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

29. července 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 29. července 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 webovou aplikaci pro identifikaci autora. V této práci projdeme výzkum, analýzu, návrh, implementaci a testování softwaru. Tato aplikace sleduje architekturu mikrservice a lze k ni připojit pomocí REST klienta.

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

Abstract

This thesis is implementing a web application for author identification. In this thesis we will go through research, analysis, design, implementation and testing of a software. This application is follow microservice architecture and can be accessed with REST Client.

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

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.

Tento text je rozdělen do několika kapitol, na začátku budeme analyzovat, jak analyzovat a filtrovat text, pak budeme sbírat a analyzovat aplikační požadavky a podle toho navrhneme architekturu aplikace, pak jde implementace s vybranými knihovnami na základě požadavků a poté otestujeme implementovanou aplikaci.

Zbývá už jen napsat dokumentaci pro vývojáře a uživatele, aby se budoucí rozšiřování a používání bylo co nejjednodušší.

Kapitola 2

Rešerše existujících řešení

Existuje několik aplikací v různých jazycích, které poskytují uživateli různé nástroje strojového učení, které pomáhají uživateli používat strojové učení bez implementace celého algoritmu.

První z nich je Scikit-Learn.[1] Tato aplikace poskytuje snadné a rychlé strojové učení v pythonu. Poskytuje klasifikaci, regresi a předzpracování, vše, co může být užitečné pro vytvoření aplikace pro identifikaci autora.

Druha je WEKA. Weka is tried and tested open source machine learning software that can be accessed through a graphical user interface, standard terminal applications, or a Java API. It is widely used for teaching, research, and industrial applications, contains a plethora of built-in tools for standard machine learning tasks.[2]

Při porovnání těchto dvou aplikací poskytují potřebné nástroje strojového učení pro klasifikaci, ale neposkytují žádnou možnost jejich použití na dálku. Jak vidíme později, WEKA se ukazuje být nejlepší, protože má Java API, které můžeme použít k implementaci naší vlastní aplikace při používání strojového učení WEKA.

Kapitola 3

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.

3.1 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. [3] (přeloženo autorem)

3.1.1 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.

