FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Název:	Systém pro automatické vytváření profilů expertů, na základě podobností jejích VaVaI výsledků
Student:	Maxim Sachok
Vedoucí:	Ing. Stanislav Kuznetsov
Studijní program:	Informatika
Studijní obor:	Webové a softwarové inženýrství
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	Do konce letního semestru 2020/21

Pokyny pro vypracování

1. Proveďte rešerši současných přístupů a řešení v dané oblasti.
2. Proveďte analýzu požadavku na systém a popište použité metody.
3. Navrhněte diagramy databáze, API a všech modulů systému.
4. Implementujte všechny požadavky z části analýzy.
5. Proveďte testování aplikace/API.
6. Proveďte nasazení výsledné aplikace na testovací server.
7. Aplikaci a její API řádně zdokumentujte.

Seznam odborné literatury

Dodá vedoucí práce.

Ing. Michal Valenta, Ph.D.
vedoucí katedry

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
děkan

V Praze dne 26. ledna 2020

FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Bakalářská práce

Systém pro automatické vytváření profilů expertů, na základě podobností jejích VaVal výsledků

Maxim Sachok

Katedra Softwarového Inženýrství
Vedoucí práce: Ing. Stanislav Kuznetsov

3. června 2020

Poděkování

Chtěl bych poděkovat vedoucímu za to, že mi pomohl porozumět strojovému učení a analýze textu.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principu při přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisu. V souladu s ust. § 2373 odst. 2 zákona č. 89/2012 Sb., občanský zákoník, ve znění pozdějších předpisu, tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programu, jež jsou její součástí či přílohou a veškeré jejich dokumentace (dále souhrnně jen „Dílo"), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli zpusobem, který nesnižuje hodnotu Díla a za jakýmkoli účelem (včetně užití k výdělečným účelum). Toto oprávnění je časově, teritoriálně i množstevně neomezené. Každá osoba, která využije výše uvedenou licenci, se však zavazuje udělit ke každému dílu, které vznikne (byť jen zčásti) na základě Díla, úpravou Díla, spojením Díla s jiným dílem, zařazením Díla do díla souborného či zpracováním Díla (včetně překladu) licenci alespoň ve výše uvedeném rozsahu a zároveň zpřístupnit zdrojový kód takového díla alespoň srovnatelným zpusobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Praze dne 3. června 2020 . . . . . .

Odkaz na tuto práci

Sachok, Maxim. *Systém pro automatické vytváření profilů expertů, na základě podobností jejích VaVal výsledků*. Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2020.

Abstrakt

Tato práce implementuje aplikaci pro snadné použití algoritmů k identifikaci autora textu. V tomto článku jsou porovnány různé způsoby identifikace autora, a to jak z hlediska přesnosti, tak z hlediska účinnosti. Tato aplikace používá REST jako prostředek komunikace s uživatelem.

Klíčová slova klasifikace textu, Java, Spring, autorství, strojové učeni

Abstract

This thesis implements an application for easy use of algorithms to identify the author of the text. Author compares different algorithms to identify an author, both in terms of accuracy and effectiveness. This application uses REST as a means of communication with the user.

Keywords text classification, Java, Spring, author attribution, machine learning

Obsah

Seznam obrázků

Seznam tabulek

Úvod

Účelem této práce je sestavit aplikaci, která by uživateli umožnila identifikovat autora tohoto textu bez znalosti strojového učení nebo textové analýzy. Toto téma bylo vybráno autorem, protože autor chtěl otestovat jeho schopnost porozumět věcem, o nichž nic neví, jako je strojové učení nebo analýza textu. Autor chtěl vyzkoušet jeho schopnost psát plnohodnotnou aplikaci od nuly, což bude užitečné. Tato práce je rozdělena do několika částí. Nejprve určíme, jak lze vzájemně porovnávat texty. Poté se podíváme na různé existující algoritmy a uvidíme, jak fungují. Následuje analýza požadavků na tuto aplikaci. Poté analyzujeme architekturu aplikace a použité knihovny. Dále implementujeme aplikace pomocí toho, co jsme se naučili z předchozích kapitol. Nakonec testujeme aplikaci na její výkon a vhodnost.

Kapitola 1

Cíl práce

Hlavním cílem této práce je implementace systému vyhledávání autora vědecké práce s využitím znalostí získaných během školení na univerzitě.

Tato práce může být rozdělena do několika částí: nejprve musím analyzovat problém, jakým způsobem lze porovnávat texty; pak musím porovnat a zvolit nejlepší způsob řešení problému; dalším krokem je analýza požadavků na tento program; poté následuje psaní kódu pro tento program; a na konci testování finálního produktu.

Kapitola 2

Analýza textu

Tato kapitola analyzuje různé možnosti porovnání několika textů a předběžné zpracování textu.

Existuje mnoho různých algoritmů pro analýzu textu, od algoritmů, které kategorizují text na základě předem napsaných pravidel, až po algoritmy, které používají strojové učení na základě předem klasifikovaného textu.

2.1 Čištění textu a předběžné zpracování

Pro klasifikaci textu je velmi důležité odstranit všechna zbytečná slova, která nenesou žádné sémantické zatížení. Tato slova jsou zastavovací slova, slova, která jsou chybně napsaná atd.

