Západočeská univerzita v Plzni
Fakulta aplikovaných věd
Katedra informatiky a výpočetní techniky

Diplomová práce

Inteligentní vyhledávání dokumentů

Plzeň 2017 Jiří Martínek

Místo této strany bude zadání práce.

Prohlášení

Prohlašuji, že jsem diplomovou práci vypracoval samostatně a výhradně s použitím citovaných pramenů.

V Plzni dne 16. května 2017

Jiří Martínek

Poděkování

Na tomto místě bych chtěl poděkovat svému vedoucímu diplomové práce doc. Ing. Pavlu Královi Ph.D. za odborné vedení, za pomoc a rady při zpracování této práce.

Jiří Martínek

Abstract

This diploma thesis deals with information retrieval in a set of scanned documents in form of raster images. First, the images are converted into the text form using optical character recognition (OCR) methods. Unfortunately, there are errors in conversion, therefore another part of the work deals with error correction. This thesis propose several error correction methods that are combined to achieve the best possible results. Then, the corrected documents are indexed into the full-text Apache Solr database. The resulting application allows to efficiently find the requested document according to a full-text query. Error correction of the OCR output helps to increase the accuracy of full-text search. The accuracy of the system was experimentally verified on the real data.

Abstrakt

Tato diplomová práce se zabývá problematikou vyhledávání informací v množině naskenovaných dokumentů v podobě rastrových obrázků. Nejdříve je proto proveden převod rastrového obrázku do textové podoby pomocí metod optického rozpoznávání znaků (OCR). V rámci převodu bohužel dochází k chybám, proto se další část práce zabývá samotnou opravou chyb. V práci je navrženo několik metod oprav chyb, které jsou zkombinovány pro dosažení co nejlepšího výsledku. Dále jsou opravené dokumenty zaindexovány do fulltextové databáze Apache Solr. Výsledná aplikace umožňuje efektivně najít požadovaný dokument dle fulltextového dotazu. Oprava chyb OCR převodu přispívá ke zvýšení přesnosti fulltextového vyhledávání. Přesnost systému byla experimentálně ověřena na dodaných datech z reálného prostředí.

Obsah

Seznam obrázků

Seznam tabulek

Seznam ukázek zdrojových kódů

1 Úvod

V současné době je většina dokumentů v nestrukturované podobě. Tato podoba dokumentů je pro počítač nečitelná. Nejčastěji se jedná o naskenované dokumenty, různé ručně psané poznámky či dokumenty staršího data psané na psacím stroji. Tyto dokumenty jsou čitelné pro člověka, ale s jejich zvyšujícím se počtem se zhoršuje schopnost v nich efektivně vyhledávat informace.

Cílem diplomové práce je naprogramovat aplikaci, která umožní efektivně vyhledávat v množině dokumentů v podobě rastrových obrázků. Rastrové dokumenty budou nejdříve převedeny do textové podoby pomocí optického rozpoznávání znaků (OCR). Výsledek OCR je ovlivněn kvalitou naskenovaného dokumentu, a tak mohou být vytvářeny chyby, které posléze můžou negativním způsobem ovlivnit výsledek vyhledávání. Podstatnou částí této diplomová práce je implementace systému na opravu takto vygenerovaných chyb. Technika opravování chyb je založena na kombinaci pravidel a metod strojového učení.

V práci budou nejdříve vysvětleny techniky vyhledávání a rozpoznávání znaků včetně jejich vylepšení o detekci a opravu chyb. Teoretická část rovněž popíše existující nástroje k řešení této problematiky a jejich analýzu. Součástí teoretické části bude také analýza nástrojů umožňující fulltextové vyhledávání. Implementační část práce popisuje vytvoření vyhledávacího systému naskenovaných dokumentů a jeho integraci s rozpoznáváním znaků včetně korekce chyb, která umožní efektivní vyhledávání.

Pro opravování chyb se používá jazykový model, který na základě OCR výstupu navrhuje nejpravděpodobnější korekce. Součástí programu je také klasifikace dokumentů, která určí nejpravděpodobnější typ dokumentu (např. smlouvy, zákony a zprávy) a díky tomu lze vyhledávat dokumenty podle třídy.

2 Indexace a vyhledávání

Vzhledem k tomu, že cílem diplomové práce je implementace vyhledávacího systému, bude v této kapitole vysvětlena problematika indexace dokumentů a jejich vyhledávání.

Vyhledávání dokumentů je proces, kdy se na základě dotazu prohledá množina dokumentů a vrátí se výsledek. Výsledek obsahuje dokumenty, které odpovídají dotazu. Velmi používaný typ vyhledávání je tzv. *fulltextové* vyhledávání.

Fulltextové vyhledávání je takové vyhledávání, při kterém se porovnává dotaz od uživatele s veškerým *naindexovaným* obsahem dokumentu.

Aby bylo možné efektivně vyhledávat, je nutné v prvé řadě dokumenty *zaindexovat* – vytvořit *index* pomocí vhodné datové struktury. Na rozdíl od pomalého sekvenčního procházení dokumentů, umožňuje *index* zvýšení rychlosti a efektivity vyhledávání díky vhodné reprezentaci dokumentů.

2.1 Index

V této části bude stručně popsán postup vytvoření *indexu*, používané datové struktury a jakým způsobem se textová data *indexují*.

Slova v dokumentu se označují jako *tokeny*. Stejných *tokenů* může dokument obsahovat více. Pro účely vyhledávání není potřeba uchovávat všechny, ale stačí uchovávat pouze jejich množinu (každý *token* je zastoupen pouze jednou). Takovéto jedinečné *tokeny*, které ještě projdou fázemi předzpracování (*normalizace*, *stemming*, *lemmatizace*) se potom označují jako tzv. *termy*. Jednotlivé fáze předzpracování budou v této kapitole ještě vysvětleny.

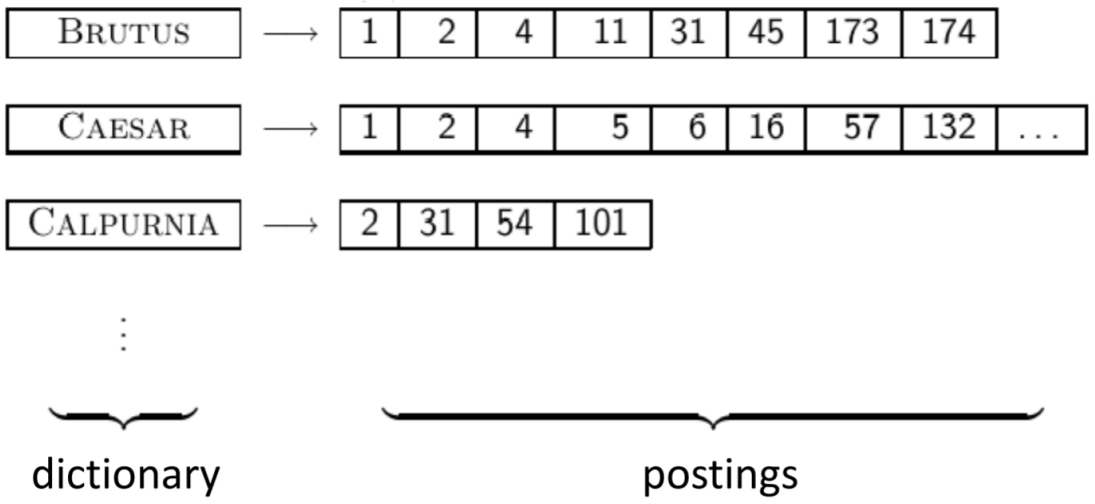
2.1.1 Incidenční matice

První z možných reprezentací *indexu* je incidenční matice. Máme-li k dispozici pro každý dokument jeho *termy*, lze sestavit matici, ve které budou sloupce tvořit dokumenty a řádky slovník *termů*. Hodnota 1 v matici znamená, že se *term* v dokumentu nachází a hodnota 0 nikoliv.

Nevýhodou této reprezentace je její velikost. Při velkém počtu dokumentů (statisíce až miliony) bude také vysoký počet *termů* a matice bude mít obrovské rozměry a navíc bude velmi řídká (budou převažovat nuly). Proto není tato reprezentace *indexu* příliš vhodná.

2.1.2 Invertovaný index

Tato reprezentace je založena na podobném principu jako předchozí incidenční matice s tím rozdílem, že se uchovávají pouze výskyty *termů* v jednotlivých dokumentech. Každý *term* ze slovníku (*dictionary*) obsahuje seznam dokumentů, ve kterých se vyskytuje (tzv. *postings*). Reprezentace pomocí *Invertovaného indexu* je znázorněna na obrázku 2.1.



Obrázek 2.1: Ukázka invertovaného indexu [18]

Z důvodu zvýšení rychlosti vyhledávání je vhodné, aby seznamy obsahující dokumenty byly seřazené.

Volba datové struktury pro reprezentaci slovníku *termů* (*dictionary*) je klíčová, protože se požaduje efektivní vyhledávání i pro obrovské množství dokumentů (stovky tisíc až miliony). Podle publikace [18] jsou vhodné datové struktury **stromy** a **hash tabulky**.

Datová struktura strom

Nejčastější a nejjednodušší příklad stromové datové struktury je *binární strom* (každý uzel stromu má nejvýše dva následníky).

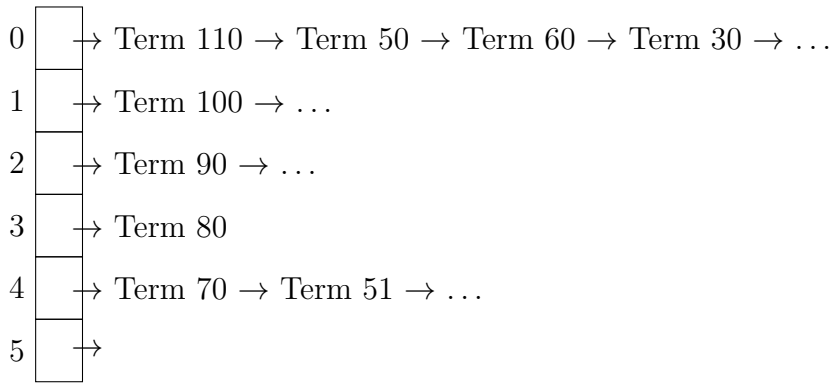
Strom díky své hierarchické struktuře může velice dobře zachycovat vztahy mezi jednotlivými uzly, čehož lze využít například při *prefixovém* hledání (dva *termy* se stejným *prefixem* budou ve stromu blízko u sebe).

Vyhledávání ve stromu má průměrně logaritmickou složitost. Nicméně pokud strom není vyvážený, tak může složitost degradovat na nejhorší možný případ, lineární složitost. Řešením by bylo použití některého z vyvažovacích stromů (*AVL*, *Red-Black*, případně *B-strom*), ale je třeba si uvědomit, že vyvažování stromu je drahá operace.

Datová struktura Hash tabulka

Hash tabulka je datová struktura sloužící k ukládání hodnot podle klíče (*hash* funkce). Oproti stromům má *hash tabulka* výhodu v tom, že rychlost vyhledávání prvků není závislá na momentálním uspořádání datové struktury, protože tabulku tvoří pole s neměnnou velikostí.

Každý *term* ze slovníku je namapován na celé číslo na základě *hash* funkce. Je třeba zvolit takovou funkci, aby nedocházelo často ke kolizím. Nicméně při značné velikosti slovníku se kolizím vyhnout nelze a je třeba je řešit. Jedním z možných řešení je tzv. *zřetězení* prvků. Ukázka je na obrázku 2.2.



Obrázek 2.2: Příklad Hash tabulky se zřetězením prvků

Vyhledávání v *hash tabulce* je rychlejší než ve stromu, ale ztrácí se výhoda hierarchické struktury a tedy i *prefixového* vyhledávání (např. všechny termy začínající na *auto*).

2.1.3 Tvorba slovníku

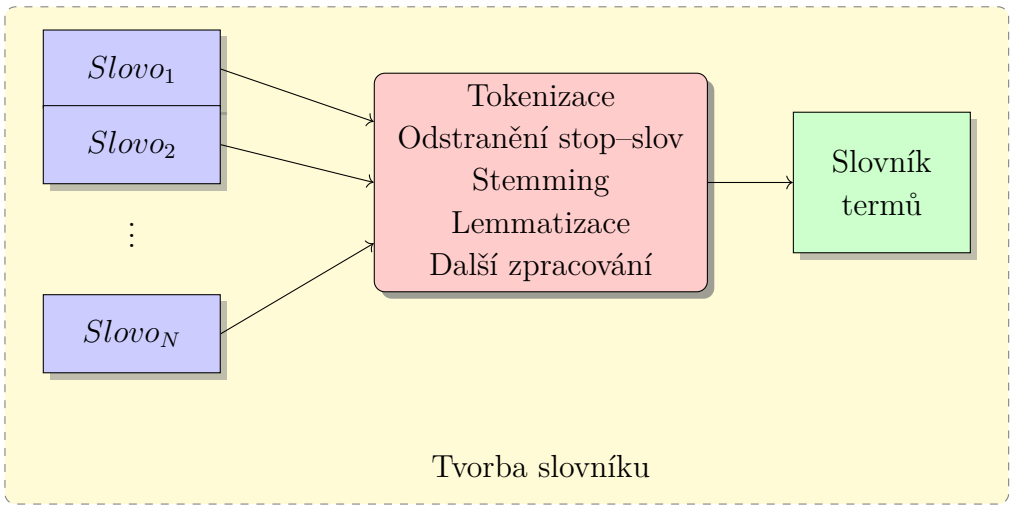
Aby bylo možné vytvořit slovník *termů* (*dictionary*), je třeba z každého dokumentu extrahovat *termy*. Každé slovo by mělo projít fázemi, které jsou uvedené na obrázku 2.3. Nyní budou tyto fáze stručně popsány.

Tokenizace

Proces *tokenizace* je základní extrakce slov z dokumentu. Základní dělení slov je dle mezer v jednotlivých větách. Je možné aplikovat **regulární výraz**, který z jednotlivých vět dokumentu vytvoří seznam slov bez interpunkčních znamének (*tokeny*).

Odstranění stop-slov

Tuto fázi lze chápat jako jakýsi filtr, který zabrání některým vybraným *tokenům* být dále zpracovávané. Jedná se o častá slova, objevující se ve většině dokumentů, která



Obrázek 2.3: Princip vytváření slovníku z dokumentu

nemá smysl zahrnout do slovníku. Takováto slova jsou označována jako *stop–slova* (*stop–words*).

Jedna možná strategie detekce *stop–slov* je frekvenční analýza slov v daném jazyce. Na dostatečně velkém reprezentativním vzorku textu se spočítají četnosti jednotlivých slov. Slova s výrazně vyšší četností, než je průměr, lze považovat za *stop–slova*. Na základě této analýzy se vytvoří seznam *stop–slov* (*stop–list*). Slova v tomto seznamu nebudou zahrnuta do indexace [18]. Obecně lze mezi *stop–slova* zařadit předložky, částice, zájmena a další často se vyskytující zpravidla nevýznamová slova.

Fáze odstranění *stop-slov* výrazně zmenší seznam *tokenů* a v důsledku toho i slovník *termů*.

Stemming a Lemmatizace

V dokumentech se objevují různé tvary téhož slova. V angličtině můžeme například uvést varianty slova *be* (*am*, *are*, *is*). V českém jazyce je situace horší, protože obsahuje pády a velké množství různých tvarů slov způsobené například skloňováním.

Společný cíl *stemmingu* a *lemmatizace* je zredukovat různé tvary slov na (pokud možno) jeden základní tvar. Pro angličtinu uveďme pro názornost příklad aplikace *stemmingu* a *lemmatizace* pro následující větu:
the boy's cars are different colors → the boy car be differ color [18]

Ačkoliv mají *stemming* a *lemmatizace* stejný cíl, liší se řešením.

Lemmatizace obvykle s použitím slovníku a morfologické analýzy slov korektně vrátí základní tvar slova tzv. *lemma* (viz např. změna *are* → *be*).

Stemming obvykle využívá heuristický přístup, který odstraní konec slova (viz např. změna *different* → *differ*). Ve většině případů se korektně dosáhne základního tvaru (kořenu) slova[1].

Další zpracování tokenů:

- odstranění duplicitních tokenů,
- převedení všech znaků na malá písmena (tzv. *lower casing*),
- normalizace (třídy ekvivalence) – např. slova *U.S.A., USA, United States* se zařadí do třídy *USA*.

Cílem všech těchto fází je redukce slov v dokumentu do takové míry, aby zůstala pouze slova, která má smysl zahrnout do *indexace*. Je třeba ještě zdůraznit, že všechny tyto fáze je nutné aplikovat při přidání nového dokumentu a také při zpracování dotazu.

2.2 Vyhledávání

Druhou částí problematiky je již samotné vyhledávání dokumentů a s tím spojená tvorba a realizace dotazů. Cílem vyhledávání je najít dokumenty, které vyhovují dotazu (obsahují některé nebo všechna slova v závislosti na dotazu).

[1]Nejznámější algoritmus pro *stemming* v anglickém jazyce je *Porterův algoritmus* [18].

3 Optické rozpoznávání znaků

V této kapitole bude popsána technika optického rozpoznávání znaků (*OCR – Optical Character Recognition*), rozdělení metod a bude stručně popsán princip rozpoznávání. Cílem *OCR* je rozpoznat znaky uložené v obrazovém formátu (případně v širším smyslu souvislý text) a uložit je v podobě textu. Díky této informaci je umožněno další zpracování [20]. Výkon a kvalita *OCR* závisí na použité technice, přístupu k řešení (algoritmu) a také především kvalitě vstupu.

3.1 Rozdělení metod OCR

Podle [8] je možné rozdělit optické rozpoznávání znaků do několika skupin.

První možné dělení je z hlediska vstupu. Rozpoznávat lze ručně psaný nebo tištěný text. *OCR* lze dále dělit na *off–line* a *on–line* zpracování. Při *on–line* zpracování probíhá analýza v reálném čase při psaní textu. *Off–line* zpracování provádí rozpoznávání z již uložených dokumentů. Digitalizace ručně psaného textu pomocí *OCR* může být použita buď pro rozpoznávání, anebo k verifikaci autora textu.

Především pro rozpoznávání ručně psaného textu platí, že čím omezenější a pravidelnější rukopis, tím lepší kvalita výstupu. Pokud v rukopisu nelze najít vzory a pravidla, je téměř nemožné pomocí metod *OCR* dosáhnout kvalitního zpracování.

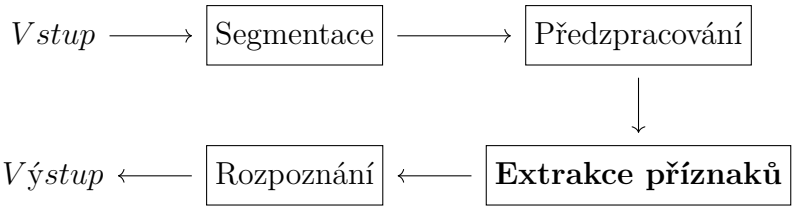
3.2 Princip OCR analýzy

Hlavním cílem v rozpoznávání znaků je najít a určit třídy vzorů, které se mohou objevit a jak vypadají [20]. V *OCR* jsou za vzory považovány tištěná či ručně psaná písmena (případně čísla a další speciální symboly) a jako třídy jednotlivé znaky dané abecedy.

Učení probíhá zjednodušeně následujícím způsobem. Nejprve je vytvořen anotovaný korpus příkladů všech relevantních znaků. Tento korpus je předán programu, který si na základě něho vybuduje modely jednotlivých tříd. Během rozpoznávání neznámého znaku se zahájí porovnání se všemi třídami. Třída, která nejvíce odpovídá neznámému znaku bude vybrána.

3.2.1 Komponenty OCR systému

OCR je systém složený z komponent, které ukazuje obrázek 3.1. Jednotlivé komponenty budou nyní popsány.



Obrázek 3.1: Komponety OCR systému [20]

Vstup

V případě *OCR* jsou vstupními daty nejčastěji naskenované dokumenty. Formát souborů je většinou **pdf**, případně některý z bitmapových formátů (např. **png**).

Segmentace

Segmentace je proces, který určuje prvky obrazu. Je potřeba najít oblasti dokumentu, které obsahují textové informace a odlišit je od ostatního obsahu dokumentu (např. grafů či obrázků).

Dochází tedy k izolaci znaků a slov. Většinou jsou segmentována slova na úrovni jednotlivých znaků, které jsou posléze rozpoznány individuálně. Obvykle se izoluje každá souvislá oblast [20]. Izolovat znaky je většinou jednoduché, ale mohou nastat problémy související s kvalitou vstupu. Mezi takovéto problémy patří:

- **Fragmentované znaky**
  Při skenování může dojít k situaci, kdy dva znaky (výjimečně i tři znaky) se mohou jevit jako jeden symbol. Může nastat i opačná situace, kdy se jeden znak může jevit jako více znaků. Obě tyto situace ukazuje následující obrázek 3.2. Pokud nastane tato situace, tak je potřeba upřednostnit ty metody extrakce příznaků, které budou odolné vůči fragmentaci.

Obrázek 3.2: Příklad fragmentovaných znaků [8]

- **Odlišení šumu**
  Díky tomu, že tečky, háčky a čárky nejsou souvisle spojeny s písmeny, mohou být považovány za šum. Opět může nastat i opačná situace, kdy šum v okolí

textu se může chybně zpracovat jako diakritika. Tuto situaci lze řešit pomocí technik předzpracování (viz dále).

- **Chybná identifikace obrázku či piktogramu**
  Tento problém nastane, když jsou k rozpoznávání zvoleny oblasti, které neobsahují text. Jedná se především o obrázky či různá obrazová sdělení (piktogramy). Existuje také opačný případ, kdy je text úzce spojen s obrázkem, a tak nemusí být zahrnut k rozpoznávání. Jako předchozí dva problémy, tak i tento lze řešit sofistikovaným předzpracování či vhodným výběrem extrakce příznaků.