3.1.2 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.
```

3.1.3 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.

3.1.4 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.

3.1.5 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.

3.1.6 Lemmatization

„We've talked about stemming, but what about the other side of things? How is lemmatization different? Well, if we think of stemming as just take a best guess of where to snip a word based on how it looks, lemmatization is a more calculated process. It involves resolving words to their dictionary form. In fact, a lemma of a word is its dictionary or canonical form!"[4]

3.2 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. [5] 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 3.1: 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.

3.3 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.

3.4 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.

3.4.1 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.
$1 \cup 2 = This, car, cat, has, four, leg, wheel$
$1 \cap 2 = This, has, four$
$Jaccard = \frac{|1 \cap 2|}{|1 \cup 2|} = \frac{3}{7}$

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

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.

3.4.2 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. [6]

3.4.3 Naive Bayes Classifier

„Naive Bayes is a family of statistical algorithms we can make use of when doing text classification. One of the members of that family is Multinomial Naive Bayes (MNB). One of its main advantages is that you can get really good results when data available is not much (a couple of thousand tagged samples) and computational resources are scarce.“ [7]

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.

3.4.4 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.

3.4.5 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 4

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.

4.1 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.

4.2 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.

4.3 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 byla aplikace stabilní.

4.4 Funkční požadavky

F1 Vyhledávaní autora textu – Aplikace umožni vyhledávaní autora textu na základe existujících dat.

F2 Vytvářeni autora a projektu – Uživatel bude moci přidat nového autora nebo projekt do aplikace.

F3 Čtení autora a projektu – Uživatel bude moci získat stávající projekty a autory.

F4 Odebraní autora a projektu – Uživatel může odebrat projekt nebo autora z aplikace.

F5 Přidávaní a odebírání autora jako autora projektu – Aplikace umožni přidávaní autora do projektu nebo jeho odebíraní.

F6 Přidávaní a odebírání projektu od autora – Aplikace umožni přidávaní projektu do autora nebo jeho odebíraní.

F7 Aktualizace algoritmu bez restartovaní aplikace – Aplikace umožni uživatelů aktualizovat algoritmus po změně dat v aplikace bez restartovaní cele aplikace.

F8 Aktualizace autora a projektu – Uživatel bude moci poslat novou verze entity do aplikace a aplikace vymění starou verze entity za novou.

F9 Otestovaní přesností použitého algoritmu – Aplikace umožni otestovat přesnost použitého algoritmu na aktuálních datech a vrací přesnost algoritmu.

F10 Komunikace s aplikace přes REST – Aplikace bude přijímat a odesílat veškeré požadavky a odpovědi přes HTTP.

4.5 Nefunkční požadavky

N1 – Aplikace musí být škálovatelná pro vetší rozsah dat

N2 – Multipltaformnost

4.6 Případy použití

Případy použití - definují přesné použití programu, zatímco funkční požadavky definují, co se očekává od funkčnosti programu. Pomáhají jasně vidět program a všechny možné výsledky s ohledem na nefunkční požadavky.

Návratové kódy budou specifikovány v dokumentaci.

UC1 – Vytváření nového projektu/autora

1. Uživatel pošle autora/projekt pomoci POST Request.
2. Nastanou dvě možnosti.
   a) Uživatel obdrží odpověď s ID vytvořené entity.
   b) Uživatel obdrží odpověď s chybou, když není autor nebo projekt validní .

UC2 – Aktualizace projektu/autora

1. Uživatel pošle autora/projekt pomoci PUT Request na adresu která taky určuje ID entity pro aktualizace.
2. Nastanou se dvě možnosti.

a) Uživatel obdrží odpověď s aktualizovanou entitou.

b) Uživatel obdrží odpověď s chybou, když neexistuje takový autor nebo projekt.

UC3 – Vymazání projektu/autora

1. Uživatel pošle DELETE Request na adresu která taky určuje ID entity.
2. Nastanou dvě možnosti.

   a) Uživatel obdrží odpověď že entita byla odstraněna.

   b) Uživatel obdrží odpověď s chybou, když autor nebo projekt neexistuje .

UC4 – Přidání autora do projektu

1. Uživatel pošle PUT Request na adresu která určuje ID autora a projektu.
2. Nastanou dvě možnosti.

   a) Autor bude přidán jako autor vybraného projektu. Uživatel obdrží odpověď se seznamem autoru daného projektu.

   b) Uživatel obdrží odpověď s chybou, když neexistuje autor nebo projekt.

UC5 – Přidání projektu do autora