V mnoha algoritmech, jako jsou statistické a pravděpodobnostní metody učení, může šum a zbytečné vlastnosti negativně ovlivnit celkový výkon. Odstranění těchto funkcí je tedy nesmírně důležité. [1] (přeloženo autorem)

2.1.1 Tokenizace

Tokenizace je proces rozdělení textu na samostatná slova, fráze, znaky nebo jakýkoli jiný prvek v závislosti na úkolu. Například:

```
The cat is cat.
{
"The", "cat", "is", "cat"
}
```

V mnoha jazycích lze tokenizaci do slov použít s mezerami, protože ve většině jazyků slova nemají mezery samy o sobě. Problém začíná, když pracujeme s jazyky, které mají mezery ve slovech, nebo pokud chceme zachovat fráze které mají mezeru. K tomu potřebujeme předem vytvořený slovník pro porozumění existujícím slovům nebo frázím v daném jazyce. V tomto článku používám anglický text.

2.1.2 Stop slova

Je téměř nemožné napsat text bez použití slov bez sémantického zatížení pro daný text. Musíme taková slova z textu odstranit, abychom text co nejlépe klasifikovali. Například:

```
The cat is cat.
Without stop words: cat cat.
```

2.1.3 Kapitalizace

Každý text se skládá z vět, které zase obsahují velká a malá písmena. To přináší problém při porovnávání slov, protože některé algoritmy mohou počítat stejné slovo jako dvě různé, jediným rozdílem je, že jedno slovo mělo první velké písmeno. Nejlehčím způsobem je omezit vše na malá písmena. Tato metoda zase přináší problém, že některá jména nebo zkratky, pokud jsou přeloženy na malá písmena, se stanou zastavovacími slovy: US us. Řešením tohoto problému je použití úplných jmen, kdykoli je to možné.

2.1.4 Odstranění hluku

Téměř každý text má nějaké speciální znaky, čísla, bílé znaky, interpunkci. To vše zavádí hluk do analýzy textu, protože tyto znaky přinášejí algoritmu více dvojznačnosti. Ano, interpunkce je důležitá pro lidské porozumění textu, ale je to nepřítel algoritmů.

2.1.5 Stemming

Text Stemming modifikuje slovo tak, aby získalo jeho základní nebo kořenovou formu. Například ”studying”"study”. Tento algoritmus není výhodný v tom, že může odříznout příliš mnoho nebo naopak ponechat příliš mnoho.

2.1.6 Lemmatization

Lemmatizace je algoritmus pokročilejší než Stemming, bere v úvahu lemma slova ve slovníku. To zahrnuje překlad slov do jejich slovní formy. Ve skutečnosti je lemma slova jeho slovní nebo kanonickou formou! [2] (přeloženo autorem)

2.2 Uložení textu

Jak může počítač porovnat dva texty? Nejjednodušší způsob je vzít slovo z každého textu a porovnáme mezi sebou, porovnáme je až do konce, dokud slova nezůstanou. Pokud jsou všechna slova stejná, pak jsou tyto dva texty stejné. Nyní můžeme zjistit, zda jsou texty stejné nebo ne. Jak ale můžeme

zjistit, zda dva texty mluví o stejné věci. Abychom to mohli udělat, musíme se naučit ukládat text do paměti.

Nejpoužívanější model je - Bag of Words. [3] Tento model ukládá slova bez ohledu na jejich pád. Například jedno slovo v různých pádech se bude v tomto modelu považováno za různá. Tento model tvoří slovník všech slov. U slovníku velikosti N je každé slovo reprezentováno N - dimenzionálním vektorem s 1 v indexu odpovídajícím slovu a 0 v každém dalším indexu.

```
The cat is hiding under the car.

{
  "The",
  "cat",
  "is",
  "hiding",
  "under",
  "car"
}
```

Takto bude text vypadat pomocí tohoto slovníku.

Tabulka 2.1: Reprezentace textu v modelu

Text	The	cat	is	hiding	under	car
The cat is cat	1	2	1	0	0	0

Pomocí tohoto modelu lze každý text přeložit na vektor stejné velikosti a tyto vektory porovnat. Tento model může ukládat n - gramy, kde n je počet slov v n - gramu. Díky tomu lze ukládat nejen jednotlivá slova, ale i celé fráze. Uložení frází umožní uložit sekvenci slov, což umožní vidět vzor v tomto textu, a pomocí tohoto lze sestavit profil často používaných frází autora.

2.3 Analýza důležitosti slova - TF-IDF

K analýze důležitosti slova použijeme tuto metodu TF-IDF. Co znamená míra opakování slov v daném textu a inverzní míra opakování slov ve všech ostatních textech. Pomocí této metody lze velmi dobře najít klíčová slova v textu nebo udělat krátký popis výběrem vět s nejvyšším významem. Tuto metodu lze snadno implementovat a ona je dostatečně rychlá.

Tato metoda umožňuje normalizovat hodnotu slov tak, aby hodnota často se opakujícího slova byla mnohem menší než hodnota slova, která se opakuje méně často. K tomu můžeme použít vektor, který jsme získali z modelu v předchozí sekci.