Předzpracování (preprocessing)

V důsledku skenování nebo kvality vstupu dochází k šumu. Některé typy šumu lze eliminovat pomocí předzpracování, konkrétně pomocí vyhlazování (tzv. *smoothing*) a normalizace. Normalizace zahrnuje úpravy symbolu, díky kterým lze znak převést na uniformní velikost a tvar. Jedná se především o transformace a rotace. Cílem vyhlazování je eliminovat potenciální defekty symbolu, které by mohly být špatně interpretovány a v konečném důsledku by mohlo dojít k chybnému rozpoznání znaku. Příklad normalizace a vyhlazování je vidět na obrázku 3.3.

Obrázek 3.3: Vyhlazení a normalizace symbolu [8]

Extrakce příznaků

Extrakce příznaků je nejdůležitější úloha z celého systému. Cílem extrakce příznaků (*feature extraction*) je zachytit klíčové charakteristiky symbolů.

Podle publikace [8] patří mezi základní techniky extrakce příznaků následující metody.

- **Distribuce bodů**
  Extrakce příznaků probíhá na základě statistické distribuce bodů. Typická technika je například zónování (*zoning*) – viz obrázek 3.4. Počty černých bodů (*pixelů*) v jednotlivých zónách slouží jako příznaky.

Obrázek 3.4: Příklad zónování [8]

- **Transformace**
  Technika extrahuje příznaky tak, že transformuje matici bodů matematickou funkcí (např. *fourierova transformace*). Rozpoznávání probíhá na základě charakteristiky funkce.

- **Strukturální analýza**
  Tato technika analyzuje a popisuje strukturu znaku. Jedná se například o počet křížení či analýzu smyček ve znaku. Strukturální analýza je v porovnání s ostatními technikami více odolná vůči šumu a různým stylům písma.

- **Porovnávání vůči šabloně**
  Z uváděných metod extrakce příznaků je tato metoda nejjednodušší. Porovná se matice bodů vstupního znaku s prototypem každé třídy. Nejedná se zcela o extrakci příznaků a tato technika je spíše spojena s fází rozpoznávání, než s fází extrakce příznaků.

Rozpoznávání

Poslední komponentou *OCR* systému je již samotné rozpoznávání. Na základě příznaků a zvolené metody jejich extrakce se zvolí třída. Součástí této komponenty je spojení rozpoznaných písmen do slov a často také detekce a případně opravování chyb. Výstupů *OCR* může být více a můžou být i ohodnoceny pravděpodobnostmi.

4 Analýza OCR systémů

V této kapitole budou analyzovány a popsány dostupné nástroje na optické rozpoznávání znaků.

Existuje celá řada placených i *open source* programů. Při hledání vhodného nástroje byl kladen důraz na to, aby nástroj především uměl kvalitně rozpoznávat text v českém jazyce, byl *open source* a byl podporován operačními systémy Windows i Linux.

Analyzovány byly i některé komerční *OCR* systémy. U nich testování probíhalo pouze na anglických textech, protože u některých zkušebních verzí testovaných nástrojů nebyly k dispozici další jazykové moduly. Testovány byly i možnosti použití nástroje jako knihovny (součást programu v Javě).

Následuje stručný popis analyzovaných systémů.

4.1 Tesseract

Na začátku je třeba zdůraznit, že nástroj *Tesseract* slouží jako *OCR* jádro pro další grafické nástroje (např. *VietOCR*), což mimo jiné svědčí o jeho kvalitě. *Tesseract* navíc obsahuje rozsáhlou sadu jazyků (respektive sadu trénovacích znaků pro jednotlivé jazyky), včetně čínštiny. *Tesseract* lze použít buď jako samostatný program nebo jako knihovnu.

V případě použití jako knihovny v jazyce Java (*Tess4j*) je snadné *Tesseract* použít a zakomponovat do programu.

Nástroj umožňuje výstup z *OCR* analýzy v několika formách:

1. čistý text, podle *Tesseractu* je vybrán nejpravděpodobnější výsledek rozpoznávání,

2. *hOCR* výstup – podrobný HTML výstup společně s tzv. **confidence** (míra jistoty jednotlivých slov ohodnocená pravděpodobností),

3. pravděpodobnostní mřížka (pozice a varianty písmen společně s **confidence** – mírou jistoty slov – pravděpodobností).

Poslední uvedený výstup není standardní součástí rozhraní příkazového řádku[1] a pro zajištění tohoto výstupu je nutné k *Tesseractu* přistupovat programově pomocí **TessBaseAPI** (bude podrobně vysvětleno dále v implementační části). Výstup ve formě pravděpodobnostní mřížky je z hlediska opravování chyb nejvhodnější.

[1] CLI – Command Line Interface

4.2 Asprise OCR

Tento nástroj je dostupný jako knihovna pro celou řadu programovacích jazyků (Java, C#, C/C++, VB, .NET, Python). Ačkoliv jeho placená verze podporuje mnoho jazyků (včetně češtiny), neplacená verze umožňuje pouze rozpoznávání angličtiny (eng), španělštiny (spa), portugalštiny (por), němčiny (deu) a francouzštiny (fra). Rovněž se také záměrně nahrazují některé znaky znakem *.

Anglické texty dokáže zpracovat *Asprise* velmi dobře. Výhoda tohoto SW je, že je možné použít nástroj jako knihovnu pro velké množství programovacích jazyků a jeho integrace do programu v Javě je snadná a intuitivní.

Asprise OCR byl testován přímo proti *Tesseractu* a i když dosahoval horších výsledků, je jeho použitelnost a kvalita na velice dobré úrovni.

4.3 OmniPage

Systém *OmniPage* umožňuje už v základní verzi detekci jazyka zpracovávaného dokumentu. Tento komerční systém patří ke špičce *OCR* nástrojů, především díky svému výkonu, přesnosti a možnosti zachovávat vlastnosti původního skenovaného textu (např. barvu a typ písma).

Instalace a používaní tohoto software je velice intuitivní. Program je podporován pouze operačním systémem Windows. Pokud by byl prioritní požadavek výkon a přesnost, tak tato placená varianta ze všech uvedených systémů bude nejvhodnější.

4.4 Další OCR systémy

Dále budou uvedeny ostatní zkoušené *OCR* systémy. Některé z nich nesplňují podmínku podpory pro oba výše zmíněné operační systémy. Jsou zde uvedeny i nástroje, které mají pouze podobu grafického či konzolového programu a není možné je jednoduše použít jako komponentu programu (knihovnu).

- **pdf-XChange Viewer**
  Jedná se o grafický nástroj pro systém Windows na pokročilé prohlížení pdf dokumentů, který umožňuje *OCR* analýzu. V základní verzi umožňuje *OCR* analýzu pouze pro některé jazyky (angličtina, francouzština, němčina a španělština). Pro další jazyky je nutné stáhnout rozšiřující balíček.

- **GOCR**
  Jedná se o konzolovou aplikaci, která je zcela *open source*. Dostupné jsou binární spustitelné soubory pro Windows i Linux. Program umožňuje zpracování i čárových kódů. Výhodou tohoto nástroje je jednoduchost a flexibilita, vše je připraveno na rychlé spuštění. Programu nicméně činí problémy rotace textu

a jiné než bílé pozadí textu. Nástroj je pomalejší a méně přesný než jiné testované *OCR* nástroje.

- **OCRopus**
  Tento *OCR* nástroj je podporovaný pouze operačním systémem Linux. V dřívějších verzích používal tento nástroj jako jedinou *OCR* komponentu *Tesseract*, ale nyní používá vlastní *OCR* systém. Nástroj rovněž využívá komponentu jazykového modelu za účelem zvýšení přesnosti rozpoznávání. Program dosahuje horších výsledků u zpracování ručně psaného textu.

- **SimpleOCR**
  Nástroj na *OCR* analýzu pro operační systém Windows. V základní verzi umožňuje zpracování tištěného textu a zkušební verzi programu pro rozpoznání ručně psaného textu. V komerční verzi jsou přidány další funkcionality, jako rozpoznávání nestandardních fontů či barev.

- **Readiris**
  V případě tohoto nástroje se jedná čistě o komerční, ale výkonný a přesný *OCR* nástroj dostupný pouze pro operační systémy Windows a Mac OS. Jeho výhodou je podpora češtiny včetně lokalizace samotné aplikace (pro verzi 11).

- **Nicomsoft OCR**
  Nicomsoft OCR je *OCR* systém podporovaný operačními systémy Windows i Linux. Nástroj je možné používat jako samostatný program či jako komponentu (knihovnu) pro programy psané v programovacích jazycích C#, Java, .NET aj.

Na závěr této části ještě uveďme, že *OCR* analýzu umožňuje i program z kancelářského balíčku *Microsoft Office – Microsoft Office Document Imaging*, případně program *Adobe Acrobat*. Nicméně tyto systémy nejsou pro tuto práci vhodné.

4.5 Výsledky analýzy

Na základě analýzy byl vybrán nástroj *Tesseract*, protože z neplacených variant dosahoval nejlepších a nejpřesnějších výsledků. Splňuje rovněž všechny podmínky, uváděné v počátku této kapitoly.

Je třeba zdůraznit, že ani Tesseract nedosahuje 100 % výsledků a v rámci zlepšení přesnosti je nutné použít předzpracování a techniky na opravu chyb.

5 Analýza vyhledávacích nástrojů

Problematika vyhledávání a indexace byla stručně vysvětlena ve druhé kapitole *Indexace a vyhledávání*. Tato kapitola se zaměří na existující vyhledávací nástroje a jejich porovnání.

5.1 Lucene

Apache Lucene je *open source* vyhledávací knihovna navržená tak, aby poskytovala relevantní výsledky a zároveň vysoký výkon. Je psaná v programovacím jazyce Java a poskytuje *API* (*Application Programming Interface*) pro vyhledávání a jazykovou analýzu [4].

Lucene je komplexní vyhledávací nástroj, který poskytuje řadu užitečných funkcí. Nyní budou popsány jednotlivé poskytované funkce a posléze celkový pohled na architekturu.

5.1.1 Jazyková analýza

Tato komponenta si klade za cíl transformovat obsah dokumentu do takové podoby, aby následné fáze zpracování (indexace a vyhledávání) byly co nejefektivnější. Jazyková analýza se dělí do tří fází [4].

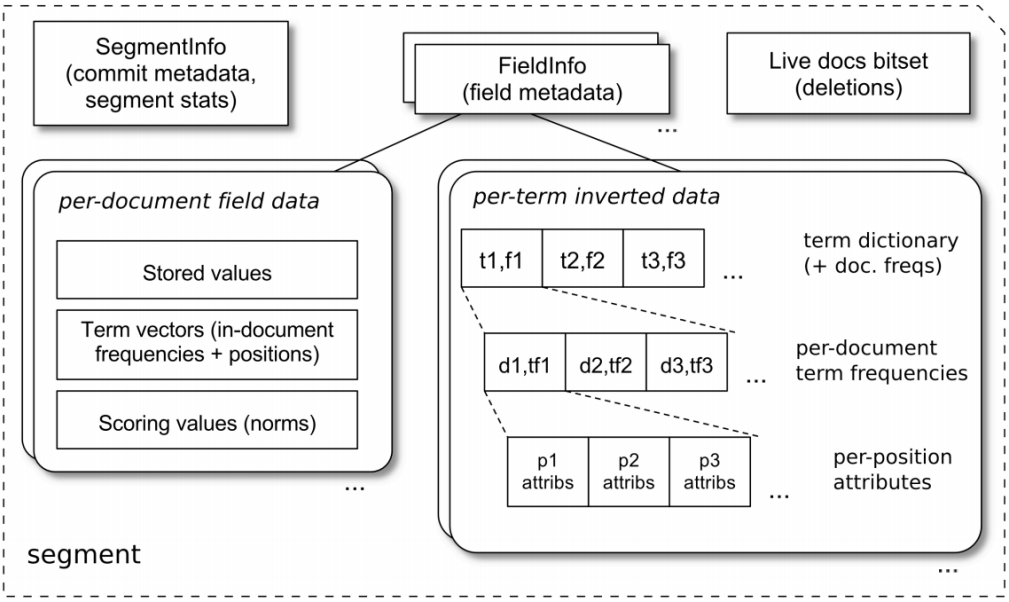
1. Normalizace a volitelná filtrace znaků (např. odstranění diakritiky).
2. Tokenizace.
3. Filtrace tokenů – stemming, lemmatizace, odstranění *stop-slov*, vytvoření *N-gramů*.

Jednotlivým fázím analýzy se věnovala druhá kapitola *Indexace a vyhledávání* na straně 4.

5.1.2 Indexace

Lucene používá pro vnitřní reprezentaci indexu invertovaný seznam. Dokumenty jsou v Lucene ve formě seřazených seznamů s položkami. Každá položka obsahuje jméno, hodnotu, váhu (používaná později pro vyhodnocení výsledků hledání) a další atributy v závislosti na typu položky. V procesu indexace je dokumentům přiřazeno celé číslo.

Celý index se v Lucene dělí na menší části, tzv. segmenty. Každý segment má v sobě vlastní index. Lucene poté vyhledává postupně napříč všemi segmenty. Schéma segmentu ukazuje obrázek 5.1.

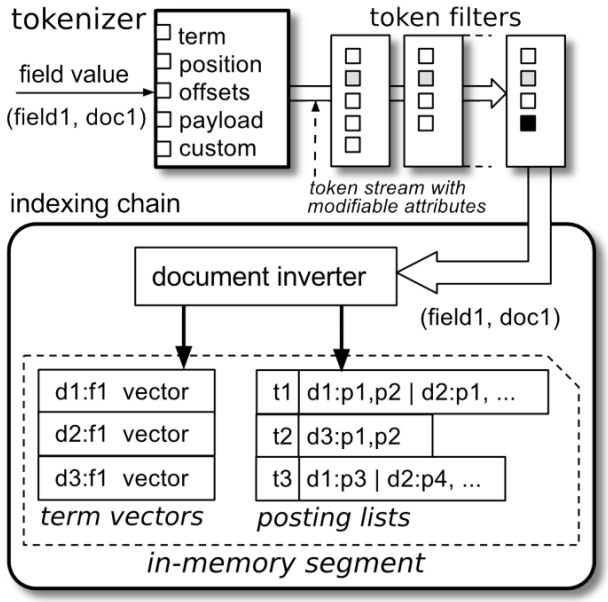


Obrázek 5.1: Struktura Lucene segmentu [4]

Segment obsahuje nejen vlastní invertovaný seznam (index), ale i další informace (metadata).

Podstatná vlastnost segmentů je, že jakmile se segment jednou vytvoří, tak už se dále nemění. Při přidávání nového dokumentu či aktualizace dokumentu se vytvoří nový segment. Periodicky se ale segmenty slučují do větších, aby se minimalizoval počet částí velkého indexu [4]. Výhodou tohoto přístupu je stabilita, snadnější záloha a aktualizace, protože není třeba neustále měnit celý index v případě přidávání nových dokumentů.

Proces indexace v Lucene ukazuje obrázek 5.2 na straně 16. Z komponenty *tokenizer* vychází *token stream*, který je dále zpracován a pomocí tzv. *indexing chain* je vytvořen segment. Na obrázku jsou rovněž znázorněny **filtry tokenů** (*token filters*), které odpovídají jednotlivým fázím jazykové analýzy (viz část 5.1.1 *Jazyková analýza*).



Obrázek 5.2: Proces indexace [4]

5.1.3 Dotazy

Pro vyhledávání a dotazy poskytuje Lucene tzv. *Query parser* – nástroj na parsování dotazů. Lucene nenabízí žádný speciální jazyk na psaní dotazů (*Query Language*), ale používá, v rámci *Query Model*, objekty dotazu [4]. Složitější dotazy je možné vytvářet programově nebo přes *Query parser*.

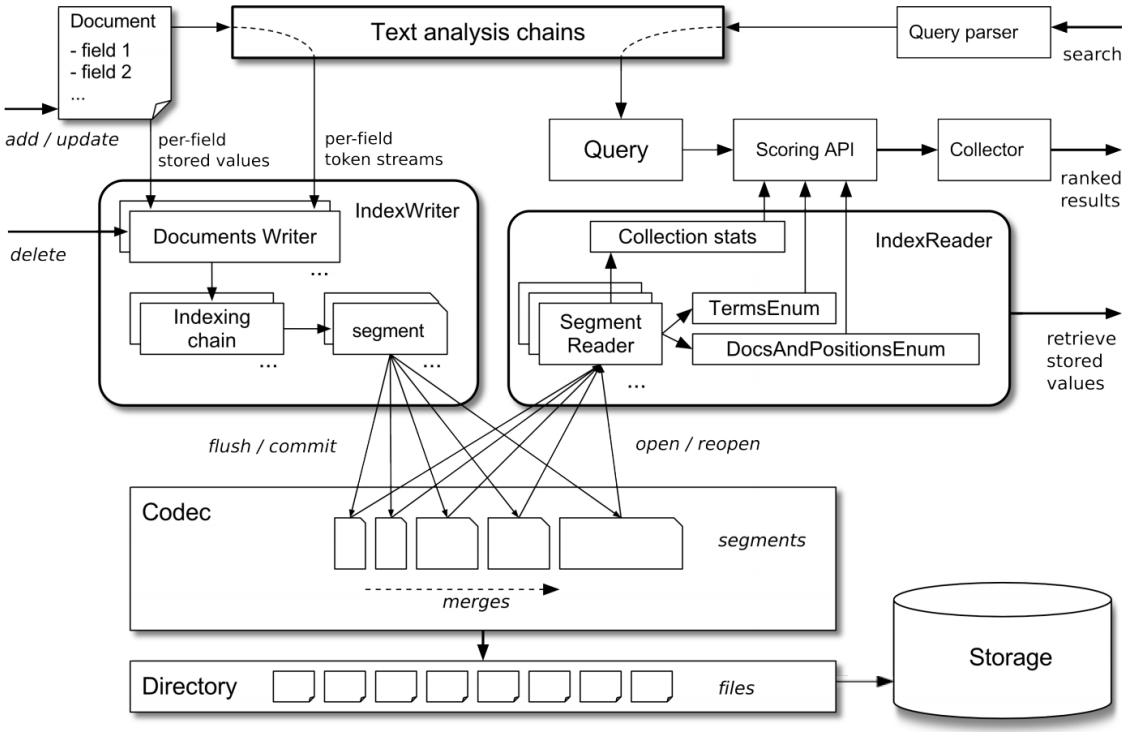
Kromě řady předdefinovaných dotazů nabízí Lucene následující typy dotazů:

- booleovské dotazy typu AND, OR a NOT (jednotlivé klauzule mohou být i další dotazy),
- dotazy na jednotlivé termy ve specifickém poli (*field*),
- dotazy na fráze i na pozice (např. hledané dva slova, před kterými je N jiných),
- tzv. *wildcard* dotazy (regulární výrazy).

Query parser transformuje dotaz do stromové struktury (tzv. *Query tree*) a poté se již dotaz vyhodnocuje procházením jednotlivých segmentů. Spočítá se skóre listů stromu na základě podobnosti a dalších technik evaluace dotazu.

5.1.4 Architektura Lucene

Celkový pohled na architekturu Lucene ukazuje obrázek 5.3. Obrázek znázorňuje všechny výše zmíněné komponenty a dává do souvislosti popsané funkce.



Obrázek 5.3: Architektura Lucene [4]

Lucene je dnes pravděpodobně nejpoužívanější vyhledávací knihovna. Je třeba zdůraznit, že dva největší a nejpoužívanější vyhledávací systémy současnosti (*Elasticsearch* a *Apache Solr*) jsou postaveny na základech knihovny Lucene. Tato skutečnost rozhodně svědčí o vysoké kvalitě této vyhledávací knihovny.

Na dalších stránkách budou popsány dva již zmíněné vyhledávací systémy *Elasticsearch* a *Apache Solr*.

5.2 Elasticsearch

Elasticsearch je distribuovaný, škálovatelný nástroj na vyhledávání a analýzu textu v reálném čase. Umožňuje rychle procházet a zkoumat data a poskytnout efektivní systém na *fulltextové* i strukturované vyhledávání (případně jejich kombinaci).

Elasticsearch využívají k vyhledávání například internetové služby *Wikipedia*, *Stack Overflow* nebo *GitHub* [10].

Elasticsearch funguje jako služba, která je dostupná standardně na portu 9200 a 9300. Při komunikaci s klientskou aplikací se používá formát *JSON* (*JavaScript Object Notation*). Po startu služby je k dispozici uzel (*tzv. node*), se kterým lze komunikovat. Uzel můžeme chápat jako běžící instanci Elasticsearch. Elasticsearch je možné spustit i v *clusteru* (skupina uzlů) a poskytnout tak výhody distribuovaného řešení.

5.2.1 Reprezentace dokumentů

Elasticsearch využívá pro reprezentaci dokumentů formát *JSON*. Ukázka jednoduchého objektu, který je reprezentovaný formátem *JSON* je následující.

```
{
    "id": 1,
    "first_name": "Jack",
    "last_name": "Smith",
    "price": 2000,
    "age" : 35,
    "tags": ["Dangerous", "Powerfull", "Nice"]
}
```

Dokument je strukturovaně popsán dvojicí položka a hodnota. Jednotlivé položky je možné vnořovat a tím lze tvořit i složitější reprezentace objektů.

5.2.2 Vyhledávání

Po indexaci dokumentů je *node* (případně *cluster*) připraven na vyhledávání pomocí dotazů. V případě dotazu není výsledkem pouze číslo dokumentu (id), ale sám celý dokument (ve formátu *JSON*). Elasticsearch poskytuje flexibilní a bohatý dotazovací jazyk *Query DSL*, který umožňuje sestavovat i komplikované a robustní dotazy [10].