1. Uživatel pošle PUT Request na adresu která určuje ID autora a projektu.
2. Nastanou dvě možnosti.

   a) Projekt bude přidán jako projekt napsaný vybraným autorem. Uživatel obdrží odpověď se seznamem projektu daného autora.

   b) Uživatel obdrží odpověď s chybou, když neexistuje autor nebo projekt.

UC6 – Nalezeni autora daného textu

1. Uživatel pošle projekt, pro který chce najít autora pomoci POST Request.
2. Nastanou dvě možnosti.

   a) Uživatel obdrží odpověď s seznamem výsledku, kde každá položka seznamu má dvojici autor a pravděpodobnost s kterou je dány autor je autorem projektu.

   b) Uživatel obdrží odpověď s jiným kódem, co znamená že algoritmus ještě není připraven na vyhledávaní autoru.

UC7 – Testovaní použitého algoritmu

1. Uživatel pošle GET Requst na určitou adresu.
2. Nastanou dvě možnosti.
    a) Uživatel obdrží odpověď s přesnosti použitého algoritmu.
    b) Uživatel obdrží odpověď s jiným kódem, co znamená že algoritmus ještě není připraven na použiti.

UC8 – Aktualizace algoritmu po přidaní nových dat do databáze

1. Uživatel pošle GET Requst na určitou adresu.
2. Uživatel obdrží odpověď, která říká, že aktualizace je spuštěna.

UC9 – Odebraní projektu od autora

1. Uživatel pošle DELETE Requst na určitou adresu která určuje autora a projekt.
2. Nastanou dvě možnosti.
    a) Projekt bude odebrán od autora. Uživatel obdrží odpověď se seznamem projektu daného autora.
    b) Uživatel obdrží odpověď s chybou, když neexistuje autor nebo projekt.

UC10 – Odebraní autora od projektu

1. Uživatel pošle DELETE Requst na určitou adresu která určuje autora a projekt.
2. Nastanou dvě možnosti.
    a) Autor bude odebrán od projektu. Uživatel obdrží odpověď se seznamem autoru daného projektu.
    b) Uživatel obdrží odpověď s chybou, když neexistuje autor nebo projekt.

	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10
UC1		X								X
UC2									X	X
UC3				X						X
UC4					X					X
UC5						X				X
UC6	X									X
UC7									X	X
UC8							X			X
UC9						X				X
UC10					X					X

Tabulka 4.1: Splnění požadavků

Kapitola 5

Návrh aplikace

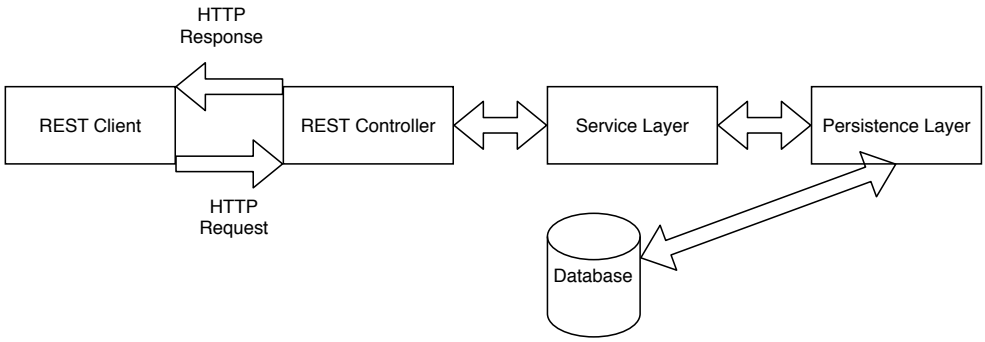
Při psaní aplikace je velmi důležité porozumět tomu, jak bude použita, a pomocí této volby zvolit správnou architekturu pro danou úlohu. Správně zvolena architektura a použiti návrhových patternu příště umožni snadné pochopeni kódu a rozšířeni aplikace. většina softwaru selhala hlavně kvůli špatnému designu, architektuře a kódu.

5.1 Architektura

Architektura je základní část návrhu softwaru. Správný vyber architektury je moc důležitý na příští údržbu projektu a jeho rozšířeni. Nesprávný vyber architektury vede za sebou hodné nekvalitních praktik. Pro změnu architektury v době implementace aplikaci musí byt přepsaná její vetší část nebo cela aplikace.

Podle případu užiti a funkčních požadavku, nejlepší architektura pro danou aplikace zni vícevrstvová architektura.

Vícevrstvová architektura rozděluje aplikace na 3 vrstvy.



Obrázek 5.1: Multilayer Architecture Diagram

5.1.1 Persistence Layer

Persistence Layer - je vrstva, která se zabývá daty v databázi. Poskytuje funkce pro ukládání, aktualizaci, mazání, vytváření, vyhledávání entit v databázi. Každá softwarová vrstva, která usnadňuje programu persistence jeho stavu, se obecně nazývá perzistenční vrstva. Většina vrstev persistence nedosáhne persistence přímo, ale použije systém správy databáze.

5.1.2 Service Layer

Service Layer - je střední vrstva mezi prezentací a ukládáním dat. Abstrakt obchodní logiky a přístupu k datům. Myšlenkou takové vrstvy je mít architekturu, která může podporovat více prezentačních vrstev, jako je web, mobilní atd. Servisní vrstva pomoha skryt složitou logiku aplikace za jednoduchým rozhraním. Tato abstrakce dává možnost změny logiky aplikace, beze změny vyšší vrstvy, protože vyšší vrstva stále vola stejné metody.

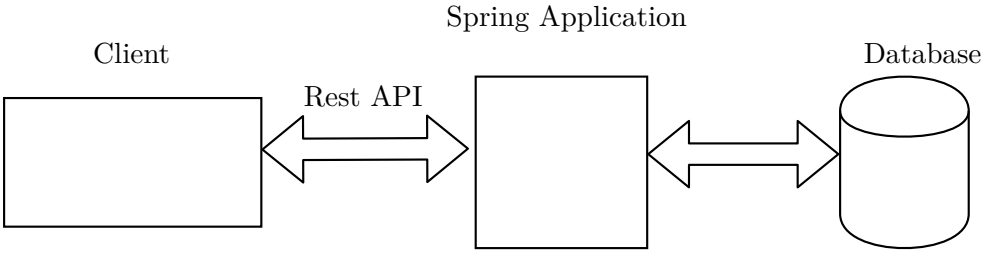
5.1.3 Presentation Layer

Presentation Layer - tato vrstva je umístěna na uživatelském zařízení, ať už je to mobilní telefon, počítač nebo webový prohlížeč. Tato vrstva se používá k zobrazovaní dat uživateli, interakci s nimi, jejich změně a provádění dalších činností poskytovaných servisní vrstvou.

Tato aplikace je webovou službou, takže by neměla prezentační vrstvu. Bylo by však užitečné implementovat jednoduchou aplikaci, která poskytuje reprezentaci dat pro interakci s daty během vývoje.

5.2 Komunikace

Komunikace mezi uživatelem a aplikací by probíhala s požadavkem HTTP v souladu s architektonickým stylem REST. Popisuje standard pro komunikaci mezi aplikacemi na webu. Systémy kompatibilní s REST, často nazývané RESTful systémy, se vyznačují tím, jak jsou bez státní příslušnosti, a oddělují zájmy klienta a serveru. REST Api bude popsán v dokumentaci.

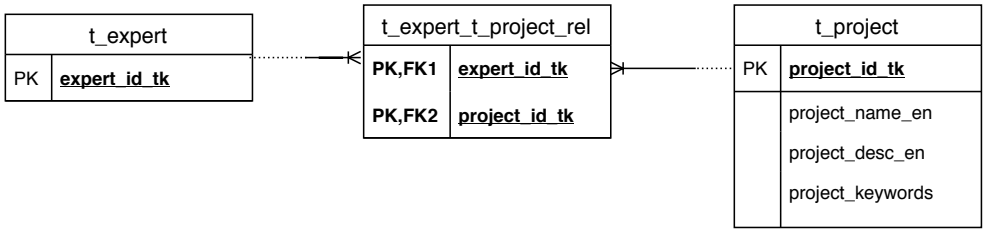


Obrázek 5.2: Diagram komunikace

5.3 Datový model