Tato metoda funguje podle vzorce:

tf = word frequency in text
N = total number of documents
C = number of documents containing the word
idf = $\log_2 \frac{N}{(C+1)}$
$weight = tf \times idf$

Pomocí tohoto vzorce lze snadno vypočítat váhu slova ve vztahu k danému textu a ke všem ostatním textům a můžeme normalizovat náš vektor z předchozí sekce. Tento vzorec je jen jedním z několika.

2.4 Porovnání textu

Nyní máme text, pomocí kterého jsme vytvořili slovník, přeložili tyto texty do vektorů a normalizovali jejich hodnoty. Nyní musíme tyto vektory porovnat. Přímé srovnání nám pouze řekne, zda mají tyto texty stejná slova nebo ne. Potřebujeme vědět, jak moc jsou podobné. K tomu můžeme použít různé metody pro porovnávání vektorů.

2.4.1 Jaccard

Jaccardova podobnost nebo průnik nad spojením je definována jako velikost průniku dělená velikostí spojení dvou sad. Abychom mohli tuto podobnost využít, potřebujeme v našem textu provést lemmatizaci. Lemmatizace redukuje slovo na jeho základní formu.

1: This cat have four legs.
2: This car had four wheels.
1∪2 = $This, car, cat, has, four, leg, wheel$
$1 \cap 2 = This, has, four$
$Jaccard = \frac{|1 \cap 2|}{|1 \cup 2|} = \frac{3}{7}$

V této situaci jsme dostali výsledek $\frac{3}{7}$. Mluvíme o různých věcech, ale máme dostatečný počet stejných slov. Podívejme se na další příklad.

1. Zeman comes to country center to give a speach.
2. Czech president greets press in Prague.
1∪2 = $Zeman, come, to, country, center, give, a, speach, Czech, president, greet, press, in, Prague$
$1 \cap 2 =$
$Jaccard = \frac{|1 \cap 2|}{|1 \cup 2|} = 0$

Zde mluvíme o téměř stejné věci, ale tyto dva texty nemají žádné stejná slova. Tato metoda nebere v úvahu sémantický význam slova v textu.

Hlavním problémem této metody je to, když dva texty mluví o různých věcech, čím větší je velikost textu, tím vyšší je šance na výskyt stejných slov.

2.4.2 Kosinová podobnost

Kosinová podobnost počítá podobnost měřením kosinusového úhlu mezi dvěma vektory. Tato metoda umožňuje porovnat dva vektory navzájem takovým způsobem, abychom viděli, jak moc jsou podobné nebo jak se liší.

Metoda vezme kosinus mezi dvěma vektory, což nakonec přinese hodnotu mezi -1 a 1, kde 1 znamená, že vektory jsou identické a -1 že jsou opačné. Kosinová podobnost je výhodná v tom, že i když jsou vektory daleko od sebe, mohou být stále orientovány v jednom směru a tato metoda může poskytnout poměrně vysoký výsledek.

Nejlepší způsob, jak použít tuto metodu, je použít předem trénované modely, které ukládají celý vektor pro každé slovo. S tímto můžeme porovnat například slova, která mají stejný význam, ale jsou psána odlišně: auto a vozidlo. Pomocí těchto modelů lze najít nejbližší slova podle významu. Google vytvořil takový model slov pomocí svých zpráv shromážděných v průběhu několika let. [4]

2.4.3 Naive Bayes Classifier

Naive Bayes je rodina statistických algoritmů, které můžeme využít při klasifikaci textu. Jedním z členů této rodiny je Multinomial Naive Bayes (MNB). Jednou z jeho hlavních výhod je to, že můžete dosáhnout opravdu dobrých výsledků, když dostupná data nejsou moc (ñěkolik tisíc označených vzorků) a výpočetní zdroje jsou vzácné. [5] (přeloženo autorem)

Tato metoda je založena na Bayesově větě. Pomocí teto věty můžeme rychle klasifikovat text na základě existujících textů a kategorií, můžeme zjistit, jaká je šance, že autor napsal tento text na základě již napsaných textů.

2.4.4 Support Vector Machines

Support Vector Machines (SVM) - toto je další z mnoha algoritmů, které jsou velmi dobré pro klasifikaci textu. Stejně jako předchozí algoritmus nepotřebuje pro trénink mnoho dat, ale vyžaduje mnohem větší výpočetní sílu. Funguje tak, že rozděluje prostor na dva podprostory, takže mezera mezi nimi je největší, a poté, když aplikujeme SVM na neznámý text, rozhodne, do které ze dvou kategorií bude text zahrnout na základě nejmenší vzdálenosti k nejbližšímu prostoru.

2.4.5 KNN

K Nearest Neighbors - je to jednoduchý algoritmus, který bere předem kategorizovaná data v prostoru a pro každý nový text, který nemá kategorii, najde nejpravděpodobnější kategorii inspekcí nejbližších sousedů daného textu v prostoru. Pokud tedy například vezme 20 nejbližších sousedů a 15 z nich patří do kategorie 1 a zbytek, například do kategorií 2, 3, 4, klasifikuje text jako kategorii 1.