Mějme uložené a zaindexované dokumenty osob. Pokud by byl úkol vrátit všechny dokumenty (osoby), které se jmenují Smith a jsou starší 30 let, tak dotaz bude reprezentován způsobem uvedeným na obrázku 5.4. První část dotazu je filtr (*filter*). Druhá část je již standardní dotaz na shodu (*match*).

Elasticsearch je velice výkonný nástroj pro vyhledávání a patří mezi nejlepší *open–source* vyhledávací systémy.

```
{
  "query" : {
      "filtered" : {
         "filter" : {
            "range" : {
               "age" : { "gt" : 30 }
            }
          },
          "query" : {
             "match" : {
                 "last_name" : "smith"
             }
           }
      }
   }
}
```

Obrázek 5.4: Ukázka reprezentace dotazu v systému Elasticsearch [10]

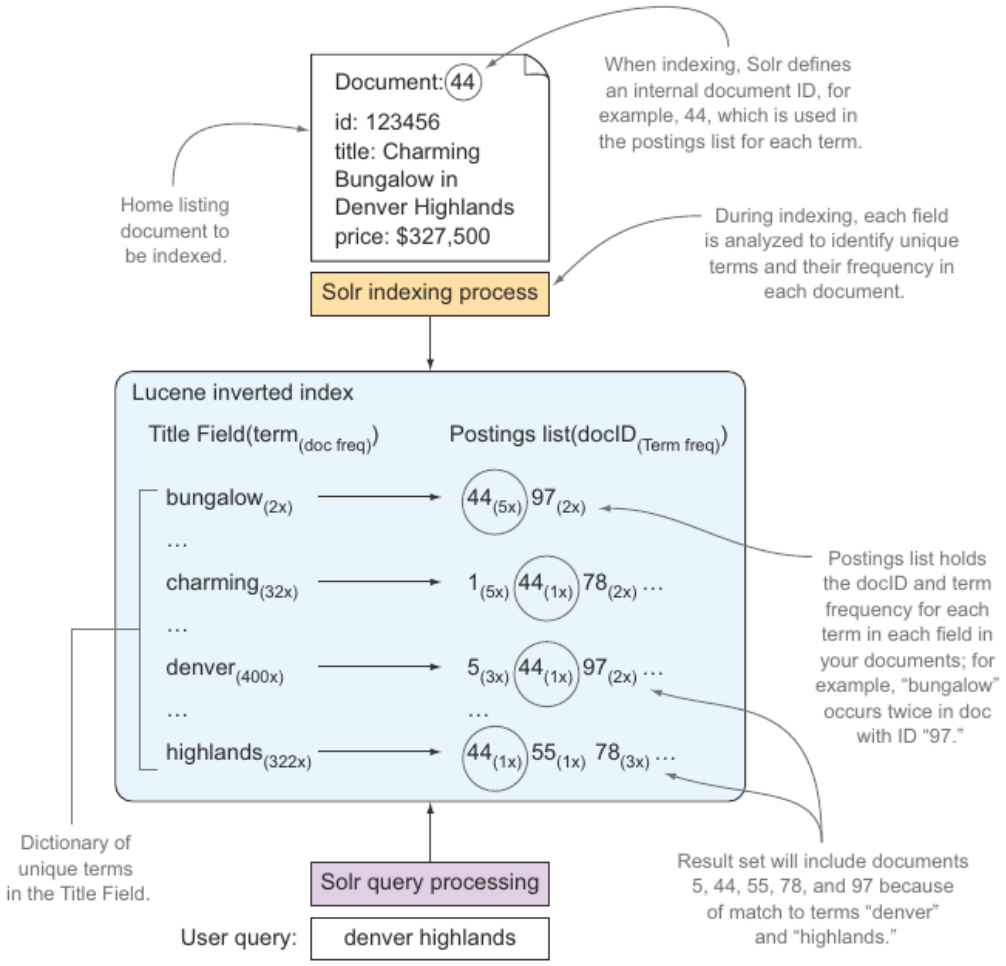
5.3 Apache Solr

Apache Solr je vyhledávací nástroj postavený na základech vyhledávací knihovny Lucene. Solr je webová aplikace vytvořená v programovacím jazyce Java (konkrétně používá technologii *Java servletů*) [11].

Proces indexace a vyhledávání znázorňuje obrázek 5.5. V prostřední části schématu je zobrazen invertovaný seznam (*index*) vyhledávací knihovny Lucene. Při indexaci dokumentu si Solr definuje jedinečné interní *Document ID*. Obrázek ukazuje i příklad dotazu a naznačení výsledné množiny dokumentů, které dotazu vyhovují.

Apache Solr je možné také spustit distribuovaně, ale na rozdíl od Elasticsearch je nutné použít technologii *Apache ZooKeeper*. Apache ZooKeeper je *open–source* služba, která umožňuje vysoce spolehlivou koordinaci distribuovaného řešení [12]. Je to centralizovaná služba uchovávající informace o konfiguraci, pojmenování a synchronizaci distribuovaného řešení. Poskytuje jednoduché a vysoce výkonné jádro pro budování komplexní koordinace služeb. Zahrnuje takové elementy, jako skupinové posílání zpráv, sdílené registry a distribuované služby v oblasti replikace a zámků.

Existuje řada připravených rozhraní pro přístup k aplikacím v systému *ZooKeeper*. Tato technologie garantuje pořadí vykonávání požadavků od jednotlivých klientů. Vysoký výkon demonstruje skutečnost, že *ZooKeeper* by měl být schopen zpracovávat desítky až stovky tisíc požadavků (transakcí) za sekundu [12].

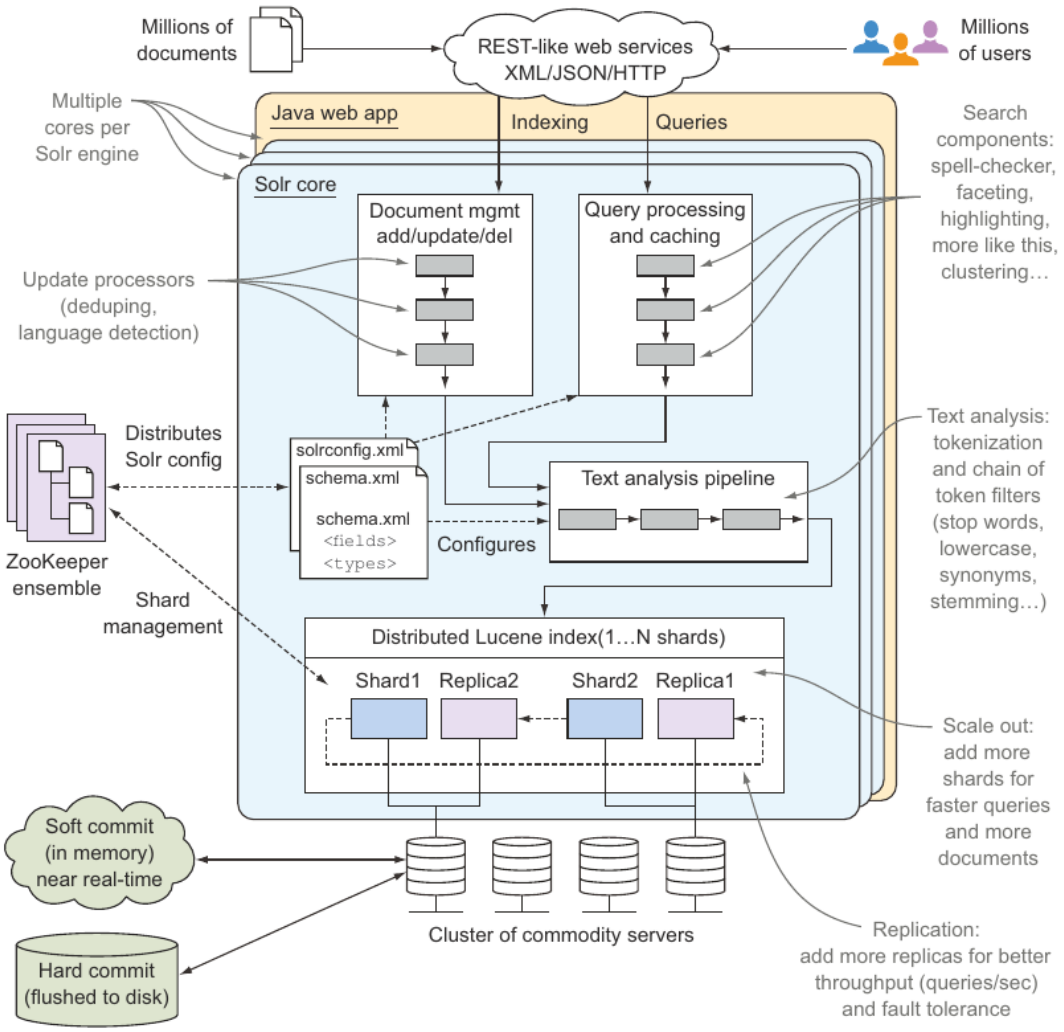


Obrázek 5.5: Ukázka procesu indexace a vyhledávání v systému Solr [11]

Vnitřní implementace *ZooKeeperu* je realizována pomocí tzv. **pipelined architecture**. Díky tomu je možné zpracovávat velké množství požadavků při zachování poměrně nízké doby odezvy (latence).

5.3.1 Komponenty Apache Solr

Běžící instance Solr obsahuje jeden (případně i více) tzv. *Solr core*. Každý *core* obsahuje vlastní *index* a komponenty pro zpracování dotazu, pro textovou analýzu a pro práci s dokumenty. Souhrnný pohled na hlavní komponenty a jejich vazby ukazuje obrázek 5.6.



Obrázek 5.6: Hlavní komponenty Apache Solr [11]

5.4 Srovnání analyzovaných systémů

V této části budou srovnány výše uvedené vyhledávací systémy. Srovnávány budou především následující hlediska.

- výkon,
- požadavky na diskový prostor,
- napojení na další aplikace (v různých programovacích jazycích).

Souhrnné srovnání obou výše popsaných vyhledávacích systémů znázorňuje tabulka 5.1. Tabulka zobrazuje i porovnání s knihovnu Lucene. Další text bude ale zaměřen především na srovnání systémů Solr a Elasticsearch.

	Elasticsearch	Apache Solr	Lucene
Formát komunikace	JSON	XML, CSV, JSON	-
HTTP Rest API	Ano	Ano	Ne
Oficiální klientské knihovny	Java, Groovy, PHP, Ruby, Perl, Python, .NET, Javascript	Java	-
Binární API	TransportClient Thrift (přes plugin)	SolrJ	Lucene API
Distribuce	Elasticsearch uzly	Nutný ZooKeeper	Ne
Replikace	Přes jednotlivé uzly	Ano	Ne
Webové rozhraní	Kibana a další pluginy	Automaticky	Ne
Úprava tokenizeru pro jeden dotaz nebo dokument	Ano	Ne	Ne
Více indexů	Ano	Ano	Ne
Dotazovací jazyk	Ano - Query DSL	Ne - nutné použít syntaxi Lucene nebo dotaz vytvořit programově	Ne
Formát souboru konfigurace	YAML	JSON	-
Podpora JMX	Pouze přes API	Ano	-

Tabulka 5.1: Srovnání vyhledávacích systémů [22]

Dotazy a výkon systémů

Dotazovací aparát systému Elasticsearch *Query DSL* je vysoce komplexní nástroj, který poskytuje širší možnosti než dotazovací aparát Apache Solr. Z hlediska *full-textového* vyhledávání jsou systémy srovnatelné.

Oba systému se vyvíjely současně a vzájemně se tedy ovlivňovaly a učily jeden od druhého. Díky této skutečnosti jsou oba systémy z výkonnostního hlediska velmi podobné.

Nároky na diskový prostor

Podíváme-li se na nároky na diskový prostor po instalaci, tak zjistíme, že *Elasticsearch* má zhruba třetinové nároky oproti systému *Solr*. Je to ale hlavně dáno tím, že *Elasticsearch* ze začátku poskytuje pouze nejnutnější funkcionalitu, naproti tomu *Solr* má od začátku k dispozici širokou škálu pluginů a různé funkcionality.

Podle autorů článku [17] je dále výhoda *Elasticsearch* v tom, že je jednodušší, snadnější na použití a ovládání. Autoři dále vidí nejpodstatnější rozdíl mezi systémy v tom, že *Solr* se soustřeďuje na rozvoj kvalitní indexace a vyhledávání, zatímco *Elasticsearch* vynakládá úsilí na zlepšení subsystému datové analýzy. Oba systémy mají srovnatelný výkon na středně velké testovací množině. Oba systémy jsou rovněž schopné indexovat nejenom textové dokumenty, ale i pdf či MS Word dokumenty[1] [17].

5.5 Výsledek analýzy

Žádné velké výkonnostní rozdíly mezi Elasticsearch a Apache Solr nebyly zaznamenány. Pro tuto práci byl nicméně zvolen Apache Solr, protože poskytuje dle mého názoru lepší integraci s programem v Javě (prostřednictvím knihovny *SolrJ*). Rovněž shledávám úroveň a přehlednost dokumentace na lepší úrovni než u systému Elasticsearch.

[1] V případě *Elasticsearch* je nutné použít externí modul (*plugin*)

6 Detekce a oprava chyb

Ve třetí kapitole *Optické rozpoznávání znaků* byly zmíněny problémy a chyby, které způsobuje optické rozpoznávání. Podstatnou částí této práce je návrh metod detekce a oprav těchto chyb. V této kapitole budou nejprve popsány typy chyb a poté jednotlivé přístupy a řešení.

6.1 Typy chyb

Vlivem šumu či fragmentovaných znaků (viz obrázek 3.2) dochází k misinterpretaci znaků (chybnému vyhodnocení). Je zřejmé, že čím horší kvalita vstupního obrázku, tím se bude generovat více chyb. Na počet chyb má vliv i rozlišení obrázku. V případě *OCR* systému *Tesseract* je v dokumentaci (viz [1]) uváděno, že optimální rozlišení je 300 DPI[1] (detailněji viz následující kapitola, část *Vliv rozlišení obrázku na chyby* na str. 35).

Nejčastější chyby, pozorované z několika testovacích vstupů zpracovávaných systémem *Tesseract*, jsou následující záměny podobně vypadajících znaků.

- l → 1
- D → O
- O → 0
- l → 1
- J →]
- h → n
- f → i
- . → ,
- S → 5

Navržené řešení by mělo být schopné detekovat a opravovat výše uvedené příklady chyb, ale i obecné chyby pramenící ze šumu či fragmentovaných znaků. Nyní bude následovat popis jednotlivých možných řešení, jejich výhody a nedostatky.

6.2 Slovníkový přístup a vzdálenost slov

Tato jednoduchá metoda využívá slovník slov daného jazyka. Jednotlivá slova se porovnávají napříč celým slovníkem. Pokud je nalezena shoda, je slovo s největší pravděpodobností správně. Pokud shoda nalezena není, je ve slově s velkou pravděpodobností chyba.

Výše popsanou techniku lze použít k detekci chyb. Nejjednodušší způsob opravy je nalezení nejbližšího slova. K úspěšnému nalezení nejbližšího slova je nutné si

[1] Dots Per Inch – počet obrazových bodů (pixelů) na palec (25,4 mm).

definovat metriku, podle které se budou slova porovnávat. Vhodná a použitelná metrika je vzdálenost slov (*Levensteinova metrika*).

6.2.1 Levensteinova metrika

Mějme dva řetězce znaků A a B. *Levensteinova metrika* $\delta(A, B)$ se definuje jako minimální cena všech sekvencí operací (vložení, odstranění, záměna), které transformují řetězec A na řetězec B [23]. Dva řetězce jsou tedy totožné, pokud je jejich vzdálenost rovna 0. *Levensteinova metrika* je známá také jako *edit distance*. Uvažujme následující příklad.

```
Řetězec A: ALFA
Řetězec B: BETA
```

Vzdálenost výše uvedených řetězců $\delta(A, B)$ je rovna 3, protože potřebujeme minimálně 3 operace nahrazení znaků, které transformují řetězec A na B. Pomocí této metriky lze najít nejbližší slovo (slovo s nejmenší vzdáleností) a tím získat pravděpodobné původní správné slovo.

6.3 Pravidlový přístup

V sekci 6.1 *Typy chyb* uvedeny nejčastější záměny znaků při optickém rozpoznávání.

Při realizaci této metody je nutné mít k dispozici rovněž slovník. Vytvoří se sada pravidel, které se postupně aplikují na chybně rozpoznané slovo. Postupně se aplikují záměny uvedených znaků a kontroluje se podobnost se slovníkem.

Formálně lze tuto metodu označit za metodu využívající *Spelling error patterns* (viz [15]). Tyto techniky využívají nejen výše uvedené záměny znaků, ale i vzory v překlepech. Vzory v překlepech vycházejí z pozic kláves na klávesnici. Je například pravděpodobnější záměna znaků a a s než a a p. Časté chyby jsou rovněž záměna $y \rightarrow z$ a naopak, případně chyby v pořadí znaků (pro anglický jazyk například $receive \rightarrow recieve$).

Tato metoda může odhalit jednoduché chyby, které vzniknou v důsledku OCR zpracování. Nicméně na zcela obecnou chybu není tato metoda dostačující. Výhodou této metody je její rychlost.

6.4 Statistické jazykové modely

Metoda jazykových modelů je z uváděného výběru metod nejsložitější. Bude postupně popsána teorie jazykových modelů, jejich jednotlivé typy a měření jejich kvalit.

Jazykový model je pravděpodobnostní rozdělení řetězců slov $P(s)$, které se snaží odpovídat frekvenci, se kterou se každý řetězec $\mathtt{s} = \{s_1, ...s_N\}$ objevuje v přirozeném jazyce (např. jako věta) [7].

Cílem jazykových modelů je odhadnout pravděpodobnost výskytu libovolného řetězce slov. Jelikož je přirozený jazyk příliš komplikovaný, tak nelze tyto pravděpodobnosti odhalit pomocí pravidlového přístupu. Jazykový model je nutné natrénovat na a na základě toho požadované pravděpodobnosti odhadnout.

6.4.1 N–gramové jazykové modely

V praxi není možné vyjádřit podmíněné pravděpodobnosti nekonečně dlouhých řetězců textu. Proto je nutné při odhadování pravděpodobností pracovat s tzv. *N–gramy*. *N–gramem* se rozumí počet po sobě jdoucích slov. Použít lze *N–gramy* s libovolným stupněm, ale nejpoužívanější jazykový model je **trigramový**.

N–gramový jazykový model se stupněm jedna se označuje jako *unigramový*. Pro úplnost ještě dodejme, že existuje i *N-gramový* jazykový model se stupněm nula – tzv. **uniformní**. Uniformní jazykový model přiřazuje stejnou pravděpodobnost každému slovu ve slovníku [13].

Pro jakýkoliv jazykový model musí platit, že suma pravděpodobností všech slov ve slovníku podmíněná libovolnou historií je rovna 1 (rovnice 6.1).

$$\sum_{w_i \in W} = P(w_i | w_1^{i-1}) = 1 \tag{6.1}$$

Pro konkrétní výpočet pravděpodobnosti by bylo nutné znát frekvence výskytu jednotlivých posloupností slov napříč celým přirozeným jazykem. Z důvodu složitosti přirozeného jazyka nelze tyto četnosti určit, je možné je pouze odhadnout na základě reprezentativního výběru trénovacích dat. Pro odhad pravděpodobností se používá **metoda maximální věrohodnosti**.

6.4.2 Metoda maximální věrohodnosti

Metoda maximální věrohodnosti (*MLE – Maximum Likelihood Estimation*) je metoda, která umožní odhadnout pravděpodobnosti posloupnosti slov pouze na základě trénovacích dat. *MLE* odhad tedy nebere v úvahu apriorní pravděpodobnost slov či frekvenci výskytu jednotlivých posloupností.

Díky tomuto zjednodušení lze odhadnout pravděpodobnosti následujícím způsobem:

$$P(w_i | w_{i-n+1}^{i-1}) \approx \frac{c(w_{i-n+1}^{i})}{\sum_{w_i \in W} c(w_{i-n+1}^{i})} = \frac{c(w_{i-n+1}^{i})}{c(w_{i-n+1}^{i-1})} \tag{6.2}$$

kde c značí četnost *N–gramu* [6].

6.4.3 Měření kvality a vyhlazování jazykového modelu

Pro určení kvality jazykového modelu se nejčastěji používají metriky entropie a perplexita.

Entropii lze charakterizovat jako míru neurčitosti. Čím vyšší je entropie, tím se o dané problematice ví méně. Obecnou snahou je tedy minimalizovat entropii. Entropii přes posloupnosti slov w_1, ..., w_n lze vyjádřit jako:

$$H(w_1, ..., w_n) = -\sum_{w_1^n \in W} P(w_1^n) log_2 P(w_1^n) \tag{6.3}$$

K přesnému určení entropie je nutné mít k dispozici nekonečné množství slov daného jazyka (rovnice 6.4).

$$H(w_1, ..., w_n) = \lim_{n\to\infty} \frac{1}{n} H(w_1, ..., w_n) \tag{6.4}$$

Je evidentní, že kvůli složitosti jazyka nelze výše uvedenou podmínku splnit. Proto je nutné pravděpodobnosti použité ve vzorci odhadnout.