Datový model popisuje data s nimiž pracuje aplikace. Pokud uživatel potřebuje více informací v Autor entitě nebo v Projekt entitě, lze ji snadno přidat do aplikace.

- **Author** - reprezentuje autora. S dodanou databází nebude nic jiného než ID kvůli ochraně identity autorů.
- **Project** - představuje projekt. Má všechny potřebné části projektu rozděleny do různých částí.
- **AuthorProject** - představuje vztah mezi autorem a projektem, což znamená, že Autor je autorem Projektu.



Obrázek 5.3: Databázový diagram

Kapitola 6

Implementace

V této kapitole implementujeme aplikaci, popíšeme použité knihovny. Aplikace napíšeme podle návrhu z předchozí kapitoly. Pro psaní tohoto projektu použijeme IntelliJ IDEA. [8]

6.1 Použité nastroje

6.1.1 IDE

Volba editoru kódu je většinou volbou vývojářů, protože pro každý populární jazyk existuje alespoň několik editorů. V této aplikaci bychom použili IntelliJ IDE. IDE pomáhá vývojářům s automatickým dokončováním kódu, možnou kontrolou chyb, pomáhá jim přizpůsobit se vzorovým kódům, například Null Object Pattern. Bez IDE by se vývoj jakéhokoli softwaru nesmírně zpomalil.

6.1.2 Verzování

Pro řízení verzí byl použit Github. Je to důležitá součást každého softwaru. Pomáhá sledovat vývoj softwaru. Každá položka v tomto systému je oddělena funkcí, na které pracuje.

K uložení projektu byl použit web Github. Hostování repozitářů Git je bezplatná služba.

6.2 Základ aplikace

Tato aplikace je webová služba a poskytuje uživateli určité služby. Většina webových služeb je napsána v jazyce Java. Java EE je sada API založená na programovacím jazyce Java. Spring Boot je open source framework založený na Java, který se používá k vytvoření micro Service. Micro Service je architektura, která umožňuje vývojářům vyvíjet a zavádět služby nezávisle. Každá spuštěná služba má svůj vlastní proces a tím se získá lehký model pro

podporu obchodních aplikací. Tímto způsobem lze tuto aplikaci začlenit do většího systému. Spring Boot poskytuje vše potřebné k rychlé implementaci a spuštění webové služby. Poskytuje integrovaný aplikační server, závislost injekci, automatickou konfiguraci, není potřeba xml, externí konfiguraci.

6.3 Použité knihovny

Dále sleduje seznam knihoven použitých v této aplikaci. K importu těchto knihoven uvnitř aplikace byl použit Maven. Maven je nástroj pro automatizaci sestavení, který se používá především pro projekty Java. Navíc to stahuje knihovny do projektu automaticky, odstraní to nutnost ručního stahování knihoven.

6.3.1 WEKA

Na PC existuje aplikace WEKA, která poskytuje snadný přístup ke strojovému učení. Knihovna WEKA poskytuje API v Javě pro použití algoritmů strojového učení a filtrů v aplikaci.

6.3.2 LibLINEAR

LibLINEAR je doplněk pro knihovnu WEKA. Implementuje SVM klasifikátor z WEKA. Většinou se používá pro velké sady dat s mnoha třídami a atributy. Poskytuje rychlou klasifikaci a několik různých metrik pro vyhodnocení. K dispozici je taky knihovna LibSVM pro menší sady dat.

6.3.3 JSON

JSON je nejdůležitější knihovna v aplikaci REST. JSON je knihovna, která transformuje jakýkoli objekt, který implementuje určité funkce do řetězce, který může být přenášen přes internet do jiného počítače, a tam bude obrácen zpět na objekt Java z řetězce json. [9]

6.3.4 JUNIT

JUNIT je jednou z nejdůležitějších knihoven každé Java aplikace. Poskytuje API pro psaní a spouštění testů v projektu. Je to důležitá součást vývoje řízeného testem. Bez testů by provádění změn v aplikaci bylo velmi obtížné, i když jsou objekty velmi volně spojené. Protože by mohly existovat nějaké změny, které by bylo obtížné najít bez dobře napsaných testů.

6.3.5 H2

H2 je databáze v paměti, která je ideální pro vývoj jakékoli aplikace. Databáze se spouští se spuštěním aplikace, lze ji nakonfigurovat tak, aby načetla data

ze souboru s již zapsanými daty, takže každé spuštění aplikace by bylo stejné. To nám dává soudržnost, abychom mohli něco změnit v naší implementaci a sledovat výsledek.

6.4 Vytvářeni projektu

S využitím Spring Boot Starter[10] můžeme vytvořit projekt se všemi nezbytnými závislostmi, které jsou již v projektu zahrnuty. Navíc si můžeme vybrat build framework. Projekt je již vytvořen s výchozí adresou **localhost:8080** a webapp serverem Tomcat. Po spuštění projektu můžeme zkontrolovat, zda vše funguje tak, jak bylo zamýšleno odesláním GET.

6.5 Controller

Controller REST je nejdůležitějším bodem v aplikaci webových služeb. Zpracovává příchozí požadavky HTTP a odesílá odpověď volajícímu. Také v Javě můžeme ověřit objekty, které jsou uvnitř těla požadavku. Takto jsou data, která jdou do služby, již konzistentní a splňují požadavky pouhým přidáním jedné anotace. Jen s jednou anotací vytvoříme jakýkoli objekt v ovladači. Aplikace může mít mnoho ovladačů pro samostatné věci. Můžeme to udělat s adresou v anotaci **@RequestMapping**. S tím by každá metoda, která přijímá HTTP požadavek, měla adresu vytvořenou z adresy aplikace a adresy Rest Controller.