Kapitola 3

Analýza požadavku

Než začnete psát jakýkoli software. je nezbytné si naplánovat, jaké cíle tento program sleduje, co by měl dělat, jaké metody nebo jiné programy by měl používat. S tímto plánem můžeme sledovat, co tato aplikace již ví, jak dělat, co nikdy neudělá, co k ní může v budoucnu přidat. Vypracování takového plánu pomáhá vyhnout se většině chyb při práci na aplikaci a pomáhá při práci s zákazníkem.

3.1 Cíle požadavku

Při práci s klientem je velmi důležité zaznamenávat žádosti, protože s pomocí klienta i klienta, který tuto aplikaci napíše, může na konci zkontrolovat, zda aplikace vyhovuje tomu, co bylo dříve dohodnuto. Ve světě softwaru se to nazývá akceptační testování.

I když aplikaci vytvoříme pro vlastní potřebu, stále hodně pomáhá vytvořit seznam cílů, které by tato aplikace měla splnit, protože často můžeme zapomenout na to, co jsme chtěli na začátku vývoje, a do konce jít na úplně jiný cíl. Je důležité si uvědomit, že je lepší zpočátku sestavit a provést plán, a teprve poté můžete přidat funkčnost k již napsané aplikaci, protože neustálé rozšiřování plánu před koncem původního plánu může vést k dalším problémům.

3.2 Vlastnosti požadavku

Požadavky lze zkontrolovat podle následujících podmínek

- Pokud je lze prakticky realizovat
- Pokud jsou platné a podle funkčnosti a domény softwaru
- Pokud existují nejasnosti
- Pokud jsou kompletní
- Pokud je lze prokázat

Každý požadavek by měl být jasný jak pro zákazníka, tak pro toho, kdo tento požadavek splňuje. Každý požadavek musí mít prioritu, musí být ověřitelný, protože pokud není možné tento požadavek ověřit, pak si zákazník nebo ten, kdo jej splňuje, nemůže být jistý, že byl splněn. Každý požadavek musí být úplný, požadavek může být rozdělen do několika menších částí. Požadavek by neměl nic skrývat, měl by být psán co nejjasněji. Písemné požadavky by měly stačit k vytvoření kompletního programu.

3.3 Typy požadavků

Funkční požadavky - to jsou požadavky, které specifikují, že aplikace by měla být schopna, jak by měla komunikovat s osobou nebo s jinými aplikacemi, jak by měla vypadat. Tyto požadavky se používají ve fázi návrhu aplikace.

Nefunkční požadavky - jedná se o požadavky, které nelze přímo vyzkoušet, jako je rozšiřitelnost aplikace nebo aby aplikace fungovala rychle, aby byla aplikace stabilní, lze ji dále rozšiřovat.

3.4 Funkční požadavky

- Aplikace musí být přístupná prostřednictvím REST
  - Vyhledávaní autora textu
  - CRUD Operace
  - Možnost otestovaní přesností použitého algoritmu přes REST
- Ukládaní dát v DB

3.5 Nefunkční požadavky

- Aplikace musí být škálovatelná pro vetší rozsah dat
- Poměrné rychlá (záleží na použitém algoritmu)
- Používat co nejméně paměti

Kapitola 4

Návrh aplikace

Při psaní aplikace je velmi důležité porozumět tomu, jak bude použita, a pomocí této volby zvolit správnou architekturu pro danou úlohu. Pro tuto aplikaci, která vám umožní vytvářet, mazat, aktualizovat projekty a autory a přidávat projekty k autorům. Pro tyto úkoly je vhodná architektura microservice (SOA).

4.1 Výběr technologie

Jelikož vytváříme aplikaci pro mikroservisy, která bude k dispozici pomocí REST, je pro tuto roli nejvhodnější Spring Boot. Spring Boot usnadňuje vytváření samostatných produkčních aplikací založených na Spring, které můžete „jen spustit“. [6] Na rozdíl od Java EE, aplikace na Spring Boot vyžaduje velmi málo nastaveni.

- Aplikační server je už v aplikace (není potřeba nahrávat WAR)
- Automatické nastaveni
- Žádné XML nastaveni
- Usnadňuje tvorbu Java aplikací, Unit testu
- Není třeba psát hodně stejného kódu

Tím vychází ze jsme použijeme Java programovací jazyk.

4.1.1 Knihovna pro porovnaní textu

Většina knihoven používaných pro strojové učení je psána v jazyce Python. [7] Ale na druhém místě hned za ním je Java. První knihovna je DeepLearning4J Word2Vec. [8] V této knihovně již existuje veřejný model slov vytvořený společností Google. [4] Google v něm shromáždil 100 miliard anglických

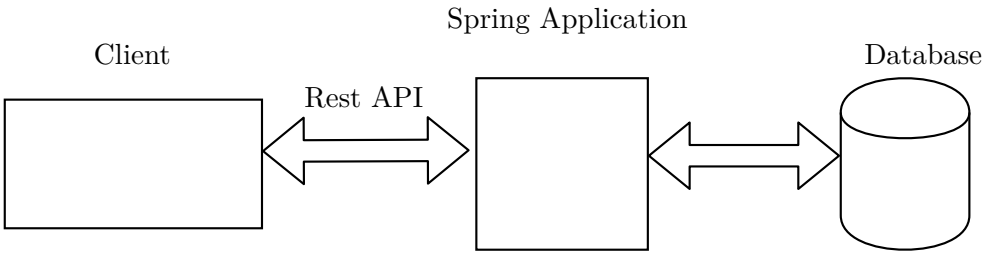
slov pomocí hlubokého učeni což by nebylo možné udělat pomoci vlastního počítače.

