table

Sem vložte zadání Vaší práce.

České vysoké učení technické v Praze
Fakulta informačních technologií
Katedra softwarového inženýrství

Diplomová práce

Crawler zaměřený na sběr Web API dokumentace

Bc. Jiří Šmolík

Vedoucí práce: Ing. Milan Dojčinovski

2. května 2015

Poděkování

Děkuji vedoucímu práce panu Ing. Milanu Dojčinovskému za připravení zajímavého tématu, jeho vstřícnou pomoc a cenné rady a připomínky v průběhu tvorby práce. Děkuji také mé rodině za poskytnutí příjemného zázemí a mé přítelkyni Anně Hrabalové za její bezmeznou podporu v době psaní této práce a za korekturu následujícího textu.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval(a) samostatně a že jsem uvedl(a) veškeré použité informační zdroje v souladu s Metodickým pokynem o etické přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 46 odst. 6 tohoto zákona tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou, a veškeré jejich dokumentace (dále souhrnně jen „Dílo“), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla, a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené. Každá osoba, která využije výše uvedenou licenci, se však zavazuje udělit ke každému dílu, které vznikne (byť jen zčásti) na základě Díla, úpravou Díla, spojením Díla s jiným dílem, zařazením Díla do díla souborného či zpracováním Díla (včetně překladu), licenci alespoň ve výše uvedeném rozsahu a zároveň zpřístupnit zdrojový kód takového díla alespoň srovnatelným způsobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Praze dne 2. května 2015 .....

České vysoké učení technické v Praze
Fakulta informačních technologií

Odkaz na tuto práci

Šmolík, Jiří. *Crawler zaměřený na sběr Web API dokumentace.* Diplomová práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2015.

Abstrakt

Diplomová práce se zabývá sběrem a analýzou dokumentací webových API zaměřeným crawlerem, který na Internetu hledá dokumenty odpovídající uživatelem zadané frázi. Následně o každém sesbíraném dokumentu rozhodne, zda je API dokumentace či není. Pro klasifikaci dokumentů je využita řada algoritmů strojového učení s učitelem.

Klíčová slova zaměřený crawler, web api dokumentace, webové služby, strojové učení s učitelem

Abstract

The diploma thesis tackles the crawling and analysis of Web API documentation with focused crawler, which searches the Internet for user-specified documents. Each document is then classified as either API documentation or Other. The classification part uses a number of supervised machine learning algorithms, which are applied to crawled documents to decide, whether document is or is not a Web API documentation.

Keywords focused crawler, web api documentation, web services, supervised machine learning

Obsah

Seznam obrázků

Seznam tabulek

Úvod

Motivace

Orientace na služby (ang. *service-orientation*) při vývoji softwarových aplikací klade důraz na znovupoužitelnost jednotlivých částí tak, že dílčí komponenty a jejich funkce jsou vystaveny skrze rozhraní nezávislém na programovacím jazyce. Tyto služby dostupné na webu jsou označovány jako Web API.

Ačkoliv libovolný poskytovatel služby (ang. *service provider*) má možnost si vytvořit vlastní způsob pro vystavení služby, dnešním trendem je implementovat rozhraní své webové aplikace s ohledem na principy REST. Díky velkým hráčům, jakými bezpochyby jsou Facebook, Flickr, Twitter, Amazon a další, kteří nabízejí přístup ke svým datům a funkcionalitě skrze tato webová rozhraní, jednoduchosti používání této technologie a standardizaci se tento způsob komunikace stává čím dál rozšířenějším. Takovéto služby je možné také množit pomocí mashup aplikací, kdy zkombinováním několika služeb vznikne nová s přidanou hodnotou.

Ovšem znovupoužitelnost a rychlost rozmachu služeb na webu jde v tomto případě proti sobě. Zatímco v jedné oblasti (např. nějaká společnost) se dá snadno zjistit, jaké funkce a jak systém nabízí, na Webu se to stává problémem. Webový graf je obrovský a o většině Web API se aplikační vývojář ani nemusí dozvědět. Jeho záchranou mohou být registry webových služeb, jako je např. ProgrammableWeb [1]. Ten se považuje za největší zdroj novinek a informací o programovacích rozhraních na světě. Jak se ale ukazuje, takto obrovské množství aplikačních rozhraní je jen velmi těžko udržitelné a i tento registr se stává pomalu nepoužitelným – vložené záznamy jsou leckdy chybné či zastaralé. Tyto záznamy jsou spravovány lidmi a ačkoliv jsou při vložení schváleny a zkontrolovány, zda alespoň odkazy fungují, časem dojde k restrukturalizaci cílového webu, změně endpointů atp. Je tedy nasnadě tento problém řešit. Nejlépe automaticky. Průběžně kontrolovat. Mít registr s platnými údaji.