Entropii tudíž lze také pouze odhadovat, a to na základě tzv. *Shannonova-McMillanova-Breimanova teorému* (detailněji viz [16]). Teorém popisuje následující zjednodušení (rovnice 6.5).

$$H(w_1, ..., w_n) = -\lim_{n\to\infty} \frac{1}{n} log_2 \widetilde{P}(w_1^n) \approx -\frac{1}{n} log_2 \widetilde{P}(w_1^n) \tag{6.5}$$

Odhad pravděpodobnosti $\widetilde{P}(w_1^n)$ se určí podle jazykového modelu (*MLE* odhad).

Perplexita úzce souvisí s entropií a definuje se jako:

$$PP = 2^{H(w_1,...,w_n)} \tag{6.6}$$

Perplexita popisuje, z kolika možností by model vybíral, pokud by všechny *N–gramy* byly stejně pravděpodobné (tzv. **uniformní model**).

Je-li entropie rovna nule, tak perplexita vyjde rovna jedné (nejlepší možný případ). Jazykový model vybírá z právě jedné možnosti a z toho plyne, že si je maximálně jistý a nemůže udělat chybu. Problém nastává pokud vyjde nekonečná entropie (některý *N–gram* není nalezen). Problém nekonečné entropie řeší tzv. **vyhlazování** (*smoothing*).

Vyhlazování

Technika vyhlazování řeší nulové pravděpodobnosti u *N–gramů*, které jazykový model nezná. Jednoduchým příkladem vyhlazování může být tzv. **aditivní vyhlazování** (*additive smoothing*). U tohoto typu vyhlazování se zvýší četnost každého *N–gramu* o nenulové číslo δ (zpravidla 1). Úpravou vzorce 6.2 lze definovat odhad pravděpodobnosti *N–gramů* s aditivním vyhlazováním jako:

$$P(w_i|w_{i-n+1}^{i-1}) \approx \frac{c(w_{i-n+1}^{i}) + \delta}{c(w_{i-n+1}^{i-1}) + \delta|W|} \tag{6.7}$$

kde |W| je velikost slovníku trénovacích dat.

Nyní i pro neznámé *N–gramy* vyjde pravděpodobnost nenulová. Tímto jednoduchým způsobem lze vyřešit problém nekonečné entropie, ale nepozorovaným *N–gramům* se přiřazuje příliš mnoho pravděpodobnostního prostoru.

Sofistikovanější přístup k vyhlazování přináší **lineární interpolace**, která kombinuje více jazykových modelů.

Každý z jednotlivých jazykových modelů bude mít v této vyhlazovací metodě svoji váhu λ. Tyto váhy zajistí, aby se neopakoval problém vysoké pravděpodobností neznámých *N–gramů* u aditivního vyhlazování. Odhad pravděpodobnosti *N–gramu* lze vyjádřit následujícím vztahem:

$$P(w_i|w_1^{i-1}) \approx \sum_{k=1}^{K} \lambda_k \widetilde{P}_k(w_i|w_{i-n+1}^{i-1}) \tag{6.8}$$

Aby nebyla porušena podmínka sčítání pravděpodobností *N–gramů* do jedné, tak i zde musí platit, že součet všech k koeficientů λ musí být roven 1.

Koeficienty se odhadují pomocí **Expectation-Maximization** algoritmu.

Algoritmus Expectation-Maximization

Algoritmus má fázi *Expectation* a fázi *Maximization*. Tyto fáze se opakují, dokud není splněna podmínka na konvergenci. Průběh algoritmu lze shrnout do následujících několika bodů [6].

1. Nejprve jsou zvoleny počáteční váhy $\lambda_k^0 = K^{-1}$ pro všechny $1 \leq k < K$ a je určena hodnota ϵ, jakožto zastavovací práh.

2. Fáze `Expectation`: Výpočet očekávaných hodnot parametrů $\widehat{\lambda}_j^t$ podle:

$$\widehat{\lambda}_j^t = \sum_{w_{i-n+1}^i} \frac{\lambda_j^t \cdot \widetilde{P}_j(w_i|w_{i-n+1}^{i-1})}{P^{LI}(w_i|w_{i-n+1}^{i-1})} = \sum_{w_{i-n+1}^i} \frac{\lambda_j^t \cdot \widetilde{P}_j(w_i|w_{i-n+1}^{i-1})}{\sum_{k=1}^{K} \lambda_k^t \cdot \widetilde{P}_k(w_i|w_{i-n+1}^{i-1})} \tag{6.9}$$

   kde t označuje počet iterací.

3. Fáze `Maximization`: stanovení nových hodnot parametrů λ_j^{t+1} pomocí normalizace:

$$\lambda_j^{t+1} = \frac{\widehat{\lambda}_j^t}{\sum_{k=1}^{K} \widehat{\lambda}_k^t} \tag{6.10}$$

4. Při splnění podmínky:

$$|\lambda_j^{t+1} - \lambda_j^t| < \epsilon \tag{6.11}$$

   ukonči výpočet, jinak návrat k bodu 2.

Je potřeba zdůraznit, že veškeré výpočty pravděpodobností v *EM* algoritmu je třeba získat z jiných než trénovacích dat (tzv. *held-out* data) [5]. Pokud by výpočet probíhal na stejných datech jako při trénování (počítání četností *N–gramů*), tak by váha *N–gramového* modelu s největším stupněm (např. pro trigramový jazykový model) měla hodnotu blížící se 1, zatímco ostatní by se blížily 0.

6.4.4 N–gramový znakový jazykový model

Slovní statistické jazykové modely umožní určit na základě okolního kontextu (historie či následníků slov) pravděpodobnost výskytu slova. Není ale nutné se omezit pouze na slovní model. Stejný přístup lze aplikovat i na jednotlivé znaky a vytvořit znakový jazykový model.

Takto postavený model určí pravděpodobnost výskytu znaku na základě předcházejících či následných znaků. Díky znakovému jazykovému modelu lze zkvalitnit a zpřesnit *OCR* analýzu, protože je možno spočítat odhady pravděpodobností výskytů znaků v přirozeném jazyce z trénovacích dat.

Pro ověření úspěšnosti rozpoznávání a následné opravy chyb jsou použity metriky *Word Error Rate* a *Character Error Rate*.

6.5 Word Error Rate a Character Error Rate

Metrika *Word Error Rate* je odvozena od *Levensteinovy metriky* (viz část 6.2.1 *Levensteinova metrika* na straně 25) a je definována následujícím způsobem:

$$WER = \frac{S + D + I}{N_r} \tag{6.12}$$

kde N_r je počet všech slov v referenčních datech, S je počet nahrazených slov, D je počet chybějících (smazaných) slov a I je počet slov navíc oproti referenčním datům [19].

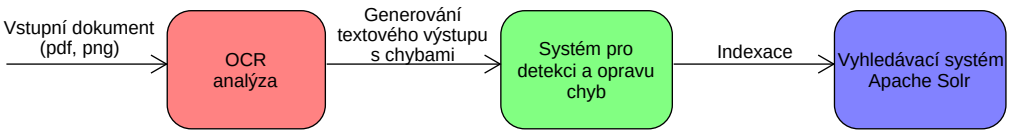
Je-li k dispozici zcela bezchybný výstup (totožný jako referenční data), počty S, D a I budou nulové a chybovost vyjde 0%.

Je evidentní, že je nutné mít referenční (tzv. *gold*) data vůči kterým bude výsledek zpracování porovnán. Metrika *CER* je definována stejným způsobem, jen se berou v úvahu písmena a ne celá slova. Tyto metriky byly zvoleny z důvodu dobré vypovídající hodnoty o výsledku rozpoznávání. Kromě těchto dvou metrik jsou ještě použity metriky

- **Accuracy** – počet správně rozpoznaných slov ku počtu slov v referenčních datech,
- **Word Accuracy** – doplněk k Word Error Rate (1 - WER).

7 Popis implementace

V této kapitole bude podrobně popsána implementace celého systému. Celkový pohled na jednotlivé dílčí části programu a jejich funkce ukazuje diagram na obrázku 7.1.



Obrázek 7.1: Zjednodušené znázornění práce systému

Do systému vstoupí dokument, který projde nejprve fází OCR analýzy. Tato analýza transformuje vstupní dokument na text, který projde fázemi detekce a opravy chyb. Text vzešlý z předchozí OCR analýzy tato fáze analyzuje a pokud v něm objeví chyby, tak se je pokusí opravit. Oprava chyb má několik fází. Výstupem OCR analýzy jsou kromě samotného rozpoznaného textu i pravděpodobnosti jednotlivých znaků u slov. Pomocí *Viterbiho algoritmu* a jazykových modelů jsou tyto pravděpodobnosti využity pro nalezení nejpravděpodobnější sekvence znaků daného slova.

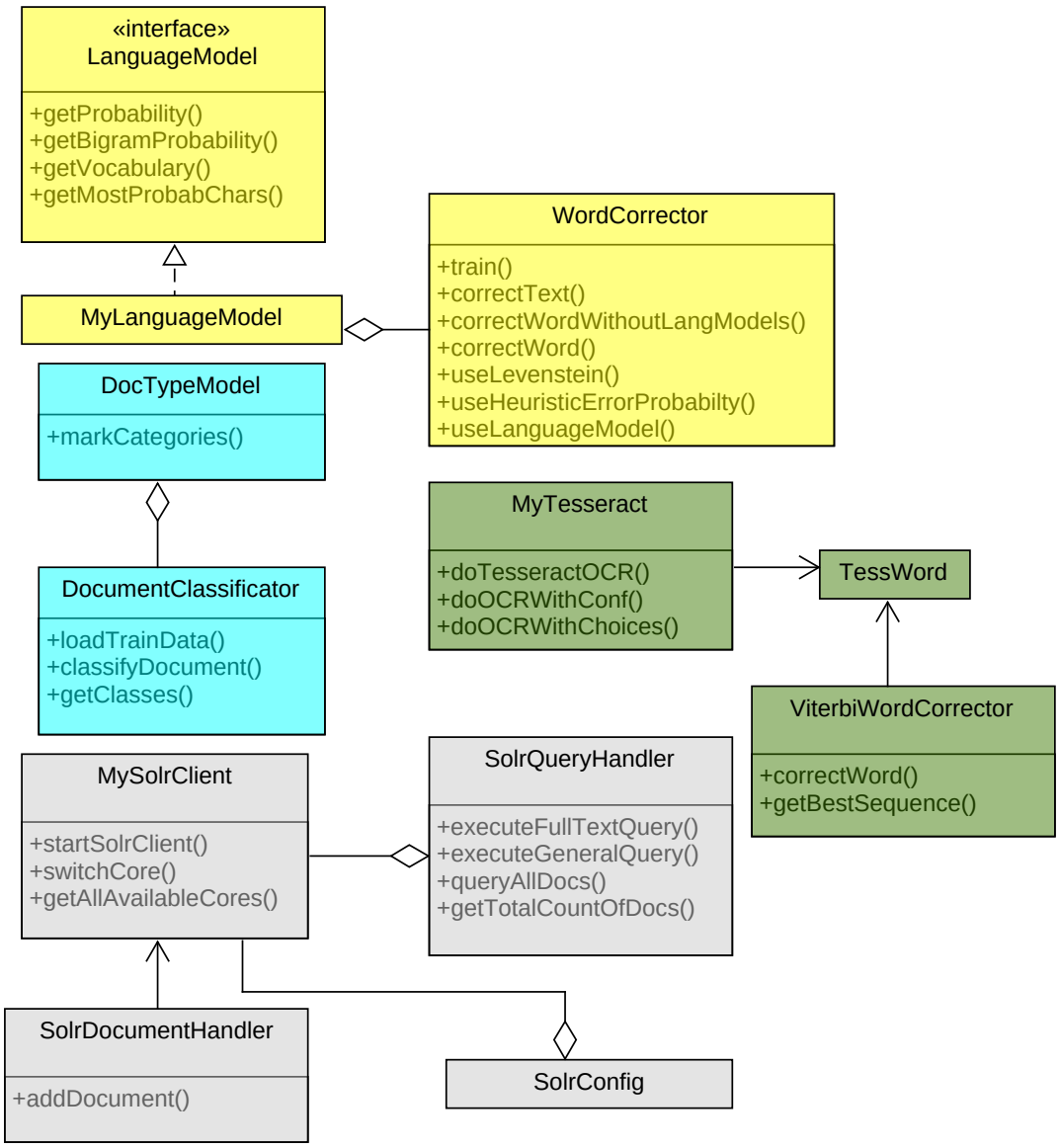
Opravená slova jsou poté seskupena do dokumentu, který se uloží do *Apache Solr* databáze. Vyhledávání probíhá v množině takto vzniklých dokumentů, a to na základě fulltextových dotazů.

7.1 Návrh systému

Ještě před začátkem popisu jednotlivých částí systému bude uvedeno schéma návrhu systému z hlediska tříd a objektů. Na obrázku 7.2 jsou znázorněny čtyři hlavní komponenty systému (barevně jsou seskupeny logicky související třídy). Z důvodu přehlednosti nejsou uvedeny všechny třídy v aplikaci, ale pouze ty nejdůležitější.

Žlutá část ve vrchní části obrázku znázorňuje implementaci jazykového modelu (třída `MyLanguageModel`) a objektu `WordCorrector`, který obsahuje metody na opravu textu. Zelená část v pravé části schématu popisuje OCR komponentu. Třída `MyTesseract` obsahuje metody pro OCR včetně výstupu z pravděpodobnostmi. Ve třídě `ViterbiWordCorrector` jsou metody pro hledání nejpravděpodobnější sekvence znaků ve slově. Obě tyto třídy používají objekt `TessWord`. Levá modrá část schématu popisuje klasifikátor dokumentů a jeho metody. Šedá část ve spodní části

schématu obsahuje třídy pro komunikaci s běžící instancí *Apache Solr*, přidávání dokumentů a tvorbu dotazů.



Obrázek 7.2: Diagram tříd

Následovat bude popis jednotlivých částí systému včetně stručných ukázek zdrojového kódu.

7.2 Implementace systému OCR

První částí systému, která bude popisována je implementace *OCR*. V analýze *OCR* nástrojů byl zvolen systém *Tesseract*. Tato část popíše knihovnu **Tess4j** (implemen-

tace *Tesseractu* v Javě) a její využití pro *OCR* analýzu.

Součástí zadání je informace, že vstup do systému bude dokument v podobě rastrových obrázků. Do této skupiny dokumentů lze řadit i pdf dokumenty i když tato skutečnost přímo nevyplývá ze zadání. V následujícím textu proto bude krátce popsána problematika pdf dokumentů.

7.2.1 Problematika pdf dokumentů

Základní podoba *Tesseractu* ovládaná přes příkazovou řádku umožňuje zpracovat pouze rastrové obrázkové soubory (např. `png`, `gif`, `tiff` či `bmp`). Základní podoba *Tess4j* interně zpracuje i pdf dokumenty. Pokud by bylo tedy cílem pouze rozpoznat text z pdf dokumentu, není potřeba žádných dodatečných úprav.

Vzhledem k potřebě další opravy chyb ze systému *Tesserarct* bude vhodné přistupovat přímo k rozhraní knihovny *Tesseract*. Tento přístup nelze aplikovat na pdf soubory. Je proto v případě použití pdf souboru jej převést na jiný (bitmapový) formát. K tomu je použita knihovna **Ghost4j**. Prostřednictvím této knihovny lze přes API využívat funkce programu Ghostscript. Jedna z funkcí je i převod pdf na rastrový formát png (detailněji viz dokumentace [2]).

Naznačení možného převodu souboru pdf na soubor png ukazuje ukázka kódu 7.1. Využívají se zde objekty `SimpleRenderer` a `pdfDocument` z balíčku org.ghost4j. Následující ukázka zpracuje i vícestránkový pdf dokument. V takovém případě se vytvoří pro každou stránku vlastní png soubor (viz cyklus v poslední části metody).

Knihovna **Ghost4j** umožňuje nejen převod pdf souborů na rastrový formát, ale i zadat s jakým rozlišením se má bitmapový soubor vytvořit. Výše uvedené metoda převodu tuto možnost využívá a pro účely projektu se převádí s rozlišením 300 DPI. Rozlišení a jeho vlivu na kvalitu *OCR* se věnuje text 7.2.3 *Vliv rozlišení obrázku na chyby* na straně 35.

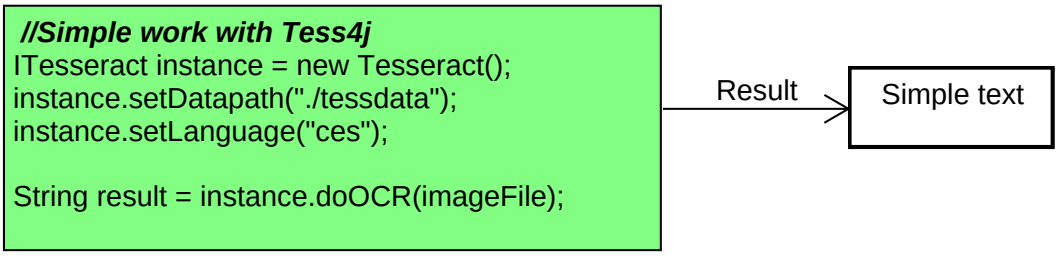
```
public void convertPdfToPng(File pdfFile, int resolution)
{
 pdfDocument document = new pdfDocument();
 document.load(pdfFile);
 SimpleRenderer renderer = new SimpleRenderer();

 /** set resolution (in DPI) */
 renderer.setResolution(resolution);
 List<Image> images = renderer.render(document);

 for (int i = 0; i < images.size(); i++) {
     ImageIO.write((RenderedImage) images.get(i), "png",
     new File("page"+i+".png"));
 }
}
```

Ukázka kódu 7.1: Náznak převodu pdf dokumentu na png

7.2.2 OCR knihovna Tess4j

Nyní bude popsána základní práce s OCR knihovnou *Tess4j* včetně fragmentů zdrojového kódu. Za účelem *OCR* analýzy lze pracovat s knihovnou dvojím způsobem. Za prvé lze využít připravených metod pro jednoduchou *OCR* analýzu a za druhé volat sofistikovanější metody prostřednictvím tzv. **TessBaseAPI**.

Nejprve bude vysvětlena jednodušší varianta. Na obrázku 7.3 je naznačeno, jakým způsobem lze jednoduše pracovat s *Tess4j*. Zelená část na obrázku obsahuje kompletní kód, který je potřeba pro jednoduchou *OCR* analýzu. Jak je z obrázku vidět, tak čtyři řádky kódu postačí k převodu rastrového obrázku do textové podoby.



Obrázek 7.3: Ukázka jednoduché práce s Tess4j

Je třeba pouze:

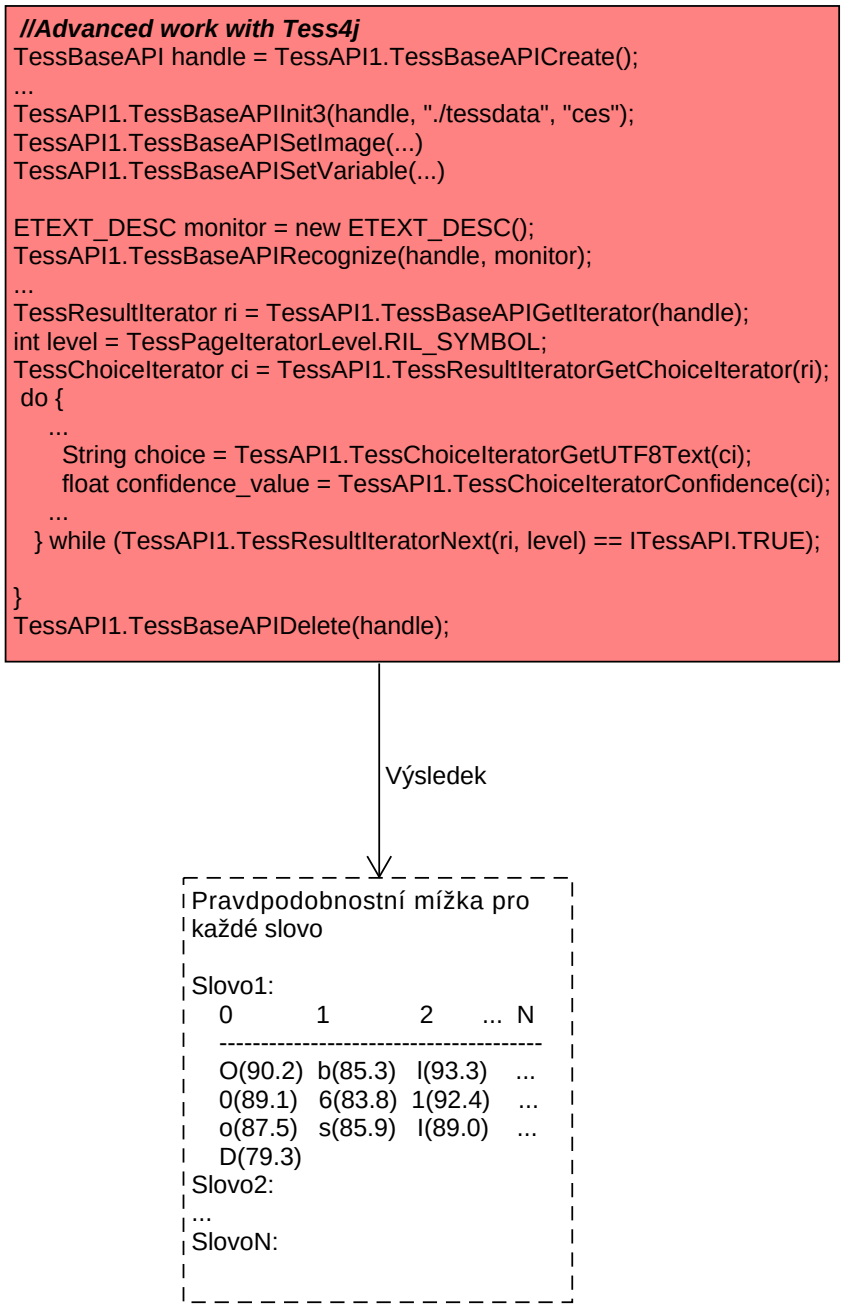
1. vytvořit objekt,
2. nastavit jazyk a cestu k trénovacím datům *Tesseractu*,
3. zavolat metodu na zpracování obrázku, která vrátí výsledek.