```
@RestController
@RequestMapping("/author-identification")
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);
    }
```

6.6 Entity

Entita je hlavní objekt, který uchovává data získaná z databáze, a objekt, který uchovává data, která by byla uložena do databáze. V tomto projektu máme pouze 2 entity, které uchovávají data, a jedna entita, což je vztah mezi nimi. Opět vytvoříme entitu přidáním anotace k jednoduchému Java objektu. S pomocí různých anotací můžeme určit požadavky na proměnné v objektu, jako **@NotEmpty**, **NotNull**.

Pomocí anotace vztahů můžeme určit vztahy mezi entitami. Kaskádování nám ušetří spoustu času, když načítáme entity z databáze. Kaskádování je takové, že když se s tímto objektem něco děje, pak se provádí s objektem, ke kterému je připojen. Můžeme definovat přesné metody, které by byly vyvolány na související objekt. Díky eager kaskádování, když čteme z databáze, Hibernace načte související objekt současně.

Pomocí anotace ID anotujeme proměnnou, která bude použita jako ID entity. Hibernate nabízí automatizované generování ID, takže uživatel nemusí implementovat žádné generování ID. Vytváří však problém, když uživatel nedostane zprávu s ID vytvořené entity.

Protože máme entitu, která definuje vztah mezi dvěma entitami, musíme tam mít 2 ID. Protože nemůžeme použít dvě proměnné ID, můžeme vytvořit složené ID, které bude obsahovat ID i sloužit jako ID entity.

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

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