Druhá knihovna je WEKA. [9] Je to šikovný nastroj pro otestovaní klasifikátoru a pochopeni strojového učeni, v něm můžeme podívat se, jak různé algoritmy představují text v paměti. Tento software poskytuje API pro použiti jeho množiny funkci ve vlastni aplikace.

4.2 Komunikace

Veškerá komunikace mezi uživatelem a aplikací se provádí pomocí REST. Všechny aplikace, které vyhovují REST, se nazývají RESTful aplikace. REST poskytuje aplikacím příležitost používat jiné aplikace bez nutnosti stahovat a spouštět je na svém počítači.

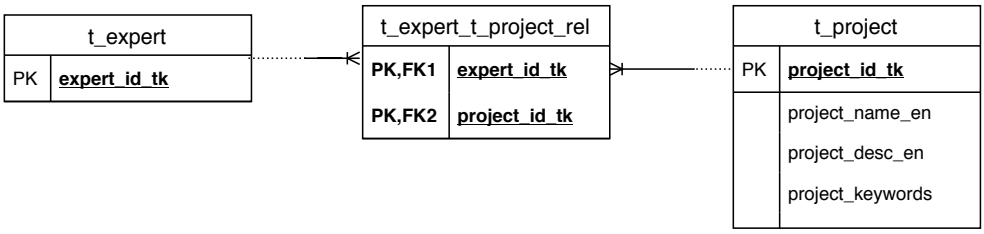
REST také umožňuje vytvářet složité systémy pomocí několika malých aplikací, které vytvářejí jednu konkrétní funkci a spolu komunikují pomocí odpočinku. Takže například kdokoli může použít Twitter API k vytvoření tweetbotu díky REST.



Obrázek 4.1: Diagram komunikace

Uživatel bude komunikovat s aplikací prostřednictvím REST, aplikace vezme data pro práci z databáze. Pro tuto práci využíváme údaje poskytnuté vedoucím mé práce.

V databázi budeme mít pouze projekty a autory.



Obrázek 4.2: Databázový diagram

Kapitola 5

Implementace

Pro psaní tohoto projektu použijeme IntelliJ IDEA. [10]

5.1 Vytváření projektu

Díky tomu, že používáme Spring Boot, bude vytvoření projektu velmi rychlé. Použijeme Spring Boot starter. [11] Pro vývoj potřebujeme závislosti: H2 - databáze v paměti; Web - poskytuje Tomcat server a REST; JPA - poskytuje moznost komunikace s databázi pomoci JPA. Použijeme verze Spring Boot 2.2.6 a Java 14 protože 14 verze je nova LTS. Než začneme psát aplikaci, zkontrolujeme, zda všechno funguje. Spustíme aplikaci a přejdeme na adresu **localhost:8080**. Tam bychom měli vidět chybovou zprávu: **There was an unexpected error (type=Not Found, status=404)**, což znamená, že vše funguje a aplikace přijímá požadavky.

5.2 Entity

Nyní musíme vytvořit entity pro přístup a práci s daty. Z databázového diagramu víme, že potřebujeme pouze dvě entity a jeden vztah mezi nimi. Vytváříme tedy autora, projekt a vztah mezi nimi. Pro účely anonymizace bychom měli odstranit všechna identifikační data od autora, takže jediné pole bude id generované databází. Pro projekt potřebujeme taky id generované databázi, jeho popis, název, a klíčová slova.

```
@Entity
@Table(name = "t_expert", schema = "semantic")
public class Author {
    @Id
    @Column(name = "expert_id_tk")
    @GeneratedValue(strategy = GenerationType.SEQUENCE)
    private Long expertidtk;

    @OneToMany(mappedBy = "author", fetch = FetchType.EAGER, cascade
        = CascadeType.REMOVE)
    private Set<AuthorProject> authorProjects;
    /*Setters and Getters*/
}
```

Generační strategie znamená, že db vygeneruje id, takže jej klient nemusí generovat, ale to znamená, že pokaždé, když klient vytvoří novou entitu, měli bychom ji vrátit, aby klient znal id nově vytvořené entity. Fetch type eager znamená, že pokud chceme načíst tuto entitu z databáze, Hibernate také načte entitu, se kterou má vztah anotovaný. Cascade type Remove znamená, že pokud tuto entitu odstraníme, vztah s ní spojený bude také odstraněn, tím se ušetří řádek kódu a nemusíme si to pamatovat.

Pro jejich vzájemný vztah vytváříme entitu, která by byla identifikována id autora i projektu. Držitelé vztahů jsou projekt a autor. To znamená, že pokud odstraníme autora nebo projekt, budou odstraněny i všechny relační entity, které obsahují jejich klíč.