Pro jednoduché aplikace a velmi kvalitní vstupní obrázek je tento jednoduchý přístup dostačující. Nicméně pokud není příliš kvalitní vstup nebo je kladen velký důraz na přesnost *OCR* analýzy, tak takto jednoduše problém řešit nelze. Je nutné použít metody, které poskytuje `TessBaseAPI`. Proto bude v dalších odstavcích tato složitější varianta naznačena.

TessBaseAPI

Z důvodu vylepšení OCR převodu je vhodné kromě samotného textu získat ze systému *Tesseract* další informace, které budou dále použity pro opravu chyb. Jedná se o míru jistoty, zda bylo slovo převedeno správně a **pravděpodobnostní mřížku**. Příklad mřížky naznačuje obrázek 7.4 ve své spodní části. Každé zpracované slovo je řetězec znaků. Na každou pozici lze najít více kandidátů, jelikož některé symboly jsou si podobné (viz část 6.1 *Typy chyb*).

Pravděpodobnostní mřížku i míru jistoty slova lze ze systému *Tesseract* získat, ale řešení není úplně přímočaré a jednoduché. Fragment kódu, který vytváří mřížku ukazuje horní červená část na obrázku 7.4.



Obrázek 7.4: Ukázka práce s TessBaseAPI

Kód není zcela kompletní – vybrány jsou pouze důležité části kódu. V proměnných *choice* (symbolizující znak) a *confidence_value* jsou uloženy informace nutné

k vytvoření mřížky. Detailnější pohled na mřížku a jak ji využít pro opravování chyb je uvedeno v části 7.3.1.

Naznačený program prochází všechny varianty symbolů, které poskytne objekt `TessChoiceIterator` (viz cyklus s podmínkou na konci). S takto zachycenými symboly a jejich ohodnocením lze poté dále pracovat.

Je třeba si uvědomit, že `TessChoiceIterator` poskytne pouze znaky, které knihovna uznala za dostatečně podobné. Pokud je tedy cílem matice (nebo chceme-li mřížka), je nutné doplnit všechny ostatní možné znaky s ohodnocením 0.

Za povšimnutí stojí i fakt, že i když je kód psán v Javě, je nutné vytvořit tzv. **handle** na objekt `TessBaseAPI` a po ukončení práce s ním ho korektně odstranit (viz první a poslední řádek červené části). Tato technika velmi připomíná jazyk C nebo C++ a jejich techniky uvolňování použité paměti.

7.2.3 Vliv rozlišení obrázku na chyby

Ještě než bude popsána implementace systému na opravu chyb, tak zde v krátkosti bude popsáno, jaký vliv má rozlišení obrázku na četnost chyb.

Podle oficiální dokumentace *Tesseractu* (viz [1]) se uvádí, že optimální rozlišení obrázku pro zpracování je 300 DPI.

K ověření této skutečnosti byla v rámci projektu naimplementován experiment rozlišení obrázku a jeho vlivu na *OCR* analýzu. Úspěšnost byla měřena pomocí *Word Error Rate* a *Character Error Rate* (*WER* a *CER*).

Experiment a výsledky

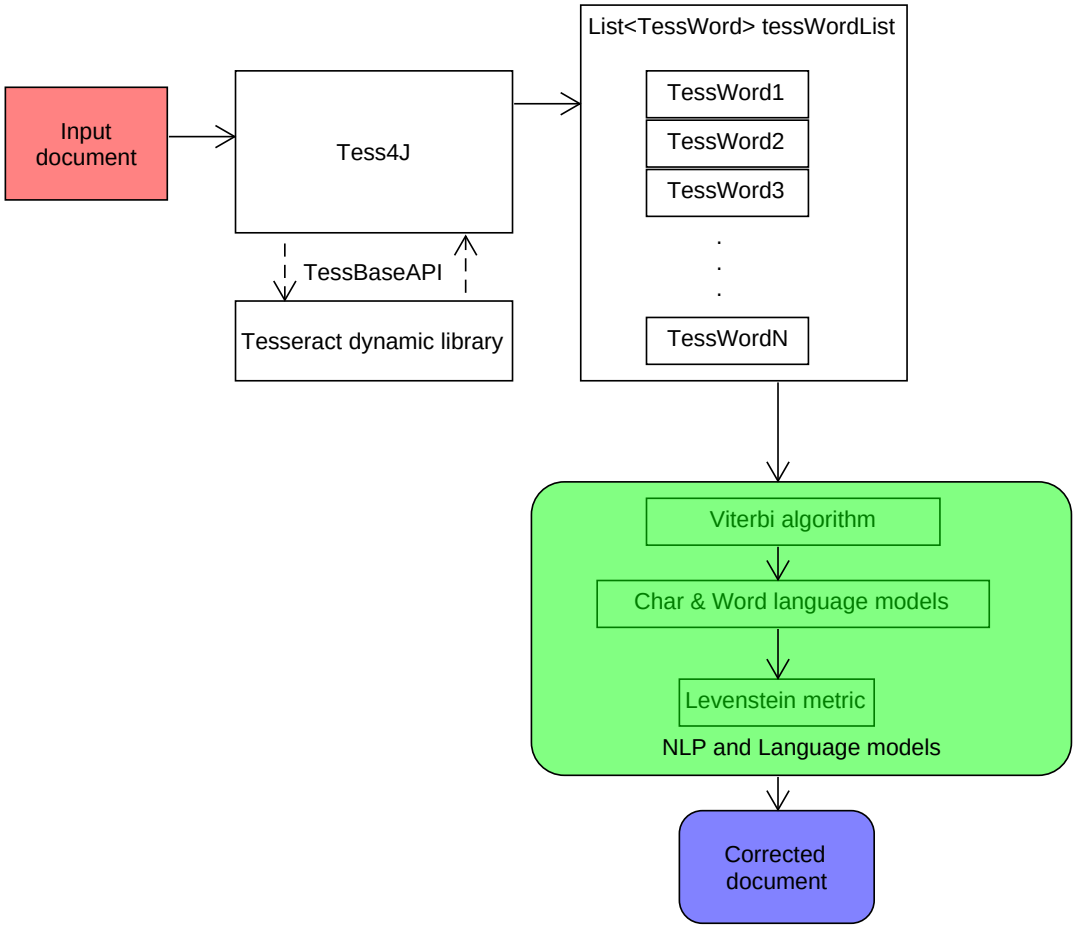
Na dodané množině pdf dokumentů se testovaly 3 typy rozlišení: 100 DPI, 300 DPI a 500 DPI. Výsledky z hlediska úspěšnosti byly srovnatelné. Lišil se ale čas zpracování a tím pádem se potvrdilo to, co bylo uvedeno v dokumentaci. Vyšší rozlišení než 300 DPI nepřináší zlepšení. Naproti tomu příliš nízké rozlišení (100–150 DPI) prodlužuje čas zpracování. Tabulka naměřených údajů s různým rozlišením je uvedena v kapitole *Testování*.

7.3 Implementace systému detekce a opravy chyb

Tato část textu popisuje implementaci nejsložitější části systému, a sice detekce a opravování chyb. V kapitole 6 *Detekce a oprava chyb* byla provedena studie možných přístupů a metod k řešení této problematiky.

Implementaci systému detekce a opravy chyb tvoří kombinace několika metod, které jsou postupně aplikovány. Celkový pohled na postupnou aplikaci technik opravování chyb ukazuje obrázek 7.5.

V první fázi je vstupní dokument zpracován nástrojem *Tesseract* (přesněji knihovnou *Tess4J*). Protože ve své základní podobě umožňuje knihovna pouze jedno-



Obrázek 7.5: Znázornění práce systému detekce a opravy chyb

duché OCR zpracování (výstup ve formě textu), je nutné pomocí **TessBaseAPI** komunikovat přímo se samotnou dynamickou knihovnou[1].

Prostřednictvím *TessBaseAPI* je možné pomocí nástroje *Tesseract* získat nejen pouhý textový výstup, ale i výstup po jednotlivých znacích s pravděpodobnostmi – pravděpodobnostní mřížku. V průběhu zpracovávání knihovnou *Tess4j* jsou postupně tvořeny objekty `TessWord`, které odpovídají jednotlivým slovům. Objekt *TessWord* je přepravka, která uchovává následující atributy (viz ukázka kódu 7.2).

[1]U platformy Windows se jedná o .dll soubor (*dynamic library*) a u platformy Linux o soubor .so (*shared object*).

```
class TessWord {
 private String offeredWord;
 private double wordConfidence;
 private Map<Integer,Map<String,Double>> confidenceMatrix;

 /** constructor, getters and setters */
 ...
}
```

Ukázka kódu 7.2: Přepravka TessWord

Popis atributů:

- *offeredWord* – slovo, které zvolil *Tesseract* jako nejlepší variantu,
- *wordConfidence* – míra jistoty (pravděpodobnost) slova, které vybral *Tesseract* jako nejlepší variantu,
- *confidenceMatrix* – pravděpodobnostní mřížka jednotlivých písmen ve slově.

Seznam objektů *TessWord* tvoří reprezentaci dokumentu, se kterou budeme dále pracovat.

7.3.1 Pravděpodobnostní mřížka Tesseractu

Tato část se zaměří na to, jak lze využít pravděpodobnostní mřížka pro detekci a opravu chyb.

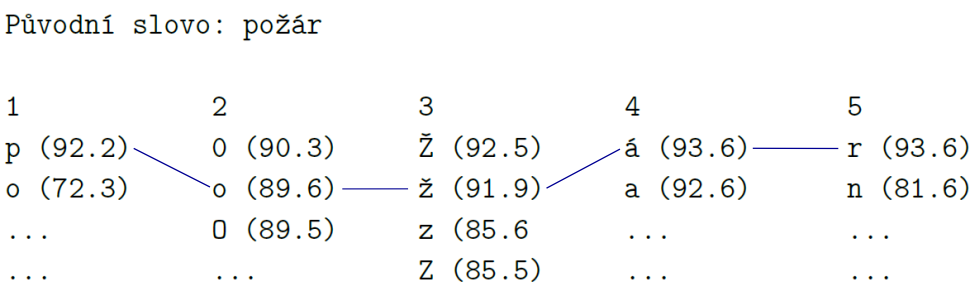
V ideálním případě by symboly v mřížce s největší pravděpodobností měly tvořit správně určené slovo. Nicméně v reálné situaci často nastává situace, kdy symboly, které jsou vyhodnoceny jako nejpravděpodobnější tvoří chybné slovo. Pro názornost mějme následující příklad (viz obrázek 7.6).

```
Původní slovo: požár

1           2           3           4           5
p (92.2)    0 (90.3)    Ž (92.5)    á (93.6)    r (93.6)
o (72.3)    o (89.6)    ž (91.9)    a (92.6)    n (81.6)
...         O (89.5)    z (85.6     ...         ...
...         ...         Z (85.5)    ...         ...

Rozpoznané slovo: p0Žár
```

Obrázek 7.6: Příklad pravděpodobnostní mřížky pro slovo *požár*

Z příkladu je vidět, že kdyby se braly v úvahu pouze nejpravděpodobnější hodnoty z mřížky, nelze zaručit správný výsledek. Díky špatné kvalitě vstupního dokumentu může dojít k situaci, kdy *Tesseract* vyhodnotí jako nejpravděpodobnější špatný (byť vizuálně podobný) symbol.

Jak je vidět z obrázku 7.6, tak správné slovo **požár** v mřížce je, ale správné symboly nemusí být nutně nejpravděpodobnější. Ve většině případů v mřížce správné slovo je, ale je potřeba použít vhodný přístup k jeho nalezení (viz obrázek 7.7).



Obrázek 7.7: Správně nalezená posloupnost symbolů v mřížce

7.3.2 Algoritmus hrubé síly

Nejjednodušší varianta hledání správného slova spočívá v použití hrubé síly (*brute force*). Algoritmus hrubé síly postupně vyzkouší všechny možné varianty symbolů a tak vzniklá slova porovnává se slovníkem. Pokud je slovo nalezeno ve slovníku, tak s velkou pravděpodobností je slovo správně určeno. Algoritmus je sice jednoduchý na implementaci, ale jeho časová složitost není dobrá.

Označíme-li délku slova jako N a počet možných znaků (symbolů) jako k, tak složitost algoritmu hrubé síly je N^k. Tento jednoduchý algoritmus je možné vylepšit tím, že se budou brát v úvahu pouze symboly, jejichž jistota je vyšší než zvolená prahová hodnota.

V další části bude popsán algoritmus nalezení nejpravděpodobnější sekvence symbolů, který využívá pravděpodobnostní mřížku, statistické jazykové modely a dynamické programování k jejímu nalezení.

7.3.3 Viterbiho algoritmus

Viterbiho algoritmus je možné definovat jako optimální řešení problému odhadu stavu sekvence **Markovského procesu** s konečným počtem stavů [9].

Markovský proces má množinu stavů a přechodové pravděpodobnosti. Markovské modely sice nejsou přímo součástí této práce, ale lze s nimi najít určité podobnosti a analogie. Jednotlivé symboly v pravděpodobnostní mřížce můžeme chápat jako stavy Markovského modelu. Přechodové pravděpodobnosti mezi stavy určuje jazykový model v kombinaci s mírou jistoty z nástroje *Tesseract*.

Výsledkem algoritmu jsou dvě matice. Matice Δ, ve které jsou ohodnocené vrcholy odpovídající mřížce a matice Ψ, ve které jsou zpětné odkazy, aby bylo možné zkonstruovat **Viterbiho cestu** (nejpravděpodobnější sekvenci).

Obě matice mají stejné rozměry. Počet sloupců je roven délce sekvence slova. Počet řádků je vždy stejný a každý řádek reprezentuje jeden symbol. V této práci algoritmus pracuje s mřížkou (maticí) s rozměry $N \times k$, kde N je délka slova a k počet všech možných znaků.

Hodnoty δ v matici Δ určují ohodnocení daného písmena na dané pozici v konkrétní sekvenci.

$$\Delta_{N,k} = \begin{pmatrix} \delta_{1,1} & \delta_{1,2} & \cdots & \delta_{1,N} \\ \delta_{2,1} & \delta_{2,2} & \cdots & \delta_{2,N} \\ \vdots & \vdots & \ddots & \vdots \\ \delta_{k,1} & \delta_{k,2} & \cdots & \delta_{k,N} \end{pmatrix} \tag{7.1}$$

Pomocná matice Ψ uchovává pouze zpětné odkazy (odkaz nejlépe ohodnoceného symbolu ze sloupečku $i+1$ na nejlépe ohodnocený symbol ze sloupečku i).

$$\Psi_{N,k} = \begin{pmatrix} \psi_{1,1} & \psi_{1,2} & \cdots & \psi_{1,N} \\ \psi_{2,1} & \psi_{2,2} & \cdots & \psi_{2,N} \\ \vdots & \vdots & \ddots & \vdots \\ \psi_{k,1} & \psi_{k,2} & \cdots & \psi_{k,N} \end{pmatrix} \tag{7.2}$$

Pro výpočet je zapotřebí nejprve definovat následující proměnné:

- $\delta_k(i)$ – ohodnocení k–tého vrchol na pozici i,
- T_c – `Tesseract confidence` (pravděpodobnost symbolu, kterou vrátil *Tesseract*) ,
- a_{ji} – pravděpodobnost přechodu ze stavu (symbolu) j do stavu i dle jazykového modelu (pravděpodobnost bigramu) ,
- π_k – pravděpodobnost, že slovo začíná k–tým stavem (symbolem),
- w_t – váha *Tesseractu* (reálné číslo od 0 do 1).

V první fázi algoritmus ohodnotí první sloupec vrcholů (počáteční symboly slova) hodnotou podle následujícího vzorce:

$$\delta_k(1) = w_t T_c + (1 - w_t)\pi_k \tag{7.3}$$

Jedná se o lineární kombinaci pravděpodobností ze systému *Tesseract* a natrénovaného znakového jazykového modelu (viz část 6.4 *Statistické jazykové modely*). Lineární kombinace vhodně kombinuje obě zmíněné pravděpodobnosti. Pokud jazykový model ohodnotí dané písmeno nízkou (či dokonce nulovou) pravděpodobností, tak se celková hodnota sníží a naopak. Stejná situace nastává i u pravděpodobností ze systému *Tesseract*. Lineární kombinace by měla eliminovat situace, kdy se například

objeví velké písmeno uprostřed slova ohodnocené velkou pravděpodobností ze systému *Tesseract*. V takovém případě jazykový model ohodnotí tento výskyt nízkou pravděpodobností a celková hodnota symbolu se sníží.

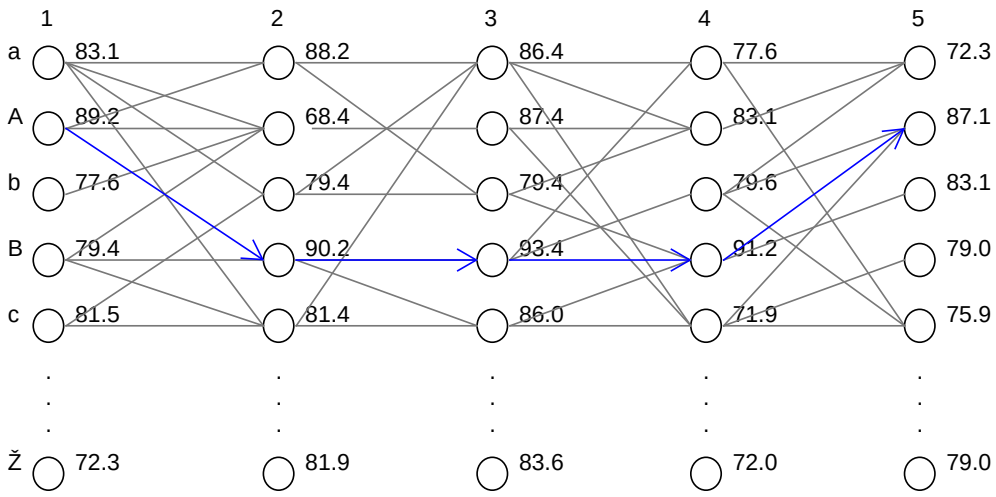
Tímto způsobem se ohodnotí všechny vrcholy v prvním sloupci a určí se maximum. V další fázi se pro každý další sloupec t (kde $N \geq t \geq 2$) ohodnotí všechny vrcholy podobným způsobem. Souhrnně to lze popsat následujícím vzorcem:

$$\delta_k(i) = \max_{1 \leq j \leq N} [\delta_k(j-1) + (w_t T_c + (1 - w_t) a_{ji})] \tag{7.4}$$

Aplikováním tohoto výpočtu vznikne matice ohodnocení. Aby v ní bylo možné najít nejlepší cestu, tak je potřeba si v každém stavu zapamatovat předchozí stav, který byl maximalizován. Tímto vznikne další matice Ψ jejíž prvky tvoří indexy, které byly pro daný sloupec maximálními. Výpočet této matice se děje současně s maticí Δ a počítá se následujícím způsobem:

$$\psi_k(i) = \arg\max_{1 \leq j \leq n} [\delta_k(j-1) + (w_t T_c + (1 - w_t) a_{ji})] \tag{7.5}$$

Tvorba Viterbiho cesty začíná v posledním sloupci matice Ψ. Z matice Δ se určí maximální hodnota a odpovídající index je počátek zpětné tvorby. Matice se prochází po zpětných ukazatelích až k prvnímu sloupci a tím se určí nejpravděpodobnější posloupnost písmen. Zjednodušené znázornění Viterbiho algoritmu ukazuje obrázek 7.8. Z důvodu přehlednosti nejsou v obrázku zaneseny všechny hrany. Modré šipky v obrázku ukazují nalezenou nejpravděpodobnější cestu.



Obrázek 7.8: Znázornění Viterbiho algoritmu

Složitost algoritmu je N^2k a je tedy závislá na počtu možných symbolů a délce sekvence. Váha *Tesseractu* w_t byla určena experimentálním způsobem a její doporučená hodnota je 0.7. Uvedená lineární kombinace spojuje pravděpodobnost bigramu a **Tesseract confidence** s danou vahou a na základě hodnot získaných tímto výpočtem se navrhuje oprava slova.

Pro názornost se vraťme k příkladu mřížky na obrázku 7.6 na straně 37. Pokud by byla nastavena váha w_t na hodnotu 1 (nebral by se v úvahu jazykový model), bylo by vráceno slovo `p0Žár`. Pokud se ale aplikují jazykové modely, je pravděpodobnost bigramů `p0 a 0Ž` blízká nule a tím pádem se sníží ohodnocení vrcholu a pravděpodobnější bigramy (např. `po` nebo `ož`) dostanou lepší ohodnocení a mají větší pravděpodobnost být vybrány.

Kdyby byla nastavena váha w_t na příliš nízkou hodnotu, tak by rozpoznaná slova s největší pravděpodobností neodpovídala nebo nedávala smysl.

Pravděpodobnosti bigramů (a_{ij}) určí jazykový model podle *MLE* metody na základě trénovacích dat (viz část *6.4.2 Metoda maximální věrohodnosti* na str. 26). Detailněji se jazykovým modelům bude věnovat následující část.

7.3.4 Jazykové modely

Bylo již řečeno, že z důvodu složitosti přirozeného jazyka je nutné použít pro stanovení odhadu pravděpodobností metodu maximální věrohodnosti (*MLE*). Podstatou této metody je, že trénovací data modelují celý přirozený jazyk a z nich vypočítané pravděpodobnosti.

