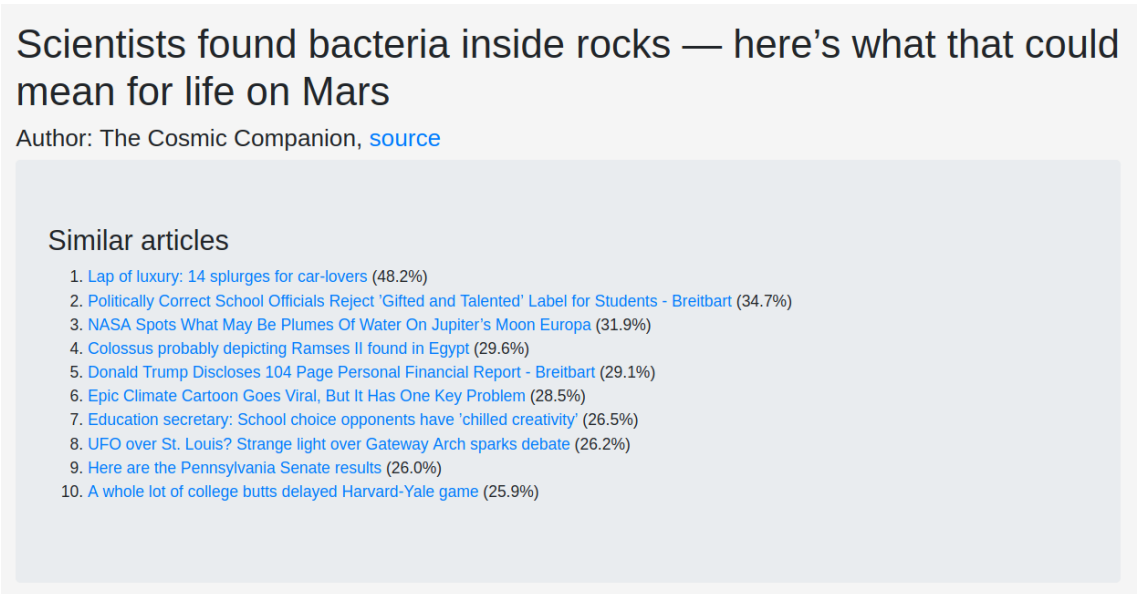
Obrázek 5.6: Čas preprocesingu při použití stemmingu versus při použití lemmatizace

Na obrázku 5.6 je vidět, že při použití lemmatizace namísto stemmingu se čas preprocesingu více než zčtyřnásobí na téměř 30 sekund. Uděláme-li si graf důležitosti konceptů jako na obrázku 5.1 vyjde při použití stemmingu i lemmatizace úplně stejný. V naší aplikaci jsme se rozhodli používat lemmatizaci a upřednostnit tak přesnější převádění slov z dokumentů na jednotlivé termy na úkor delšího času preprocesingu.

5.4.4 Odstranění číslovek při preprocesingu

Číslovky z dokumentů jsme se v naší aplikaci rozhodli odstraňovat, jelikož nám pak aplikace vrací o něco lepší výstupy. Číslovky v našem datasetu novinových článků nejsou většinou pro podobnost moc relevantní spíše naopak.

5.4.5 Použití jiného vzorce pro výpočet vah termů



Obrázek 5.7: Zhoršení výstupů při použití jiné metody normalizace četnosti termů

V aplikaci jsme se rozhodli při výpočtu *tf_idf* vah termů počítat normalizovanou četnost termů podle vzorce 2.2. Aplikace lze však pomocí konfiguračního souboru *server/lsa_config.json* nastavit, aby četnost termů počítala podle vzorce 2.3, tak jak je uveden v přednáškových slidech. Zjistili jsme, že při této změně má aplikace výrazně horší výstupy, jak je vidět na obrázku 5.7.

6 Diskuze

Aplikaci by určitě stálo za to vyzkoušet na větším datasetu obsahujícím více než 1000 článků. V reálné aplikaci by se také pro ukládání dat používala databáze, my jsme je však z důvodu zjednodušení ukládali do CSV souborů. Bylo by také zajímavé porovnat výsledky, které vrací náš LSI vektorový model, s nějakým jiným modelem (boolským, vektorovým...).

7 Závěr

V této práci jsme naimplementovali LSI vektorový model pro podobnostní vyhledávání v kolekci textových dokumentů. Model vrací relevantní výsledky. Práce pro nás byla přínosem, seznámili jsme se díky ní s velmi zajímavou problematikou prakticky využívající lineární algebru, objevili jsme jak velké možnosti nabízí jazyk Python pro práci s přirozeným jazykem a vyzkoušeli jsme si práci v týmu na větším projektu.