```
@Entity
@Table(name = "t_expert_t_project_rel")
@IdClass(AuthorProjectCompositeId.class)
public class AuthorProject {
    @Id
    @ManyToOne(fetch = FetchType.EAGER)
    @JoinColumn(name = "expert_id_tk_t_expert", referencedColumnName
        = "expert_id_tk")
    private Author author;

    @Id
    @ManyToOne(fetch = FetchType.EAGER)
    @JoinColumn(name = "project_id_tk_t_project",
        referencedColumnName = "project_id_tk")
    private Project project;
    /*Setters and Getters*/
```

6.7 Repository

Teď, když máme Controller a entity, můžeme vytvořit úložiště, které poskytuje funkce pro provádění operací CRUD s entitami. V Spring Boot můžeme použít JpaRepository. Poskytuje všechny základní funkce, které požadujeme. Navíc je snadné ji rozšířit, pokud to budeme v budoucnu potřebovat. S Spring Boot můžeme pouze přidat anotaci a rozšířit třídu JpaRepository a vytvořit úložiště s našimi vlastními třídami.

```
@Repository
public interface AuthorRepository extends JpaRepository<Author, Long>
    {
}
```

6.8 Service

Service drží hlavní logiku v Web Service aplikaci. V naší aplikaci máme dvě služby, jedna řeší operace na projektech a druhá zpracovává autory. Službou může být jakýkoli obyčejný objekt java anotovaný anotací @Service. Pomocí anotace @Autowired můžeme vložit Repositories do služby a samotnou službu vložit do Controller. Tato anotace nám umožňuje vložit objekt injekcí do konstruktoru nebo pomocí setteru.

6.9 Author Classifier

Pro snadné škálování aplikací a zaměnitelné algoritmy pro klasifikaci textu a abstrakci z implementace algoritmu by mělo existovat rozhraní, které musí každý klasifikátor implementovat. Vzhledem k tomu, že se algoritmy od sebe navzájem velmi liší a používáme správné postupy kódování, nemůžeme si dovolit tvrdé kódování jedné implementace do služby. Kromě toho bychom měli poskytnout obaleni pro klasifikátor, toto obaleni by melo zprovoznit ověřeni stavu klasifikátoru. S pomocí rozhraní a wrapperu, pokud chceme změnit algoritmus na jiný, vše, co musíme udělat, je jen změnit jeden řádek kódu ve srovnání s odstraněním mnoha funkcí a psaní nových, pokud bychom nepoužili správné postupy kódování a abstrakce.

```
public interface AuthorClassifier {
    double testClassifier();
    void initClassifier(List<Author> authors);
    List<ImmutablePair<Double,String>> classifyText(String text);
    void resetClassifier();
}

public interface AuthorClassifierWrapper {
    Boolean isInitialized();

    Boolean isInitializing();

    Boolean isRefreshRequested();

    double testAlgorithm();

    void initClassifier(List<Author> allAuthors, AuthorClassifier
        authorClassifier);

    List<SearchResultDto> findPossibleAuthor(ProjectDto project);