Trénování jazykového modelu

Pro tuto práci byl zvolen *trigramový* znakový jazykový model (písmeno a dvě jeho předchozí). Trénování probíhá extrakcí *N–gramů* z textového souboru do vhodných datový struktur (`HashMap`). Kromě *N–gramů* samotných je ještě třeba uložit i jejich četnost a vytvořit si slovník všech písmen, které se objevují v trénovacích datech.

```
HashMap<Long, Integer> trigrams;
HashMap<Long, Integer> bigrams;
HashMap<Character, Integer> unigrams;
```

Ukázka kódu 7.3: Ukázka datových struktur pro uchování *n–gramů*

Přestože byl zvolen trigramový jazykový model, tak v předchozí ukázce kódu jsou připravené struktury i pro uchování bigramů a unigramů. Důvodem je vyhlazování (viz část *Vyhlazování* na str. 27).

Pro reprezentaci unigramu byl zvolen objekt `Character`. Pro vhodnou reprezentaci bigramů a trigramů byl vybrán datový typ `Long`, do kterého se funkcí namapuje bigram a trigram. Mapovací funkci ukazuje následující vzorec:

$$bigramIndex = a(vocabularySize) + b \tag{7.6}$$

kde a je pozice znaku ve slovníku a b je pozice historie ve slovníku.
Zpětný chod mapovací funkce lze vyjádřit následujícími úpravami:

$$a = \frac{bigramIndex}{vocabularySize} \tag{7.7}$$

$$b = bigramIndex \mod (vocabularySize) \tag{7.8}$$

Výsledné znaky bigramu poté lze získat jednoduše ze slovníku. U trigramu je mapovací funkce složitější:

$$trigramIndex = a(vocabularySize^2) + b(vocabularySize) + c \tag{7.9}$$

A vyjádřením lze opět získat indexy do slovníku a,b a c.

$$a = \frac{trigramIndex}{vocabularySize^2} \tag{7.10}$$

$$b = \frac{trigramIndex}{vocabularySize} \tag{7.11}$$

$$c = trigramIndex \mod (vocabularySize) \tag{7.12}$$

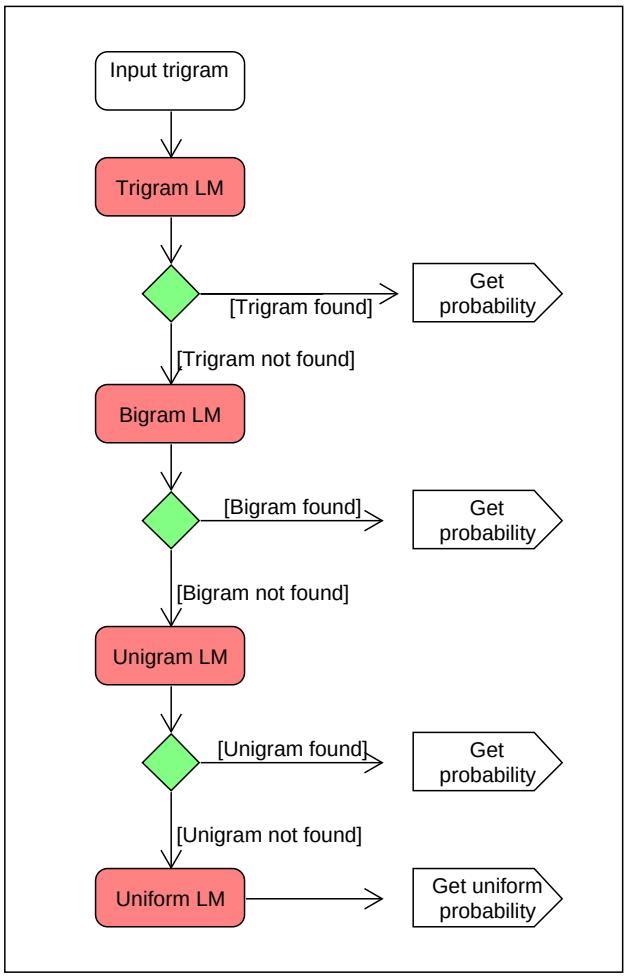
Tato reprezentace byla zvolena z důvodu rychlosti. Nejdříve byla zvolena reprezentace *n–gramů* pomocí objektů, ale s narůstající velikostí trénovacích dat se snižovala rychlost trénování.

Vyhlazování jazykového modelu

V teoretické části bylo popsáno, jakým způsobem se lze vyhnout nekonečné entropii pomocí **lineární interpolace**. Zjednodušený princip lineární interpolace jazykových modelů ukazuje obrázek 7.9.

Obrázek ukazuje konkrétně čtyři jazykové modely (trigramový, bigramový, unigramový a uniformní). Uniformní jazykový model je takový model, který každému unigramu přiřadí stejnou pravděpodobnost ($|W|^{-1}$, kde $|W|$ je velikost slovníku). Pokud je pravděpodobnost trigramu nulová (nepozorovaný trigram), trigram je nahrazen bigramovým modelem a určí se jeho pravděpodobnost. Pokud nastane situace nepozorovaného bigramu, tak je bigram nahrazen unigramem a aplikuje se unigramový jazykový model. Pokud je i unigram neznámý, konečnou pravděpodobnost určí uniformní jazykový model, který již vrátí nenulovou pravděpodobnost.

Jednolivé koeficienty λ jsou určené pomocí **E-M algoritmu** (viz část 6.4.3).



Obrázek 7.9: Ukázka postupné aplikace jednotlivých jazykových modelů

Výpočet pravděpodobnosti trigramu pomocí lineární interpolace jazykových modelu je na následující ukázce kódu 7.4.

```
public double getProbability(char c, char h1, char h2) {

 double totalProbability = (lam1 * getUniformProbability())
  + (lam2 * getUnigramProbability(c))
  + (lam3 * getBigramProbability(c, h1))
  + (lam4 * getTrigramProbability(c,h1,h2));
 return totalProbability;

}
```

Ukázka kódu 7.4: Lineární interpolace jazykových modelů

Takováto kombinace písmenkových jazykových modelů umožní jednoduše přistupovat k pravděpodobnostem a je zaručeno, že v případě **nenalezení** *N–gramu* je vrácena **nenulová** pravděpodobnost a tím pádem se zabrání problému nekonečné entropie jazykového modelu.

Na závěr této části ještě uveďme, že program má možnost zapnutí korekce pomocí *Levensteinovy* metriky. Při zapnutí této korekce je ověřeno, zda slovo vytvořené z kombinace mřížky *Tesseractu* a jazykových modelů je ve slovníku. Pokud slovo ve slovníku není, tak se při zapnutí korekce najde nejbližší slovo a to je považováno za výsledek. Výhodou je, že všechna výsledná slova budou dávat smysl. Nevýhodou je, že pokud se v textu objeví například nějaká cizí jména či slova v jiném než českém jazyce, tak tato slova korekce znehodnotí.

Nicméně tato závěrečná korekce má velký vliv na úspěšnost (viz kapitola 8 *Testování*).

7.4 Implementace vyhledávacího systému

Tato část se zaměří na komunikaci s již běžící instancí *Apache Solr* a to prostřednictvím klientské aplikace s využitím *SolrJ* (pro detailní popis knihovny a dokumentaci viz zdroj [21]).

7.4.1 Popis klientské aplikace a SolrJ

SolrJ je Java klient, prostřednictvím kterého lze přistupovat k instanci *Apache Solr*. Knihovna nabízí rozhraní k přidávání (případně úpravě) dokumentů a tvorbu dotazů [3].

Knihovna neumožňuje přímé vytvoření **Core**. Core je v terminologii *Apache Solr* běžící instance *Lucene* indexu společně s veškerou *Solr* konfigurací [3]. Běží-li jedna nebo více těchto instancí, tak je možné komunikovat. V následujícím textu budou popsány všechny důležité požadavky na knihovnu včetně ukázek zdrojového kódu.

Práce s dokumenty

Nejčastější akce je přidání nového dokumentu. V prostředí *SolrJ* je potřeba nejprve vytvořit objekt `SolrInputDocument`. Dále se tomuto objektu přiřadí atributy (*fields*), podle kterých je možné posléze vyhledávat. Jako poslední krok je nutné instanci **SolrClient** přidat objekt dokumentu a následně metodou *commit()* potvrdit přidání. Ukázku jednoduchého přidání dokumentu demonstruje ukázka kódu 7.5.

```
private SolrClient solrClient;

public void addDocument(String documentText)
{
 SolrInputDocument doc = new SolrInputDocument();
 doc.addField("id", documentCounter);
 doc.addField("datetime", DatetimeUtils.printNow());
 doc.addField("text", documentText);

 solrClient.add(doc);
 solrClient.commit();
 documentCounter++;
}
```

Ukázka kódu 7.5: Metoda přidávající dokument

Dotazy

Podobně jednoduchým způsobem lze pracovat s *Apache Solr* dotazy. Je třeba vytvořit objekt `SolrQuery` a zavolat jeho metodu *query()*. Výsledky budou následně uloženy do objektu `SolrDocumentList`.

Metodu, která vrací seznam dokumentů odpovídající fulltextovému dotazu, je naznačena v ukázce kódu 7.6.

```
public SolrDocumentList executeQuery(String queryText)
{
 SolrQuery query = new SolrQuery();
 query.setQuery(queryText);

 QueryResponse qresponse = solrClient.query(query);
 SolrDocumentList list = qresponse.getResults();

 return list;
}
```

Ukázka kódu 7.6: Metoda volání fulltextových dotazů

Apache Solr umožňuje i pokročilé hledání, kdy je možné filtrovat výstup či dotazovat se na konkrétní atributy dokumentu. Pro názornost uveďme příklad některých pokročilých dotazů.

Boolean dotazy:

```
+solr + apache elastic -system
solr AND apache OR elastic NOT system
```

Takto sestavené dotazy vyberou všechny dokumenty, které obsahují slova `solr` a zároveň `apache` nebo slovo `elastic`, přičemž nesmí obsahovat slovo `system`.

Dotazy podle jednotlivých polí:

```
text:solr
age:18
```

V tomto případě se vyberou všechny dokumenty, které mají v atributu `age` hodnotu **18** a v atributu `text` slovo `solr`.

Dotazy na rozsah:

```
age:[18 TO 22]
age:[18 TO 22}
age:[18 TO *]
```

V první případě dotaz vybere dokumenty, jejich hodnota v atributu `age` je mezi **18** a **22** včetně obou krajních hodnot. Ve druhém případě se vyberou ty dokumenty, které mají hodnotu `age` větší než **18** a zároveň ostře menší než **22**. Třetí příklad vybere dokumenty, které mají hodnotu `age` větší než **18**.

Fragment metody, která vrátí pouze *id* dokumentu a které mají pole "*year*" mezi 1993 a 1994 je vidět na obrázku ukázky kódu 7.7.

```
...
query.setFields("id"); /** return only document id */
query.set("q", "year:["+1993+"TO"+1994+"]");
QueryResponse qresponse = solrClient.query(query);
SolrDocumentList list = qresponse.getResults();
...
```

Ukázka kódu 7.7: Fragment volání pokročilejších dotazů

Tímto způsobem lze tedy konstruovat složitější dotazy. Do parametru *q* (tzv. **Query String**) se uloží text dotazu (všechny výše uvedené varianty) a takto postavený dotaz se odešle. Výsledky dotazů lze také řadit.

7.5 Klasifikace dokumentů

Za účelem zlepšení efektivity a přesnosti vyhledávání byla implementována funkcionalita klasifikace dokumentů. Tato klasifikace vznikla z důvodu potřeby vyhledávat dle tématu. Po rozpoznání textu dokumentu je těsně před indexací nasazen algoritmus klasifikace, který určí nejpravděpodobnější zařazení dokumentu. Při následném vyhledávání dokumentu bude možnost zobrazení podobných dokumentů (dokumentů ve stejné kategorii). V následujícím textu bude popsána klasifikace dokumentů pomocí knihovny *Brainy*. Každý realizovatelný klasifikátor modeluje následující funkci:

$$h : \mathbf{x} \rightarrow y \tag{7.13}$$

a snaží se najít optimální h. Vektor $\mathbf{x}$ je vektor příznaků (tzv. **features**) a skalár y je výsledná třída, do které objekt, popsaný vektorem $\mathbf{x}$, je zařazen.

Strojové učení používá k určení klasifikační třídy metody statistiky a pravděpodobnosti na základě pozorování trénovacích dat. Je tedy nutné mít k dispozici dostatek (označkovaných) dokumentů.

Klasifikátor nad těmito dokumenty provede tzv. **extrakci příznaků** (*feature extraction*). Příznak je charakteristika objektu, podle které lze ode sebe odlišit jednotlivé objekty. Pro každý trénovací i testovací objekt musí tato extrakce proběhnout. Příznaky je třeba vhodně reprezentovat a uložit.

Nejčastější reprezentace příznaků je vektor. Jednotlivé složky vektoru symbolizují daný příznak. Reprezentace postavená na vektorech má výhodu v tom, že vektory se dají snadno porovnávat na základě jejich (eukleidovské či kosinové) vzdálenosti.

Klasifikátor založený na strojovém učení vykazuje obecně dobrou přesnost. Příklady standardních klasifikačních metod, založených na strojovém učení, jsou například naivní Bayesovský klasifikátor, k-nejbližších sousedů, logistická regrese, *Support Vector Machines*, *Maximum Entropy* klasifikátor a *neuronové sítě*.

V další části bude popsána knihovna *Brainy*, která je použita pro klasifikaci dokumentů v této práci.

7.5.1 Knihovna Brainy

Brainy je *machine learning* knihovna napsaná v Javě. Definuje základní rozhraní pro běžné typy úloh strojového učení a obsahuje základní implementace populárních algoritmů [14].

Pro použití knihovny je potřeba realizovat následující kroky:

1. příprava trénovacích dat,
2. příprava příznaků (*features*),
3. získání příznakových vektorů (*feature vectors*),
4. trénování klasifikátoru.

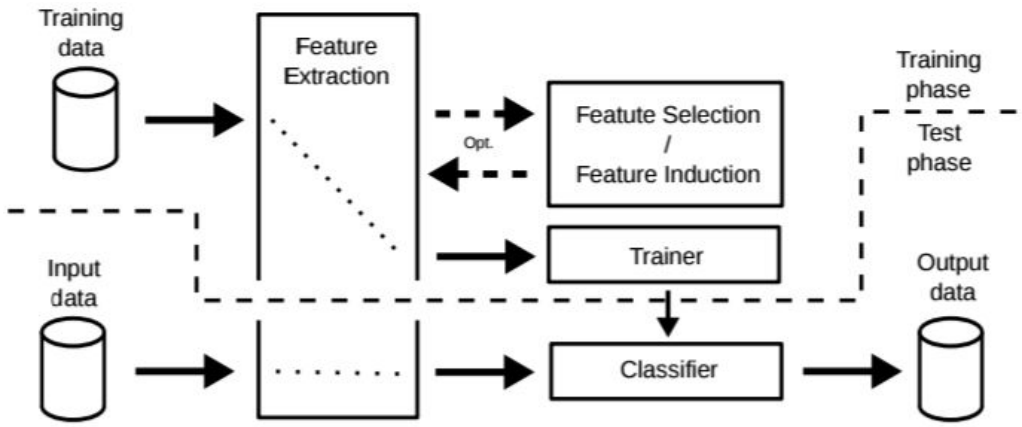
Obrázek 7.10 na straně 48 ukazuje souhrnně práci klasifikátoru.

Příprava trénovacích dat

```
BasicTrainingInstanceList<Sentence> trainingSet
  = new BasicTrainingInstanceList<Sentence>(trainData, null, labels,numClasses);
```

Ukázka kódu 7.8: Fáze přípravy dat

Předchozí ukázka kódu zobrazuje přípravu trénovacích dat. `Sentence` je objekt, který bude klasifikován (viz ukázka kódu 7.9). Do seznamu trénovacích dat – `BasicTrainingInstanceList` vstoupí seznam objektů `Sentence`, seznam označkovaných `Sentence` ve formě celých čísel (0 až `numClasses-1`) a počet tříd.



Obrázek 7.10: Schéma úlohy klasifikace v Brainy [14]

Objekt `Sentence` obsahuje text věty, seznam tokenů (slov) a především typ, který určuje do které klasifikační třídy objekt věty patří.

```
public class Sentence {
    private String type;
    private String text;
    private List<String> tokens;
    /** constructor, getters and setters */
}
```

Ukázka kódu 7.9: Objekt `Sentence`

Příprava příznaků

Každý tzv. *Feature template* musí implementovat rozhraní knihovny `Feature`. Pro klasifikaci v tomto projektu byly jako příznaky zvoleny unigramy a bigramy. Za tímto účelem byla vytvořena obecná třída `Ngram`. Rozhraní vynutí implementaci následujících tří metod (viz ukázka kódu 7.10).

```
class Ngram implements Feature<Sentence> {
 public void extractFeature(ListIterator<Sentence> it,
      FeatureVectorGenerator generator) {
     /** extract ngrams from list of sentences and set features to the generator */
 }
 public int getNumberOfFeatures() {
    /** return number of features */
 }
 public void train(InstanceList<Sentence> instances) {
    /** use threshold and store count of Ngrams */
 }
}
```

Ukázka kódu 7.10: Fáze přípravy příznaků

Pro natrénování klasifikátoru je nastaven práh (*threshold*), kdy jsou v trénovací fázi použity pouze *N–gramy* s četností větší než práh.

Knihovna *Brainy* odlišuje termíny *Feature, Feature template* a *Feature set*. *Feature* je jeden konkrétní příznak (např. jeden *N–gram*). *Feature template* je definice *Features* – jeden *feature template* je obvykle zodpovědný za více *features* (např. *feature template* jsou unigramy a *features* jednotlivá slova) [14]. *Feature set* je skupina *feature templates*.

Pro klasifikaci je třeba vytvořit *feature set* a přidat do něj jednotlivé *feature templates* (ukázka kódu 7.11). Jako jedna *feature template* jsou zvoleny unigramy a jako druhá bigramy. Hodnota 5 u obou prahů byla zvolena na základě několika experimentálních konfigurací s různými prahy.

```
FeatureSet<Sentence> featureSet = new FeatureSet<Sentence>();
featureSet.add(new Ngram(5,1));
featureSet.add(new Ngram(5,2));
```

Ukázka kódu 7.11: Vytvoření sady příznaků (*Feature set*)

Trénování klasifikátoru

Pro tuto práci byl zvolen klasifikátor maximální entropie, protože dosahoval lepších výsledků než klasifikátory *Naive Bayes* a *Support vector machines*, které knihovna nabízí. Trénování klasifikátoru zobrazuje ukázka kódu 7.12.

```
/** Feature set filled with feature templates */
set.train(trainDataList);
//get feature vectors
DoubleMatrix data = set.getData(trainDataList);
/** get labels in a vector */
IntVector ls = set.getLabels(trainDataList);
/** create and train classifier */
SupervisedClassifierTrainer trainer = new MaxEntTrainer();
Classifier classifier = trainer.train(data,ls,numOfLabels);
```

Ukázka kódu 7.12: Trénování klasifikátoru

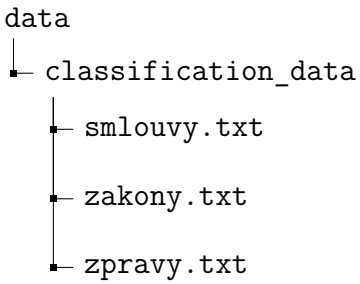
Použití klasifikátoru

Použití natrénovaného klasifikátoru je již jednoduché. Je potřeba vytvořit seznam testovaných (neoznačkovaných) objektů – `Sentence` a nechat klasifikátor určit třídu. Metodu `markCategories`, která vrací seznam označených objektů ukazuje následující ukázka kódu 7.13. Je-li k dispozici pro každou větu třída, výsledný dokument patří do třídy, která převládá mezi jednotlivými větami dokumentu.

Jako trénovací data byly použity pro tuto práci tři třídy: zákony, zprávy a smlouvy. Každá třída má svá trénovací data uložena v textovém souboru. Pro přidání třídy není nutné měnit zdrojový kód, je pouze potřeba vytvořit trénovací data a uložit je do nového textového souboru (pojmenovaný jako název třídy). Tento textový soubor stačí uložit do složky `data/classification_data` a program si novou třídu automaticky vytvoří a natrénuje. Strukturu uložení textových souborů ukazuje obrázek 7.10.

```
public List<Sentence> markCategories(List<Sentence> sents){
 BasicInstanceList<Sentence> instances = new
        BasicInstanceList<Sentence>(sents, null);
 DoubleMatrix data = set.getData(instances);
 BasicClassificationResults results =
  BasicClassificationResults.create(numberOfLabels,
         data.columns());
 classifier.classify(data,results);
 IntVector labels = results.getLabels();
 int i = 0;
 for(Sentence s : sents) {
  s.setType(inverseClassesMap.get(labels.get(i)));
  i++;
 }
 return sents;
}
```

Ukázka kódu 7.13: Použití klasifikátoru



Obrázek 7.11: Trénovací data v adresářové struktuře

8 Testování

Tato práce byla testována především průběžnými jednotkovými (*JUnit*) testy. Dále byly provedeny experimenty s cílem nalezení optimální konfigurace celého systému. V rámci těchto testů byly hledány optimální váhy *Tesseractu* a jazykových modelů.

8.1 Vliv rozlišení dokumentů na výsledky OCR analýzy

Z důvodu ověření skutečnosti, že optimální rozlišení pro OCR analýzu *Tesseractem* je 300 DPI, byl vytvořen další test, který měřil při nastavených různých rozlišení dobu zpracování a kvalitu výstupu. Výsledek experimentu víceméně potvrdil skutečnost, že velké rozlišení nepřináší zlepšení. Při malém rozlišení (100 DPI) výrazněji roste čas zpracování. Pro kolekci několika dokumentů ukazuje detailnější výsledky tabulka 8.1.