Pro snadný přístup k datům a operace s nimi používáme JPARepository. Toto je rozhraní pro ukládání, vyhledávání, aktualizaci a odstraňování dat.

V ideálním případě by řadič neměl být logický, takže vytváříme službu pro každou entitu, aby s touto entitou prováděla operace.

Pro uživatele, kteří nám zasílají aktualizované entity nebo pro nás, abychom je odeslali, vytváříme DTO. Používá se pro přenos dat a skrytí vnitřní struktury. To také pomáhá s serializací (překlad objektu do řetězce JSON [12]), protože máme obousměrný vztah, takže by serializace byla v smyčce.

5.3 REST

Nyní můžeme vytvořit náš REST řadič pro komunikaci s uživatelem. Nejprve můžeme jednoduše přidat operace CRUD pro naše entity.

```
@RestController
public class Controller {

    private AuthorService authorService;
    private ProjectService projectService;
    @Autowired
    public Controller(AuthorService authorService, ProjectService
        projectService) {
        this.authorService = authorService;
        this.projectService = projectService;
    }

    @GetMapping("/author")
    public ResponseEntity<?> getAuthors(){
        List<AuthorDto> authorDtoList = new ArrayList<>();
        for(Author author : authorService.getAuthors()){
            AuthorDto authorDto = new AuthorDto(author.getExpertidtk());
            authorDtoList.add(authorDto);
        }
        return ResponseEntity.ok(authorDtoList);
    }

    @GetMapping("/project")
    public ResponseEntity<?> getProjects(){
        List<ProjectDto> projectDtoList = new ArrayList<>();
        for(Project project : projectService.getProjects()){
            ProjectDto projectDto = new ProjectDto();
            projectDto.setNameEn(project.getNameEn());
            projectDto.setKeywords(project.getKeywords());
            projectDto.setDescEn(project.getDescEn());
            projectDto.setId(project.getProjectIdTk());
            projectDtoList.add(projectDto);
        }
        return ResponseEntity.ok(projectDtoList);
    }

    @GetMapping("/author/{id}")
    public ResponseEntity<?> getAuthor(@PathVariable Long id){
        if(authorService.getAuthor(id).isPresent()){
            return new ResponseEntity<>(new
                AuthorDto(authorService.getAuthor(id).get().getExpertidtk()),
                HttpStatus.OK);
        }
        return new ResponseEntity<>(HttpStatus.NOT_FOUND);
    }
```

Zde vkládáme repozitáře, abychom mohli pracovat s entitami. Když uživatel odešle požadavek GET na adresu namapovanou na funkci, funkce se volá. Takže předtím, než se pohneme kupředu, měli bychom vyzkoušet, že jsme vše napsali správně. Spustíme aplikaci a pošleme žádost GET na adresu kterou jsme namapovali **localhost:8080/author**, měli bychom získat seznam všech autorů v databázi.

Nyní můžeme přidávat funkce pro aktualizaci, mazání nebo přidávání projektů autorům. Byly by zmapovány respektive jako požadavky PUT, POST nebo DELETE. Přidáme další požadavek, pomocí kterého můžeme vyzkoušet přesnost použitého algoritmu.

5.4 Analýza textu

Zbývá pouze napsat třídy pro zpracování textu, filtrování a analýzu.

5.4.1 Zpracování

Nejjednodušší součástí je zpracování textu. Protože v této práci používáme anglický text, jednoduše rozdělujeme text na slova pomocí mezer a ukládáme do mapy kde počítáme kolikrát se slovo v textu opakuje. Převedeme všechno na mala písmena.

5.4.2 Filtrování

Vymažeme vše, co není písmenem nebo interpunkčními znaménky obsaženými ve slovech anglického jazyka, to znamená také všechna čísla. Zredukujeme slova na jejich základní formu.

5.5 Testovaní naivních algoritmu

Nemůžeme použít mnoho různých testů, pokud je většina z nich nepřesná. V této části testujeme různé algoritmy popsané na začátku této práce. Testování bylo provedeno tímto způsobem. Bylo vybráno 60 autorů, kteří měli 2 nebo více projektů. Měli jeden projekt odstraněn a algoritmus měl zjistit, který autor patří ke kterému projektu. Pokud je autor spoluautorem projektu, pro který v současné době hledáme autora, bude tento autor z vyhledávání odstraněn.

5.5.1 Jaccard

Pro tento algoritmus stačí rozdělit text na slova a vypočítat poměr běžných slov ke všem slovům. Tento algoritmus zabírá přesně tolik paměti jako všechny autorovy projekty, se kterými v současné době porovnáváme. Porovnání s jedním autorem trvalo přibližně 14 milisekund. Přesnost tohoto algoritmu byla

0. Použití lemmatizace nepomohlo. Tento algoritmus nemohl najít žádného autora.

5.5.2 Kosinová podobnost

Pro tento algoritmus vypočítáme TF-IDF pro každý text a zjistíme kosinus úhlu mezi vektory. Čas pro každého autora je stejný jako u Ja card, protože musíme projít všechny texty a spočítat počet opakování slova. Přesnost tohoto algoritmu byla 3 procent. Zde lemmatizace snížila přesnost algoritmu do 1.5 procent.