    void refreshClassifier(List<Author> allAuthors);
}
```

6.10 Analýza textu

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

6.10.1 Zpracování

Nejjednodušší součástí je zpracování textu. Protože v tomto textu používáme anglický text, můžeme text rozdělit na slova pouhým oddělením mezer a odstraněním koncových mezer. U klasifikátorů v WEKA se to provádí pomocí

filtru StringToVector. V klasifikaci ne strojového učení používáme 1 slovní sáček slov. V WEKA klasifikaci můžeme použít víceslovný pytel slov, navíc poskytuje prořezávání slov na základě frekvence.

6.10.2 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. Jak uvidíme později, lemmatizace snižuje přesnost algoritmu pro náš účel.

6.11 Testovaní naivních algoritmu

Nemůžeme použít mnoho různých algoritmu, pokud je většina z nich nepřesná. V této části testujeme algoritmy popsané na začátku této práce. U algoritmů strojového učení nebyl jeden autor z projektu odstraněn, a poté byl vyvolán algoritmus pro získání možného autora pro odstraněný projekt, byl opakován pro každého autora

6.11.1 Jaccard

Toto je nejhorší algoritmus, protože používá počet odpovídajících slov ve srovnání s celkovým počtem jedinečných slov, pomáhá, když chceme najít dokumenty, které mluví o stejném tématu, ale nepomůže, když chceme najít autora daného textu.

Lemmatizace zde neudělala nic, protože pomohla pouze identifikovat nesprávné autory jako autory daného textu. Počet shod na každé velikosti datové sady byl 0.

6.11.2 Kosinová podobnost

Kosinová podobnost je mnohem užitečnějším algoritmem, protože nám dává míru podobnosti vektorů. Navíc k tomu používáme TF-IDF, což je normalizace slov, která jsou používána, která jim dává nižší prioritu, a dává vyšší prioritu slovům, která se používají zřídka. Protože porovnává směry vektorů, většinou se používá k určení, zda mají texty stejná témata. Z 500 nalezl správně pouze 1 autora. Lemmatizace snižuje přesnost na 0. Protože od většího počtu autorů nedostává žádné informace ostatním autorům, je testování na větší počet autorů zbytečné.

I když nám TF-IDF v tomto algoritmu dalo 1 nalezeni, je to opravdu užitečné pro normalizaci dat v algoritmech strojového učení.

6.11.3 Word2Vec Google Model

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. Zde nemůžeme udělat žádný TF-IDF, protože by posunul vektor do slova, což by změnilo jeho význam. Bylo nalezeno pouze 4 z 500 autorů. Je to opět pouze proto, že používá sémantiku slova a je nejvhodnější pro určení, o čem tento text mluví. Protože je model před-připraveny, není žádný rozdíl v množství autorů, na kterých by byl testován, protože to nepřináší novou informaci do modelu.

6.12 Testování algoritmu strojového učení

Testování bylo provedeno metodou skládání. Když je celý datový soubor rozdělen do n bloku a pak jsou k tréninku použity n-1 bloku a poslední je používán k testování, je spuštěn n krát, a v každém spuštěni testovací blok je jiný. Tímto způsobem můžeme snížit zkreslení výsledků testů, když byl blok na testovaní příliš dobrý nebo špatný. Zde bylo použito 4 skládání, takže 3 bloky se používají pro trénovaní a 1 pro testování.

Při strojovém učení, čím více dat dáte modelu, tím přesnější bude model, ale nezvyšuje se nekonečně nahoru, po určitém množství dat se přesnost přestane růst. Hluboké učení je lepší, ale nemáme k dispozici potřebné množství dat, abychom je mohli používat, protože ne každý autor má tisíce projektů, na kterých lze model trénovat.

Bylo by dobré přidat text rozdělený do vět. Protože by to udělalo více záznamů na autora.

6.12.1 Naive Bayes Classifier

Používáme textový klasifikátor, který je již implementován v WEKA. Pro tento klasifikátor bylo provedeno testování se zapnutou a vypnutou lemmatizací, byly použity různé n-gramy, různé minimální frekvence.

Accuracy	Data Size	Min Term Freq	Min N-Gram	Max N-Gram	Lemmatization
3.57%	500	2	2	3	On
4.57%	500	2	2	3	Off
4.9%	500	1	2	3	Off
0.90%	2500	2	2	3	Off

Tabulka 6.1: Naive Bayes Multinomial Text Results

Jak je vidět z tabulky, přidání dalších autorů nepomohlo s přesností kvůli přidání autorů, kteří nejsou stejní jako přidávání dalších projektů autorovi.

Lemmatizace pouze snižuje přesnost algoritmu, jak zde vidíme, kvůli problému, který se snažíme vyřešit. Je to užitečné při určování textového tématu, ale není to užitečné při určování autora textu.

S minimální periodickou frekvencí 1 vidíme, že jsme zvýšili přesnost, ale není užitečné mít v modelu slova, která popisují pouze jednoho autora, protože zabírá prostor a zpomaluje aplikaci, když můžeme používat slova, která se používají v jiných texty a stále s nimi určují autora.

6.12.2 KNN

Tento algoritmus je implementován ve WEKA a zde bychom testovali stejné parametry jako v předchozím. Tento algoritmus je nejvíc citlivý na malý dataset.

Accuracy	Data Size	Min Term Freq	Min N-Gram	Max N-Gram	Lemmatization
47.11%	500	2	2	3	On
47.17%	500	2	2	3	Off
43.22%	500	1	2	3	Off
42.45%	2500	2	2	3	Off

Tabulka 6.2: KNN Results

6.12.3 LibLINEAR

Tento algoritmus [11] navíc používá regresi. Tento algoritmus má různé vzdálenosti, ale byl použit s výchozí vzdáleností kvůli tomu, že byla relativně stejná ve srovnání s ostatními z hlediska přesnosti a byla nejrychlejší.

U tohoto algoritmu je použití pouze slov, která se vyskytují 2 nebo vícekrát, velmi užitečné, protože vyžaduje mnoho času na sestavení a u velkých slovníků to může být velmi dlouhá doba. Použití slov s frekvencí alespoň 2 pomáhá zkrátit čas bez omezení přesnost, zůstává relativně stejná.

S tímto algoritmem měla větší sada dat nižší přesnost, ale je to pravděpodobně způsobeno tím, že někteří autoři mají nižší počet textů, a proto je průměrný počet textů na autora nižší

Accuracy	Data Size	Min Term Freq	Min N-Gram	Max N-Gram	Lemmatization
52%	500	2	2	3	On
52.12%	500	2	2	3	Off
53.17%	500	1	2	3	Off
51.90 %	2500	2	2	3	Off

Tabulka 6.3: LibLINEAR SVM Results

6.13 Porovnání algoritmů

Můžeme vidět, že nejpřesnějším algoritmem je LibLINEAR. Tento algoritmus je velmi rychlý pro velké soubory dat. Použijeme-li minimální termínovou frekvenci rovnou 2, stane se velmi rychlou a pro dodávaný datový soubor se staví asi 5 minut.

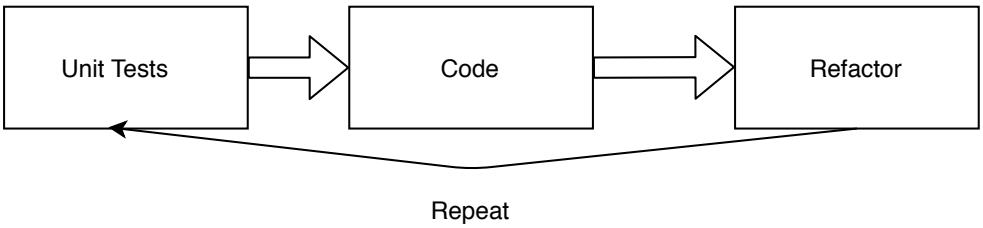
Algorithm	Data Size	Accuracy
Jaccard	Irrelevant	0%
Cosine Similarity with TF-IDF	Irrelevant	0.002%
Word2Vec Average Vector	Irrelevant	0.008%
Naive Bayes	500	4.9%
Naive Bayes	2500	0.9%
KNN	500	47.17%
KNN	2500	42.45%
LibLINEAR SVM	500	53.17%
LibLINEAR SVM	2500	51.90%

Tabulka 6.4: Algorithm Results

Kapitola 7

Testovaní

Testy jsou velmi důležité při vývoji softwaru. Bez nich by bylo velmi obtížné opravit kód, přidat nové funkce, upravit jej a zjistit, že to, co dříve fungovalo, stále funguje, tak jak to bylo myšleno. Vždy předtím, než vývojář napíše kód, píše Unit test, který testuje, zda funkce dělá to, co má dělat, a neudělá to, co nemá dělat. Testy liší se nejen tím, co testují, ale také tím, jak to testují. Zatímco jeden test používá Mock k napodobení objektu k oddělení testování jedné části od druhé, jiné testují, jak objekty vzájemně interagují.

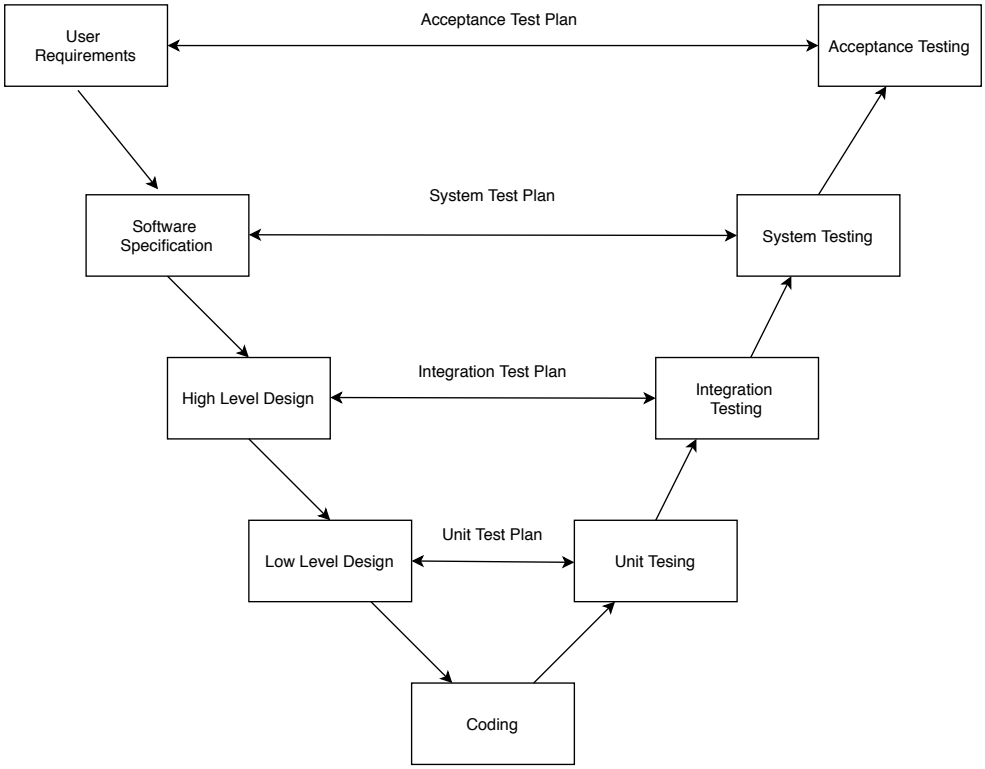


Obrázek 7.1: Test Driven Development Cycle

Existují různé testy, které testují aplikaci odlišně. Unit testy jsou velmi rychlé a testují nejmenší části aplikace. Integration testy je úroveň testování softwaru, kde jsou jednotlivé jednotky kombinovány a testovány jako skupina. Akceptační testování je typ testů, které testují, že software dělá to, co bylo popsáno v diagramech použití. Jsou prováděny, aby určily, zda byly splněny všechny požadavky.

Model V je nyní jedním z nejpoužívanějších procesů vývoje softwaru. Zavedení modelu V skutečně prokázalo implementaci testování již od fáze požadavku. Model V se také nazývá ověřovací a ověřovací model. Verification je technika statické analýzy. V této technice se testování provádí bez provedení kódu. Příklady zahrnují: recenze, inspekce a návody. Validation je technika dynamické analýzy, kde se testování provádí spuštěním kódu. Pro vývojáře

je to snadný způsob, jak určit, do které fáze Softwarového designu by se měl vrátit, opravit vzniklý problém.



Obrázek 7.2: V Model

7.1 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.