	100 DPI	300 DPI	500 DPI
WER (průměr)	0,35	0,35	0,35
CER (průměr)	0,23	0,22	0,22
Čas zpracování (průměr)	33 540 ms	23 689 ms	25 754 ms

Tabulka 8.1: Vliv rozlišení na výsledky OCR

8.2 Jednotkové testy korekce slov

Po implementaci slovního korektoru pomocí jazykových modelů byla napsána sada jednotkových testů, které si kladly za cíl vyzkoušet všechny kombinace přístupů k opravě chyb.

V první řadě bylo testováno pravidlové nahrazení všech kombinací podezřelých znaků (1 → l, O → 0 a další viz 6.1 *Typy chyb* na straně 24). Další testy pokrývaly ostatní obecné chyby a testovaly především základní chyby, se kterými se korektor slov musel nutně vypořádat.

V této části byly jednotkové testy velice užitečné, protože po úpravách implementace korektoru slov to byl snadný a rychlý způsob, jak ověřit správnost úpravy či přidání funkcionality.

8.3 Jednotkové testy Levensteinovy metriky

Vzhledem k tomu, že *Levensteinova* metrika se počítá pro všechna slova ve slovníku, tak je nutná efektivní implementace. Nejprve byl realizován algoritmus *Levensteinovy* metriky pomocí rekurzivního přístupu. Jednotkové testy pomohly detekovat neefektivitu rekurzivního algoritmu. Ačkoliv byl rekurzivní algoritmus jednoduchý na implementaci, ve srovnání s nerekurzivní variantou pracoval značně pomaleji.

8.4 Nastavení váhy Tesseractu a jazykových modelů

V rámci tohoto experimentu byly nalezeny optimální váhy *Tesseractu* a jazykových modelů. Byly vyzkoušeny váhy w_t v intervalu $\langle 0; 1 \rangle$. byl použit jednoduchý přístup (algoritmus) hrubé síly (*Brute force*). V průběhu testování byla vždy vytvořena instance *Tesseractu* pro každou váhu (od 0 do 1 s krokem 0.1). Testovány byly případy s korekcí *Levensteinovou* metrikou i bez této korekce. Pro měření úspěšnosti byly použity metriky *Accuracy*, *Word accuracy*, *Word error rate* a *Character error rate* (viz 6.5 *Word Error Rate a Character Error Rate*).

Ze všech uvedených metrik má *Accuracy* nejmenší vyjadřovací sílu, protože nebere v úvahu chybějící či přidaná slova. Nicméně i na této metrice by mělo být vidět zlepšování systému.

Výsledky experimentu jsou znázorněny na grafech 8.1 a 8.2. Graf 8.1 ukazuje situaci s korekcí výsledků pomocí *Levensteinovy* metriky a graf 8.2 ukazuje situaci bez použití této metriky.

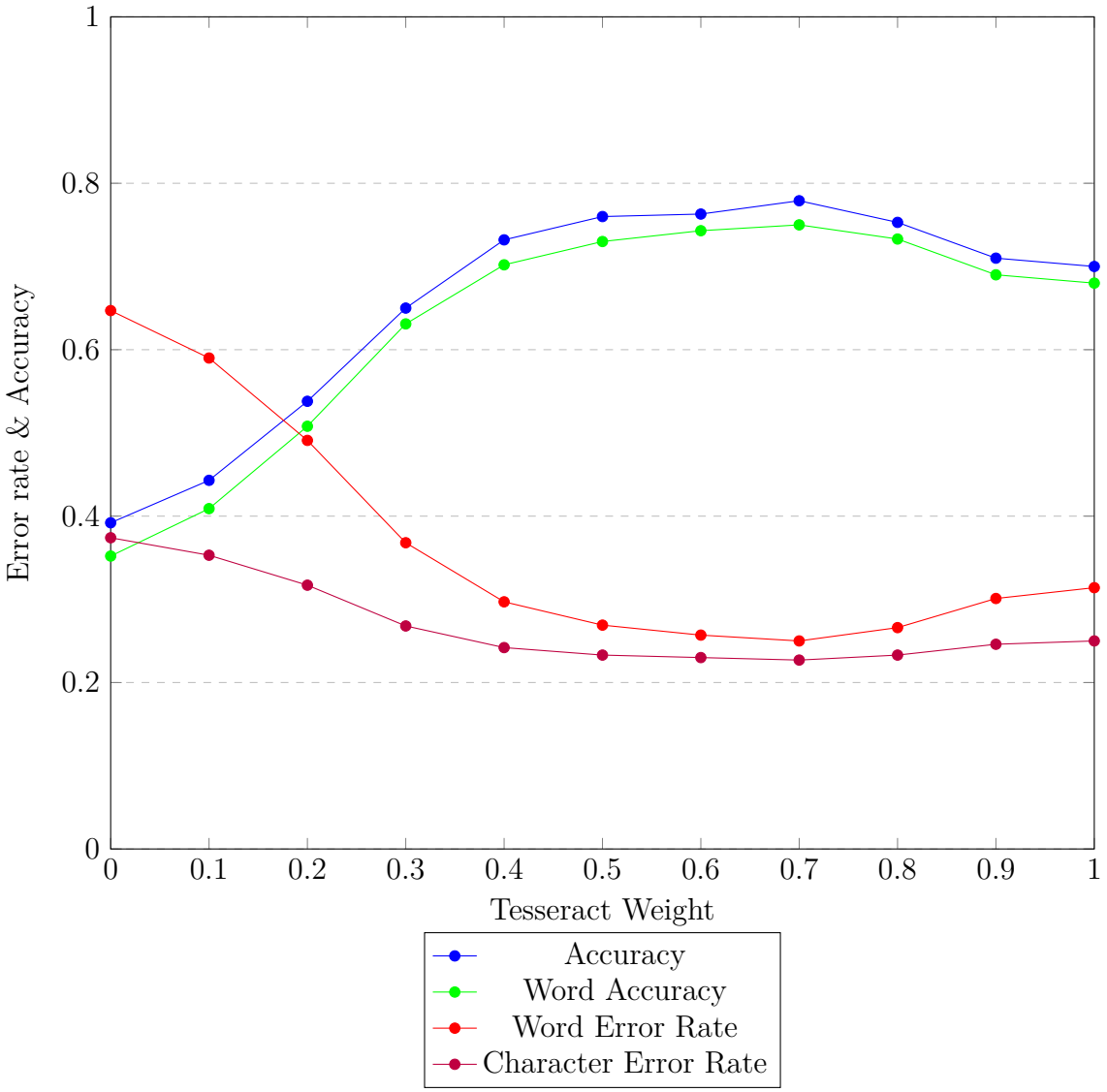
Pro připomenutí uveďme, že váha *Tesseractu* rovná 0 znamená použití pouze výstupu jazykového modelu a váha rovná 1 znamená použití pouze *Tesseractu*. Čím blíže k jedné, tím se více upřednostňuje *Tesseract* a čím blíže je váha k nule, tím se více upřednostňuje jazykový model (detailněji viz 7.3.3 *Viterbiho algoritmus*).

Očekávané chování je takové, že nejhorší výsledky budou u váhy 0. Se vzrůstající vahou by měla úspěšnost stoupat (a zároveň chybovost klesat).

Z obou grafů je vidět, že nejlépe se program chová s váhou 0.7. Pro hodnoty větší než tato hodnoty lze v grafu pozorovat mírné zhoršení. Zhoršení je způsobené především tím, že *Tesseract* zvolí při rozpoznávání chybný symbol a díky tomu, že váha jazykového modelu je malá (či žádná v případě váhy rovné 1), není jak opravit chyby.

Je třeba zdůraznit, že mezi testovanými dokumenty byly dokumenty s horší kvalitou i dokumenty, které obsahovaly cizí jména či různé zkratky, které mohly být *Levensteinovou* metrikou opraveny na zcela jiná (chybně opravená) slova, protože je slovník neobsahoval.

Pro hodnotu váhy 0.7 jsou k dispozici v tabulce 8.4 další údaje, včetně minimální a maximální hodnoty, kde je nejlépe vidět rozdíl mezi zpracovávanými dokumenty.

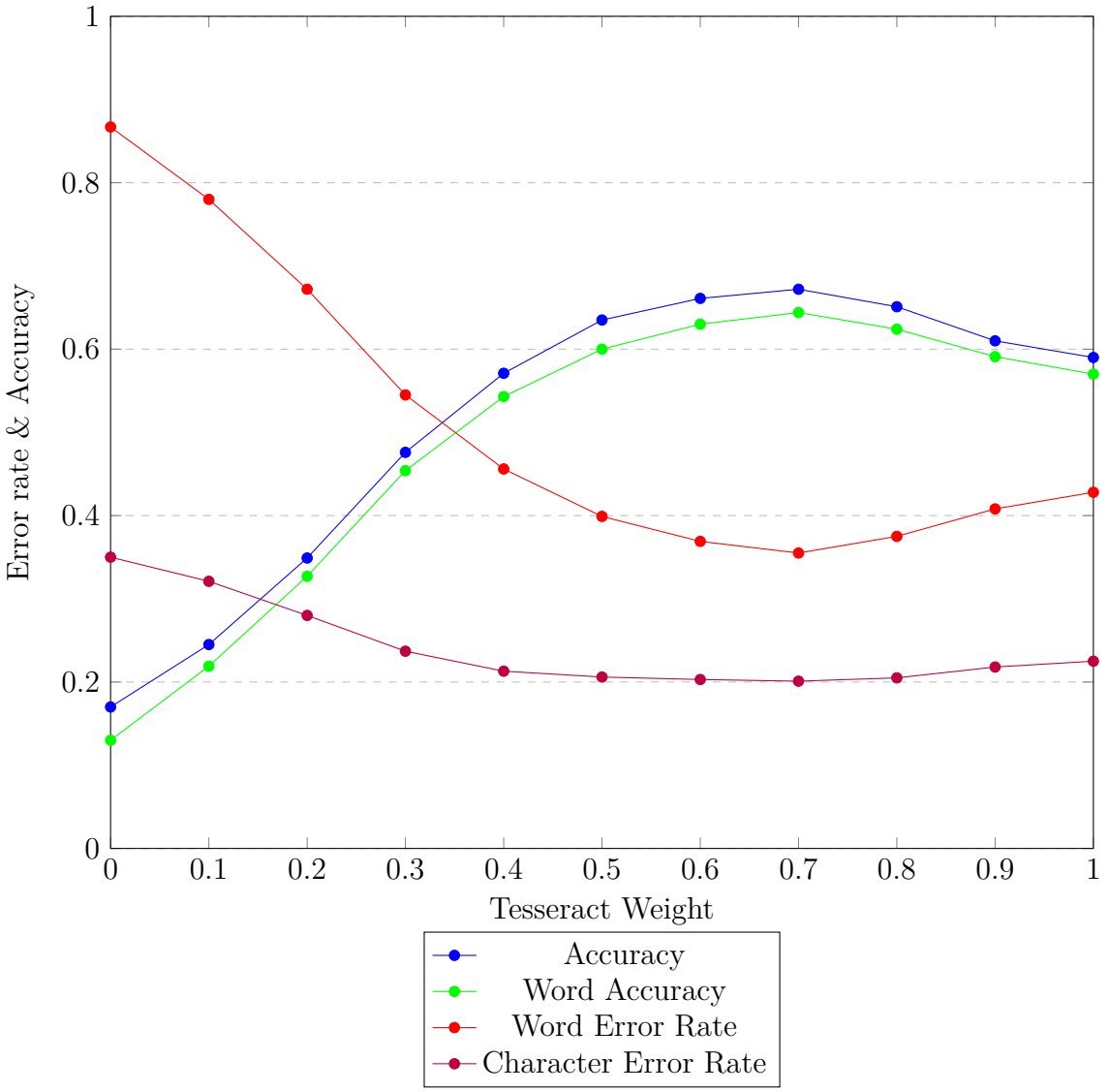


Graf 8.1: Výsledky OCR rozpoznávání v závislosti na váze systému *Tesseract* w_t v intervalu $\langle 0; 1 \rangle$ s korekcí Levensteinovou metrikou

Při testování byl kladen důraz na to, aby vstupní dokumenty byly reprezentativním výběrem různých kvalit a obsahů. Grafy reflektují skutečnost, že samotný *Tesseract* (váha 1) vykazuje horší přesnost než za pomoci jazykových modelů.

Nejlepší přesnosti z hlediska **Word Accuracy** dosahuje systém při nastavení váhy *Tesseractu* 0.7 se zapnutou korekcí *Levensteinovou* metrikou (viz v grafu 8.1 modrá křivka). Korekce *Levensteinovou* metrikou dokázala vylepšit přesnost u všech testovaných vah.

Na samotný závěr této kapitoly uveďme, že aplikace jazykových modelů pomohla vylepšit úspěšnost pravděpodobnostní mřížky *Tesseractu*.



Graf 8.2: Výsledky OCR rozpoznávání v závislosti na váze systému *Tesseract* w_t v intervalu $\langle 0; 1 \rangle$ bez korekce Levensteinovou metrikou

Measure	Min	Max	Avg
Accuracy	0.5	0.86	0.78
WordAccuracy	0.49	0.85	0.75
WER	0.15	0.49	0.25
CER	0.14	0.32	0.22

Tabulka 8.2: Výsledky OCR rozpoznávání u váhy 0.7

9 Závěr

Cílem diplomové práce bylo naprogramovat aplikaci, která umožní efektivně vyhledávat v množině dokumentů v podobě rastrových obrázků. V rámci teoretické části byly vysvětleny problematiky vyhledávání a rozpoznávání znaků. Dále byla provedena analýza dostupných nástrojů daných problematik. Teoretická část se rovněž zabývala strojovým učením v oblasti zpracování přirozeného jazyka, za účelem navržení systému detekce a opravy chyb.

V rámci praktické části byl popsán návrh a implementace programu umožňující integraci rozpoznávání textu v naskenovaných dokumentech a vyhledávání. Součástí řešení je návrh a ověření modelu pro opravy chyb OCR systému *Tesseract*, které kombinuje pravděpodobnostní mřížku a znakové jazykové modely. Oprava chyb přispívá ke zlepšení přesnosti vyhledávání celého systému.

Pro ověření úspěšnosti a přesnosti rozpoznávání textu byly vytvořeny experimenty, jejichž účelem bylo otestovat přesnost a chybovost rozpoznávání. V kolekci testovaných obrázků byly dokumenty různých kvalit a obsahů. Experimentálně byla nastavena optimální konfigurace celého systému. Při tomto optimálním nastavení se průměrná hodnota úspěšnosti (*Word Accuracy*) blížila 80 %. Pro kvalitně skenované dokumenty byla zaznamenána úspěšnost až 90 %.

U vstupního textu, kde se objevují cizí jména, názvy, zkratky či celé pasáže v jiném než českém jazyce, vykazuje program horší úspěšnost. Je to dáno především tím, že na takovéto situace není jazykový model natrénován.

Program by bylo možné vylepšit přidáním dalších metod strojového učení, například metody sumarizace textu, kdy by byl bezprostředně po fázi rozpoznávání textu vytvořen souhrn, který by obsahoval pouze důležité a klíčové termíny. S takovouto reprezentací by bylo možné snížit velikost indexu a v důsledku toho zefektivnit vyhledávání. Všechny body zadání byly splněny.

Seznam zkratek

API	Application Programming Interface
BMP	Windows Bitmap
CER	Character Error Rate
CLI	Command Line Interface
CSV	Command-Separated Values
DPI	Dots Per Inch
DSL	Domain-Specific Language
EM	Expectation-Maximization
GIF	Graphics Interchange Format
HTML	HyperText Markup Language
JMX	Java Management Extension
JSON	JavaScript Object Notation
MLE	Maximum Likelihood Estimation
MS	Microsoft
OCR	Optical Character Recognition
PDF	Portable Document Format
PHP	PHP: Hypertext Preprocessor
PNG	Portable Network Graphics
REST	Representational State Transfer
SW	Software
TIFF	Tagged Image File Format
VB	Visual Basic
WER	Word Error Rate
XML	EXtensible Markup Language
YAML	Ain't Markup Language

Literatura

[1] *FAQ tesseract-ocr/tesseract Wiki GitHub* [online]. 2016. [cit. 2017/02/08]. Dostupné z: `https://github.com/tesseract-ocr/tesseract/wiki/FAQ`.

[2] *Ghostcript API* [online]. Ghostscript, 2016. [cit. 2017/03/10]. Ghost4J. Dostupné z: `http://www.ghost4j.org`.

[3] *FrontPage - Solr Wiki* [online]. Apache, 2013. [cit. 2017/03/10]. Solr Wiki. Dostupné z: `https://wiki.apache.org/solr/`.

[4] Białecki, A. – Muir, R. – Ingersoll, G. Apache lucene 4. In *SIGIR 2012 workshop on open source information retrieval*, s. 17, 2012.

[5] Brown, P. F. et al. Class-based n-gram models of natural language. *Computational linguistics*. 1992, 18, 4, s. 467–479.

[6] Brychcín, T. Unsupervised methods for language modeling: technical report no. DCSE/TR-2012-03. 2012.

[7] Chen, S. F. – Goodman, J. An empirical study of smoothing techniques for language modeling. In *Proceedings of the 34th annual meeting on Association for Computational Linguistics*, s. 310–318. Association for Computational Linguistics, 1996.

[8] Eikvil, L. Optical character recognition. *citeseer. ist. psu. edu/142042. html*. 1993.

[9] Forney, G. D. The viterbi algorithm. *Proceedings of the IEEE*. 1973, 61, 3, s. 268–278.

[10] Gormley, C. – Tong, Z. *Elasticsearch: The Definitive Guide*. "O'Reilly Media, Inc.", 2015.

[11] Grainger, T. – Potter, T. *Solr in Action*. Manning Publications Co., 1st edition, 2014. ISBN 1617291021, 9781617291029.

[12] Hunt, P. et al. ZooKeeper: Wait-free Coordination for Internet-scale Systems. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'10, s. 11–11, Berkeley, CA, USA, 2010. USENIX Association. Dostupné z: `http://dl.acm.org/citation.cfm?id=1855840.1855851`.

[13] Jurafsky, D. – James, H. Speech and language processing an introduction to natural language processing, computational linguistics, and speech. 2000.

[14] Konkol, M. Brainy: A Machine Learning Library. In Rutkowski, L. et al. (Ed.) *Artificial Intelligence and Soft Computing*, 8468 / *Lecture Notes in Computer Science*. Springer, Heidelberg: Springer International Publishing, 2014. s. 490–499. doi: 10.1007/978-3-319-07176-3_43. Dostupné z: http://dx.doi.org/10.1007/978-3-319-07176-3_43. ISBN 978-3-319-07175-6.

[15] Kukich, K. Techniques for automatically correcting words in text. *ACM Computing Surveys (CSUR)*. 1992, 24, 4, s. 377–439.

[16] Leroux, B. G. Maximum-likelihood estimation for hidden Markov models. *Stochastic processes and their applications*. 1992, 40, 1, s. 127–143.

[17] Luburić, N. – Ivanović, D. Comparing Apache Solr and Elasticsearch search servers.

[18] Manning, C. D. et al. *Introduction to information retrieval*. 1. Cambridge university press Cambridge, 2008.

[19] McCowan, I. A. et al. On the use of information retrieval measures for speech recognition evaluation. Technical report, IDIAP, 2004.

[20] Mori, S. – Nishida, H. – Yamada, H. *Optical character recognition*. John Wiley & Sons, Inc., 1999.

[21] *Solr 6.1.0 solr-solrj API* [online]. Lucene, 2016. [cit. 2017/03/02]. SolrJ API Documentation. Dostupné z: https://lucene.apache.org/solr/6_1_0/solr-solrj/index.html?overview-summary.html.

[22] Tan, K. *Apache Solr vs Elasticsearch - the Feature Smackdown!* [online]. 2017. [cit. 2017/02/10]. Dostupné z: http://solr-vs-elasticsearch.com/.

[23] Wagner, R. A. – Fischer, M. J. The string-to-string correction problem. *Journal of the ACM (JACM)*. 1974, 21, 1, s. 168–173.

Příloha A

Uživatelská dokumentace

Spuštění aplikace

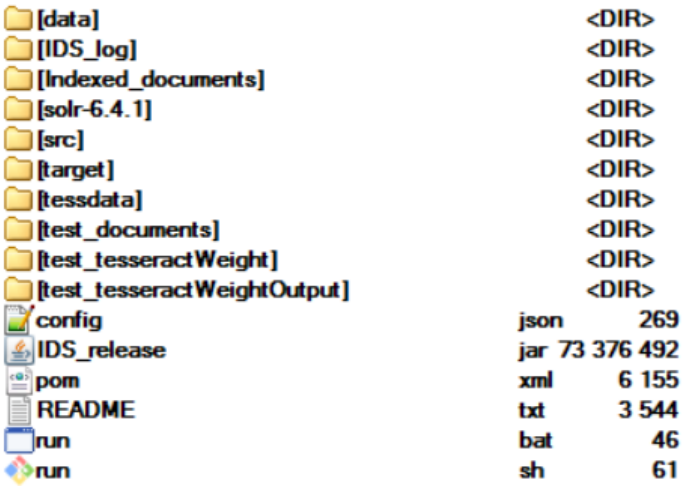
Aby bylo možné aplikaci spustit, musí být spuštěný *Apache Solr* (buď lokálně či na vzdáleném serveru). V případě vzdáleného serveru je nutné znát IP adresu a mít k tomuto serveru přístup.