5.5.3 Word2Vec Google Model

Vidíme, že algoritmy, které berou v úvahu počet opakování slova, jsou velmi nepřesné. Zde používáme model napsaný společností Google, který má sémantický význam pro každé slovo jako vektor 300d. Algoritmus pro tento model je velmi jednoduchý, vezmeme vektor pro každé slovo, sčítáme všechny vektory a vydělíme počtem slov, čímž získáme průměrnou hodnotu, kde je náš text v prostoru 300d. Tento algoritmus vyžaduje stažení 3,5 gigabajtového modelu, doba načítaní aplikace tedy není 5 sekund, ale 20 minut. Navíc neustále využíváme 3,5 gigabajtů paměti RAM. Srovnávací doba pro každého autora na úrovni několika milisekund. Přesnost tohoto algoritmu je něco přes 8 procent. Lemmatizace pro tento algoritmus není nutná, protože tento model rozpoznává syntaktický význam slova.

5.6 Testování algoritmu strojového učení

Jak vidíme naivní algoritmy, které porovnávají počet běžných slov, které počítají počet opakování, nebo algoritmus, který jsme vytvořili pomocí syntaktického modelu, nefungují.

Z toho všeho můžeme usoudit, že pro tento problém nemusíme hledat syntaktický význam textu ani to, co tento text říká. Hlavní věc pro nás je najít vzorec v psaní textu konkrétním autorem a hledat tento vzor v textu, pro kterého chceme autora najít. Tento vzorec je zničen lemmatizací. Například v angličtině tato dvě slova znamenají totéž, ale jsou psána odlišně: „has not“, „hasn't“. Lemmatizace převede tyto dva slovy na: „has not“, „has“. Což vytvoří jedno slovo navíc a zničí pravě to co napsal autor.

Pro zjednodušení práce na těchto algoritmech byli všichni autoři rozděleni do skupin po 2. Tyto skupiny byly následně sestaveny do skupin po 12 skupinách. Tento proces se opakuje, dokud nezůstane jeden autor. To si lze představit jako turnajovou tabulku. Můžeme tedy distribuovat zatížení na 12 vláken. A nemusíme sestavovat klasifikátor okamžitě pro všechny autory. Vzhledem k tomu, že používáme n-gramy, většina slov se bude několikrát brát v různých frázích, ale máme omezené množství paměti RAM. Tato distribuce

nejen snižuje využití RAM, ale také zrychluje práci. Navíc, čím více autorů v jednom klasifikátoru, tím déle trvá jeho kompilace. Pokud máme klasifikátor pro všechny autory, pak při přidávání nebo odebírání jakéhokoli autora nebo projektu musíme klasifikátor znovu sestavit. Tato metoda kompilace klasifikátoru pro dva autory tuto potřebu odstraní.

5.6.1 Naive Bayes Classifier

Používáme textový klasifikátor, který je již implementován v WEKA. Výhodou této metody je, že pro použití jiného klasifikátoru stačí změnit pouze jeden řádek v kódu, namísto psaní zcela nové třídy pro každý klasifikátor. Výhodou tohoto algoritmu je, že bere v úvahu pouze ta slova, která jsou v textu. Čas na vyhledání autora textu pro tento algoritmus v databázi (500 autoru) v projektu je 4 sekundy. Jeho přesnost byla 22 procent.

5.6.2 KNN

Tento algoritmus má přesnost 3 procenta. Tento algoritmus zřejmě nejvíce ovlivňuje malé množství dat. Žádný autor nemá tisíce vědeckých prací, s nimiž můžeme zlepšit přesnost tohoto algoritmu. Čas na nalezení jednoho autora je 350 milisekund.

5.6.3 LibLINEAR

Tento algoritmus [13] má největší výhody z rozdělení autorů do skupin po 2, protože používá SVM. Navíc používá regresi. Bez rozdělení autorů do skupin tento algoritmus vyžadoval mnohem více než dostupných 8 gigabajtů paměti RAM (před snížením objemu dat z 20 000 autorů na 500 pro urychlení testování). I pro 500 autorů byla doba kompilace, při použití n-gramu velikosti 1 až 3, 2 hodiny a 40 minut. Pokud by tedy byla databáze aktualizována každé 2 hodiny, tato aplikace by byla k ničemu. Při dělení do skupin 2 autorů takový problém neexistuje. Přesnost tohoto algoritmu byla 37 procent. Tento algoritmus je mnohem rychlejší než ostatní, když se používá rozdělení do skupin po dvou. Nejpravděpodobnějšího autora najde za 4 sekundy. Používá kolem 1 gigabajtu paměti RAM.

Kapitola 6

Testovaní

Testování je velmi důležitou součástí programování. pomocí testování můžete sledovat správnou činnost programu po jakékoli změně. Osoba může zapomenout, co by funkce nebo konkrétní část kódu měla dělat. Ale test napsaný včas a správně řeší tento problém okamžitě. Během provádění této práce když byla napsaná nova funkce byly do ní okamžitě napsané testy.

6.1 Smoke Testy

Tyto testy jsou velmi jednoduché testy, které identifikují vážné chyby, s nimiž program nemůže být předán uživateli. V našem případě zkontrolujeme, že Spring Boot aplikoval všechny závislosti.