```
@SpringBootTest
public class SmokeTests {

    @Autowired
    Controller controller;

    @Autowired
    AuthorRepository authorRepository;

    @Autowired
    AuthorProjectRepository authorProjectRepository;
```

```
    @Autowired
    ProjectRepository projectRepository;

    @Test
    public void contextLoads() throws Exception {
        assertThat(controller).isNotNull();
        assertThat(authorRepository).isNotNull();
        assertThat(authorProjectRepository).isNotNull();
        assertThat(projectRepository).isNotNull();
    }

}
```

7.2 Unit Testy

Tyto testy testují konkrétní části programu bez použití zbývajících částí. Zde ukážeme příklad testování funkcionality create v servisu.

```
    @Test
    public void createProjectTest(){
        ProjectRepository projectRepository =
            mock(ProjectRepository.class);
        AuthorProjectRepository authorProjectRepository =
            mock(AuthorProjectRepository.class);
        ProjectService projectService = new
            ProjectService(projectRepository, authorProjectRepository);
        Project project = new Project();
        project.setDescEn("a");
        project.setNameEn("b");
        project.setKeywords("c");
        project.setProjectIdTk(1L);
        ProjectDto projectDto = new ProjectDto();
        projectDto.setDescEn("a");
        projectDto.setNameEn("b");
        projectDto.setKeywords("c");
        when(projectRepository.save(any())).thenReturn(project);
        assertEquals(projectService.createProject(projectDto),project.getProjectIdTk());
    }
```

Zde jsme nahradili repozitář třídou Mock, protože chceme otestovat naši službu a nikoliv cizí knihovnu.

7.3 Integration Testy

Protože naše aplikace nemá prezentační vrstvu, naše integrační testy jsou ve skutečnosti také přejímací testy. Zde testujeme naši aplikaci tak, že posíláme žádosti obvyklým uživatelem a kontrolujeme, zda aplikace nastavila správné proměnné, zda měla inicializovaný klasifikátor, nebo zda odstranila správnou entitu atd.

```
    @Test
public void removeProjectFromAuthor() throws Exception {
    setUp();
    String uri = API_PATH+"/author/8/project/7273";
    MvcResult mvcResult =
        mvc.perform(MockMvcRequestBuilders.delete(uri)
            .accept(MediaType.APPLICATION_JSON_VALUE)).andReturn();

    int status = mvcResult.getResponse().getStatus();
    assertEquals(200, status);
    String content = mvcResult.getResponse().getContentAsString();
    ObjectMapper objectMapper = new ObjectMapper();
    List<ProjectDto> projects = objectMapper.readValue(content,
        new TypeReference<List<ProjectDto>>(){});
    assertEquals(1, projects.size());
}
```

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.

Na začátku jsme provedli průzkum existujících řešení tohoto problému. Se ukázalo, že jich je několik, ale nemohou být nasazeny jako webové aplikace.

Poté jsme v textové analýze analyzovali různé způsoby, jak ukládat text, filtrovat jej od nežádoucího obsahu a vzájemně porovnávat texty nebo klasifikovat text.

Dále jsme analyzovali požadavky, napsali případy použití pro budoucí aplikaci a analyzovali možnou architekturu aplikace na základě požadavků.

Po návrhu softwaru jsme začali s implementací aplikace. Vybrali jsme určité knihovny pro jejich ohromnou funkčnost a snadné použití. Jak se ukázalo, WEKA má API v Javě, které jsme používali k strojovému učení, aniž bychom sami implementovali celé algoritmy. Otestovali jsme algoritmy a vybrali jsme ten nejpřesnější.

Nakonec jsme testovali aplikaci na základě požadavků na ni. Napsali jsme a provedli Unit testy, Integration testy.

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.

Toto řešení není ideální, protože se mi podařilo dosáhnout pouze 53 procent přesnosti. Bylo by hezké používat hluboké učení, ale pro něj nezbytné miliony dat, které nikdy nebudou.

Požadavky byly plně splněny. Aplikaci lze nasadit na server vybraný uživatelem nebo na výchozí server, který se již nachází v aplikaci. Uživatel se může připojit ke své vlastní databázi. Byl vytvořen databázový diagram. API bylo popsáno v uživatelské dokumentaci a jako příklad pro uživatelskou aplikaci byla napsána malá GUI aplikace, která pomáhá při vývoji. Aplikace byla plně testována. Protože používáme Javu a na tuto aplikaci lze připojit přes internet, může tuto aplikaci používat každá platforma, která se může připojit k internetu.

Bibliografie

1. . *Scikit-learn* [software]. Dostupné také z: https://scikit-learn.org/stable/. [cit. 2020-05-27].
2. 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].
3. 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].
4. HEIDENREICH, Hunter. *Stemming? Lemmatization? What?* [online]. 2018. Dostupné také z: https://towardsdatascience.com/stemming-lemmatization-what-ba782b7c0bd8 [cit. 2020-05-27].
5. 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.
6. 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].
7. *Text Classification* [online]. Dostupné také z: https://monkeylearn.com/text-classification/ [cit. 2020-05-27].
8. JETBRAINS S.R.O. *IntelliJ IDEA* [software]. Dostupné také z: https://www.jetbrains.com/idea/. [cit. 2020-05-27].
9. NETSCAPE COMMUNICATIONS. *JSON* [software]. Dostupné také z: https://www.json.org/json-en.html. [cit. 2020-05-27].
10. VMWARE, INC. *Spring Boot Starter* [software]. Dostupné také z: https://start.spring.io/. [cit. 2020-05-27].

11. 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

IDE Integrated development environment

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
- `doc` ..... uživatelská dokumentace
- `text` ..... text práce
  - `thesis.pdf` ..... text práce ve formátu PDF