Pro spuštění v prostředí Windows jsou veškeré knihovny (včetně *Tesseractu*) přibaleny v souboru JAR. Pro spuštění v prostředí operačního systému Linux je třeba mít nainstalovaný *Tesseract* aspoň verze 3.04. V opačném případě hrozí, že nastanou problémy při práci s **TessBaseAPI**.

Jelikož má aplikace grafické uživatelské rozhraní, tak spustit JAR soubor lze buď prostřednictvím grafického rozhraní operačního systému nebo v příkazovém řádku pomocí příkazu `java -jar -Dfile.encoding=UTF8 IDS.jar`. Pro spuštění na obou operačních systémech jsou připraveny spouštěcí skripty (`run.bat` pro Windows a `run.sh` pro Linux).

Struktura adresáře a konfigurační soubor

Struktura adresáře vypadá následujícím způsobem:



Obrázek 9.1: Struktura archivu

Popis archivu a spuštění aplikace obsahuje soubor README.txt

Pro samotné spuštění aplikace je připravený JAR soubor. Parametry spuštění obsahuje konfigurační soubor `config.json`. Ukázku konfiguračního souboru ukazuje

obrázek 9.2. Konfigurační soubor je rozdělen na tři části. První část slouží jako údaje

```
{
    "Solr url" : "localhost",
    "Solr port": "8983",

    "Data path" : "data/.",
    "Tessdata path" : ".",
    "Letter language model train data" : "data/train.txt",

    "Logging to file"       : "enable",
    "Log file path"         : "IDS_log/",
    "Log file extension"    : "txt",

}
```

Obrázek 9.2: Ukázka konfiguračního souboru

k připojení klienta na běžící server *Apache Solr*. Ve druhé části jsou uvedeny cesty k nezbytným souborům pro spuštění aplikace. Jedná se o trénovací data jazykového modelu a data (společně s jazyky) *Tesseractu*. Třetí část se týká logování.

Instance *Apache Solr* běží na dané IP adrese a tato IP adresa je uvedena v konfiguračním souboru. V případě, že vyhledávací systém je spuštěn na jiném stroji, je nutné do tohoto souboru vyplnit parametry *Solr url* a *Solr port* IP adresu a port.

Logování

Aplikace v rámci standardního chování loguje do souboru (defaultně `IDS_log.log`). Parametry logování (zapnutí logování, umístění logu, přípona souboru) lze upravit pomocí hodnot v konfiguračním souboru. Ukázka logovacího souboru je vidět na obrázku 9.3.

Logování rovněž rozlišuje závažnost (tzv. *severitu*) logovaných událostí. Obrázek 9.3 ukazuje jejich stručný výběr.

Logovací soubor tvoří chronologicky uspořádané řádky. Každý řádek začíná informací o závažnosti (severitě). Nejméně závažné zprávy jsou `DEBUG`, `NOTICE` a `INFO`. Na tyto stupně závažnosti navazuje `WARNING`. Poslední dva stupně jsou `ERROR` a nejzávažnější `CRITICAL`, při kterém se program ukončí.

Za stupněm závažnosti následuje datum a čas události. Jako poslední je na řádku uveden stručný výpis události. Na obrázku je například vidět několik nepovedených pokusů o start aplikace. Spodní část logů ukazuje povedený start aplikace.

```
DEBUG   [Pondělí, 27.3.2017 21:35:46] - --------------Checking service http://localhost:8983
CRITICAL[Pondělí, 27.3.2017 21:35:48] - Destination service (http://localhost:8983) is not available.
                                        Please make sure, if Apache Solr is running on desired destination.
CRITICAL[Pondělí, 27.3.2017 21:35:51] - Connection to Solr Server could not be established!
NOTICE  [Pondělí, 27.3.2017 21:35:51] - Shutting down the program
DEBUG   [Pondělí, 27.3.2017 21:36:22] - --------------Checking service http://localhost:8983
NOTICE  [Pondělí, 27.3.2017 21:36:27] - Shutting down the program
DEBUG   [Pondělí, 27.3.2017 21:36:32] - --------------Checking service http://localhost:8983
DEBUG   [Pondělí, 27.3.2017 21:36:32] - Destination service (http://localhost:8983) is available
ERROR   [Pondělí, 27.3.2017 21:36:40] - Solr Service cannot be contacted.
                                        Probably another service is running on localhost:localhost
CRITICAL[Pondělí, 27.3.2017 21:36:42] - Service Solr Server is not listening on port 8983
NOTICE  [Pondělí, 27.3.2017 21:36:42] - Shutting down the program
DEBUG   [Pondělí, 27.3.2017 21:37:11] - --------------Checking service http://localhost:8983
DEBUG   [Pondělí, 27.3.2017 21:37:12] - Destination service (http://localhost:8983) is available
INFO    [Pondělí, 27.3.2017 21:37:16] - Service bound on port 8983 is Solr
NOTICE  [Pondělí, 27.3.2017 21:38:06] - Creating word corrector
NOTICE  [Pondělí, 27.3.2017 21:38:58] - Program IDS - Intelligent Document Searching has been started
```

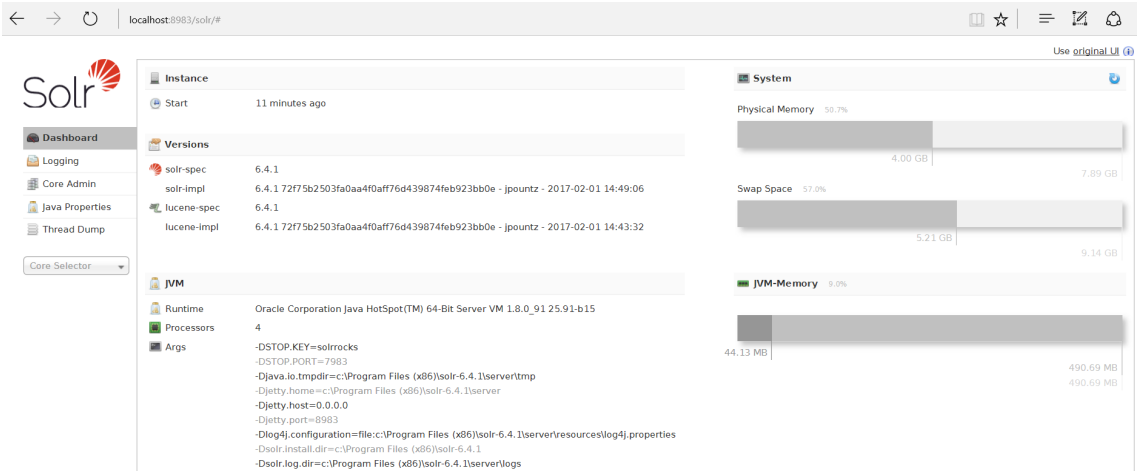
Obrázek 9.3: Ukázka souboru logování

Klasifikační třídy

Pro účely testování klasifikace byly vytvořeny následující třídy dokumentu: *Zákony, zprávy* a *smlouvy*. Přidání další třídy je jednoduché. Je potřeba pouze vytvořit soubor s trénovacími daty, který se pojmenuje stejně, jako nová třída. Takto vytvořený soubor je poté nutné umístit do adresáře `data/classification_data`.

Webové rozhraní

Kromě klienta v Javě je možné se k běžícímu *Apache Solr* připojit prostřednictvím webového rozhraní. Ukázku úvodní strany webového rozhraní ukazuje obrázek 9.4. Webové rozhraní umožňuje přehledně zobrazit údaje o běžící instanci *Apache Solr*,



Obrázek 9.4: Ukázka webového rozhraní

včetně možnosti zadání dotazu.

Grafické uživatelské rozhraní programu

Při spuštění se klient pokusí navázat spojení s *Apache Solr* serverem. Pokud se spojení navázat nepodaří, program vygeneruje chybové hlášení. Pokud se spojení podaří navázat bude uživatel vyzván k výběru *Core*.

Program poté začne načítat soubory ze složky `data` a začne se trénovat klasifikátor a jazykový model. Grafické uživatelské rozhraní programu obsahuje menu panel karet, které jsou odlišeny tématicky. První dvě karty aplikace (*SOLR query* a *Documents*) kombinují nalezené optimální nastavení systému korekce chyb a tvoří hlavní část aplikace. Další panely: *Tesseract*, *Document Classificator* a *Word Corrector* jsou v aplikaci z důvodu vyzkoušení různých nastavení jednotlivých částí systému a k testování.

Menu

V horní části programu je umístněné menu, které obsahuje:

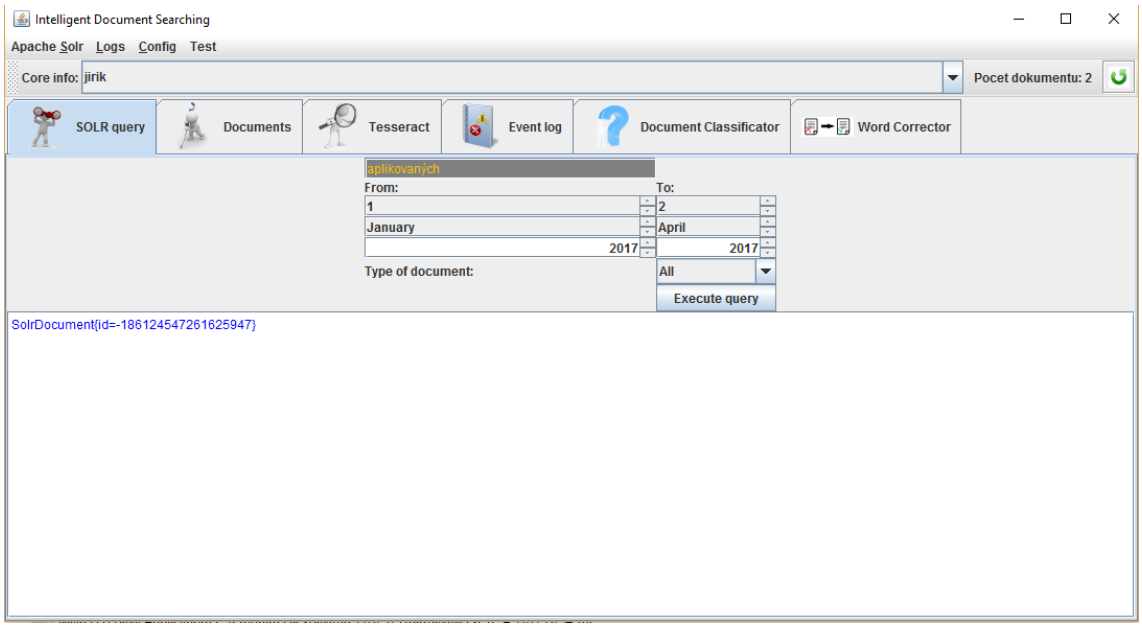
- sekci **Apache Solr**, která zobrazuje informace o jednotlivých *Cores*,
- sekci **Logs**, která obsahuje prohlížeč logů v souborech,
- sekci **Config**, která zobrazí konfigurační parametry, se kterými byl program spuštěn,
- sekci **Test**, díky které lze spustit test na přesnost a chybovost se zadanou vahou *Tesseractu*,
- sekci **About**, která zobrazuje informace o projektu.

Přesnost a chybovost testu se spočítá pouze pro dvojici souborů txt (*gold data* a png umístěné ve složce `test_tesseractWeight`.

Dále bude pro každou kartu uveden screenshot a stručný popis.

Dotazy

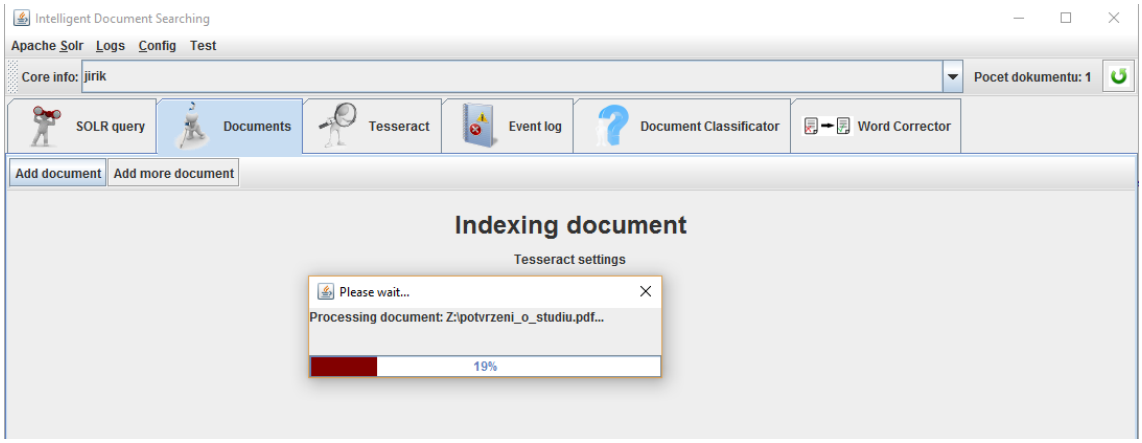
První karta, která se po startu programu zobrazí je karta pro *SOLR query*. Panel dotazů je horizontálně rozdělen na dvě části. V horní části je formulář na zadání dotazu. První řádek formuláře je hledaný text (*fulltextové vyhledávání*). Další část formuláře se týká vyhledávání podle časového intervalu. Poslední část formuláře je možnost výběru typu dokumentu. Po stisknutí tlačítka *Execute query* se v dolní části objeví ID vyhovujících dokumentů. V případě, že vyhledávacím parametrům neodpovídá žádný dokument, je vráceno `No results`.



Obrázek 9.5: Ukázka panelu dotazů

Dokumenty

Druhý panel nabízí možnost přidání jednoho či více dokumentů. Po stisknutí tlačítka přidání dokumentu se zobrazí průzkumník souborů, kde je možné najít požadovaný dokument ke zpracování. Akceptovatelné dokumenty jsou pouze pdf a png. Při zvolení jiného formátu program vygeneruje chybové hlášení. Po vybrání dokumentu se spustí dialog, ve kterém je vidět průběh zpracování dokumentu. Po stisknutí tlačítka přidání více dokumentů se spustí stejný průzkumník souborů, který ale očekává, že se vybere celá složka. Po vybrání složky se posléze zpracují všechny dokumenty pdf či png v dané složce.



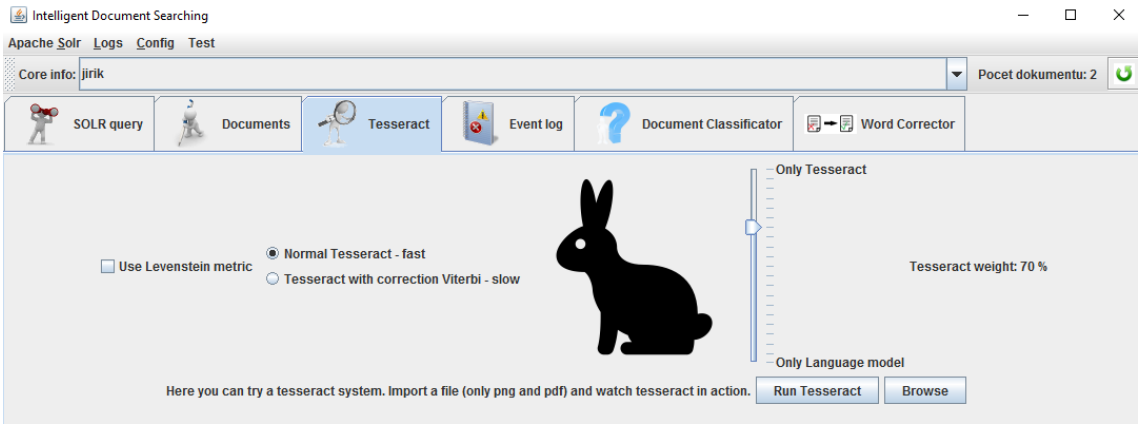
Obrázek 9.6: Ukázka panelu dokumentů

V dokumentech se nejprve rozpozná a opraví text. Poté se jim přidělí ID a zaindexují se. Program vytvoří složku *Indexed_documents*, ve které budou zaindexované dokumenty, pojmenované podle svého ID. Při dotazování program vrátí ID vyhovujícího dokumentu, a díky tomu je možné ho ve složce *Indexed_documents* nalézt.

Tesseract

Účel třetího panelu je otestování a vyzkoušení analýzy dokumentů pomocí systému *Tesseract*. Na panelu se nachází různá nastavení, kterými lze ovlivnit kvalitu a rychlost OCR analýzy.

Nejprve je nutné vybrat soubor s dokumentem (tlačítko *Browse*) a zvolit váhu systému *Tesseract* pomocí posuvníku.

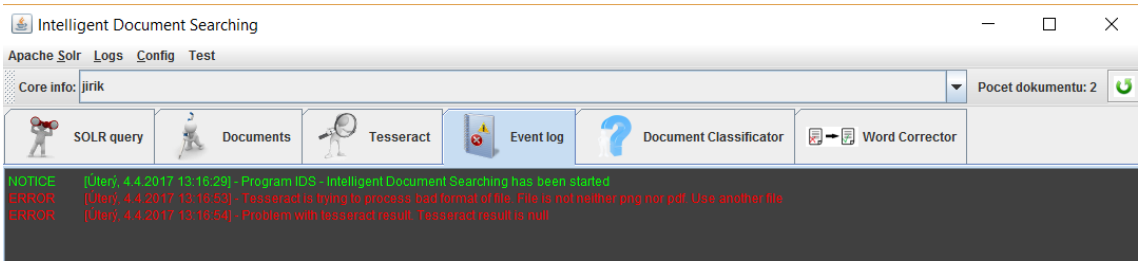


Obrázek 9.7: Ukázka panelu nastavení vah *Tesseractu*

Po stisknutí tlačítka *Run tesseract* se spustí samotná analýza a zobrazí se výsledek. Panel dokumenty neindexuje ani nezpracovává, pouze zobrazuje výsledek s daným nastavením vah.

Log událostí

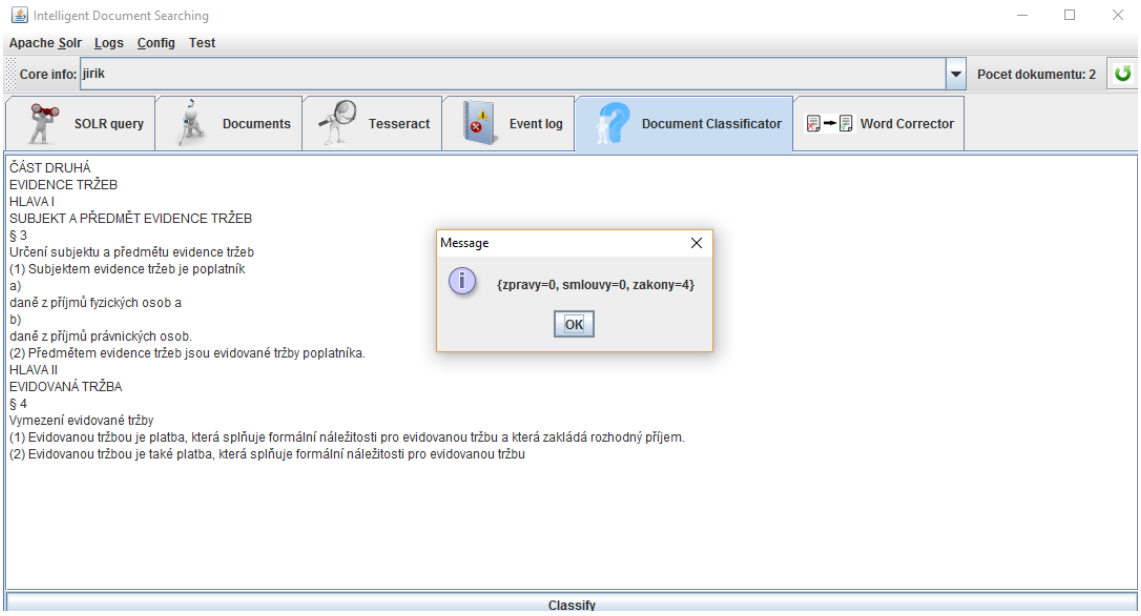
Další panel je pouze prohlížeč událostí, které se logují do souboru. Pro prohlížení starších logů či logů uložených do souboru je připraven prohlížeč, ke kterému se lze dostat pomocí menu v horní části programu.



Obrázek 9.8: Ukázka prohlížeče logů událostí

Klasifikace

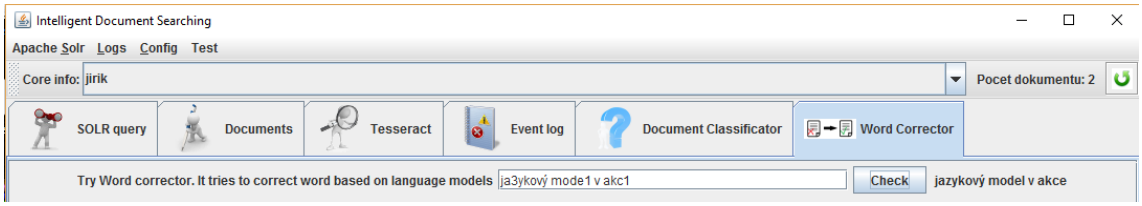
Další panel v pořadí je testování klasifikace dokumentů. Panel obsahuje pouze prostor pro vložení textu a tlačítko *Classify*. Ukázku je možné vidět na obrázku 9.9.



Obrázek 9.9: Ukázka panelu zkoušení klasifikace dokumentů

Word corrector

Posledním panelem je testování a vyzkoušení opravování slov pomocí jazykových modelů.



Obrázek 9.10: Ukázka panelu zkoušení *Word correctoru*

Od uživatele se očekává zadání textu do připraveného textového pole. Po stisknutí tlačítka *Check* bude zobrazena navržená oprava. *Word corrector* projde jednotlivé znaky slova a aplikuje jazykový model na opravu.