6.2 Unit Testy

Tyto testy testují konkrétní části programu bez použití zbývajících částí.

6.2.1 Testovaní Rest

Protože naše aplikace je malá a vykonává jednu konkrétní funkci, můžeme testovat pouze REST řadič. Pomocí testů zkontrolujeme, zda odstraní objekt, když je požádán o jeho odstranění, zda při mazání neexistujícího objektu způsobí chybu.

Závěr

V této práci jsem se zabýval analýzou, návrhem, implementací a testováním aplikace, abych našel autora textu na základě již napsaných textů. S výslednou aplikací jsem spokojeny, protože splňuje všechny cíle této práce a může někomu opravdu pomoci.

Výsledná aplikace je z hlediska kódu velmi malá, ale je to pouze kvůli tomu, že v současné době existuje mnoho knihoven, které dělají velmi složité věci. Přestože se ukázalo, že tato aplikace je malá, pod jedním řádkem kódu se skrývá mnoho různých věcí.

Znalosti získané během školení jsem využil k vytvoření plnohodnotného pracovního programu. Nevěděl jsem, jak textová analýza funguje, studoval jsem problém, analyzoval různá řešení a dospěl k závěru, že jsem vybral nejlepší způsob.

Tato aplikace byla zpočátku velmi pomalá kvůli nesprávnému přístupu k řešení problému. Ale analyzoval jsem problém a našel jeho řešení paralelizací zátěže. Tímto způsobem je dosažena škálovatelnost pro velké objemy dat.

Toto řešení není ideální, protože se mi podařilo dosáhnout pouze 38 procent přesnosti. Bylo by hezké používat hluboké učení, ale pro něj nezbytné miliony dat, které nikdy nebudou.

Bibliografie

1. KAMRAN, Kowsari. *Text Classification Algorithms: A Survey* [online]. 2019. Dostupné také z: https://medium.com/text-classification-algorithms/text-classification-algorithms-a-survey-a215b7ab7e2d [cit. 2020-05-27].
2. HEIDENREICH, Hunter. *Stemming? Lemmatization? What?* [online]. 2018. Dostupné také z: https://towardsdatascience.com/stemming-lemmatization-what-ba782b7c0bd8 [cit. 2020-05-27].
3. ZHANG, Yin; JIN, Rong; ZHOU, Zhi-Hua. *Understanding bag-of-words model: a statistical framework* [online]. Springer, 2010. Č. 1-4. Dostupné také z: https://doi.org/10.1007/s13042-010-0001-0.
4. GOOGLE INC. *Tool for computing continuous distributed representations of words* [online]. 2013. Dostupné také z: https://code.google.com/archive/p/word2vec/ [cit. 2020-05-27].
5. *Text Classification* [online]. Dostupné také z: https://monkeylearn.com/text-classification/ [cit. 2020-05-27].
6. VMWARE INC. *Spring Boot* [software]. Dostupné také z: https://spring.io/projects/spring-boot. [cit. 2020-05-27].
7. BANSAL, Himani. *Best Languages For Machine Learning in 2020!* [online]. 2019. Dostupné také z: https://becominghuman.ai/best-languages-for-machine-learning-in-2020-6034732dd242 [cit. 2020-05-27].
8. *Eclipse Deeplearning4j (version) 1.0.0-beta6* [software]. Dostupné také z: https://deeplearning4j.konduit.ai/. [cit. 2020-05-27].
9. THE UNIVERSITY OF WAIKATO. *WEKA (version) 3.8.4* [software]. Dostupné také z: https://www.cs.waikato.ac.nz/ml/weka/. [cit. 2020-05-27].

10. JETBRAINS S.R.O. *IntelliJ IDEA* [software]. Dostupné také z: `https://www.jetbrains.com/idea/`. [cit. 2020-05-27].

11. VMWARE, INC. *Spring Boot Starter* [software]. Dostupné také z: `https://start.spring.io/`. [cit. 2020-05-27].

12. NETSCAPE COMMUNICATIONS. *JSON* [software]. Dostupné také z: `https://www.json.org/json-en.html`. [cit. 2020-05-27].

13. NATIONAL TAIWAN UNIVERSITY. *LibLINEAR* [software]. Dostupné také z: `https://www.csie.ntu.edu.tw/~cjlin/liblinear/`. [cit. 2020-05-27].

Příloha A

Seznam použitých zkratek

TF-IDF Term Frequency Inverse Document Frequency

SOA Service oriented architecture

WAR Web Application Resource

REST Representational state transfer

API Application programming interface

JPA Java persistence API

LTS Long Time Support

DTO Data transfer object

CRUD Create Read Update Delete

SVM Support Vector Machine

Příloha B

Obsah přiloženého CD

- `readme.txt` ..... stručný popis obsahu CD
- `exe` ..... adresář se spustitelnou formou implementace
- `src`
  - `impl` ..... zdrojové kódy implementace
  - `thesis` ..... zdrojová forma práce ve formátu LaTeX
- `text` ..... text práce
  - `thesis.pdf` ..... text práce ve formátu PDF