Problém a nástin řešení

Problém automatizace sběru webových API spočívá v jejich nalezení a identifikaci. Tato práce je zaměřena sběr Web API dokumentací, neboť se předpokládá, že každé Web API bude mít (někde na webu) dokumentaci k používání služeb poskytovatele. Naneštěstí jsou Webová API dokumentována většinou v HTML textu v libovolné struktuře a jazyce určeným pro člověka a neobsahují žádné prostředky usnadňující automatizované nalezení či invokaci služby.

S těmito znalostmi se dá rozdělit problém na dvě základní fáze, s kterými tato práce pracuje:

- Sběr webových dokumentů. Tato část je zaměřena na to, jak vůbec najít relevantní dokumenty. K vyhledávání jsem využil webového vyhledávače Google, konkrétně jeho customizovatelné části Google Custom Search Engine (CSE)[1].
- Klasifikace dokumentů. Tj. rozhodnout o daném dokumentu, zda je, či není nějakou Web API dokumentací. K tomu bude využito algoritmů strojového učení nad trénovacími daty a následná klasifikace sesbíraných dokumentů.

[1] https://cse.google.com/cse/

Kapitola 1

Cíl práce

Cílem práce bude vytvořit co možná nejpoužitelnější crawler[2] pro sběr Web API dokumentací na webu, který vhodně projde určitou část webového grafu a zaindexuje specifikované dokumenty, o kterých následně rozhodne, zda jsou, či nejsou dokumentací webového API.

Na závěr bude crawler podroben testu kvality, tj. jak dobře si počínal při sběru a ohodnocování dokumentů. Pozorován bude rozsah nalezených stránek, domén a poskytovatelů, ale i kvalita a přenost klasifikace. Jelikož se jedná o binární klasifikaci dokumentů, využiji k ohodnocení kvality výsledků statistických ukazetelů Přesnosti (ang. *Precision*) a Úplnosti (ang. *Recall*).

Kvalitativní hodnocení se bude testovat na řadě algoritmů strojového učení při různých reprezentacích dokumentů. Pak z výsledků bude jasnější, kterou metodiku zvolit při dalším počínáním nad tímto problémem.

1.1 Omezení

1.1.1 Rozsah záběru crawleru

Ačkoliv by bylo jistě užitečné crawlovat celý web a mít detailní přehled o tomto prostoru jako celku, bude prostor zredukován jen na jeho zlomek. Zejména proto, že nedisponuji dostatkem prostředků jakými jsou hlavně fyzické diskové místo a dostatečná rychlost Internetového připojení. Crawlování bude vycházet z několika desítek vstupních bodů získaných z Google CSE. Vyhledávání a cílové dokumenty tak budou zaměřeny na požadovanou oblast specifikovanou klíčovým slovem/frází, kterou najde vyhledávač.

[2] Internetový bot (program), který systematicky brouzdá po World Wide Webu za účelem indexace nalezených dokumentů.

1.1.2 Crawler a indexace

Samotný crawler by náročností vyšel na samotnou práci podobného rozsahu, proto využiji již existující, osvědčené a navíc open-source řešení – Apache Nutch (podrobněji v 4.1.2, dále jen Nutch). Modulární crawler navržen pro práci nad clusterem Hadoop pro indexaci může nativně využít několik databází, pro účely práce ale zvolím Apache Solr (více v 4.1.3, dále jen Solr), který je samotný též vhledávací platformou.

Solr pak bude vhodným nástrojem pro vyhledávání v nacrawlovaných dokumentech. Buď přímo pomocí svého adminitrátorského rozhraní pro dotazy, anebo jen nad ním možno vybudovat aplikaci, která využije REST API Solru.

1.1.3 Klasifikace za běhu

Po několika úváhach se ustálil názor od sebe zcela oddělit fáze crawlování a klasifikace dokumentů a zařadit je tak za sebe – tedy prvně dojde ke sběru dokumentů a až pak se spustí klasifikační algoritmy nad nasbíranou množinu dokumentů. Tento způsob má výhodu v tom, že nedochází ke zdržování při stahování dokumentů a je možné pak nasbírané dokumenty kdykoliv překlasifikovat podle jiných algoritmů a jejich nastavení.

1.2 Postup

Pro realizaci cíle bude postup následující.

1. Vytvoření a konfigurace vyhledávače Google CSE.

   Vyhledávač Google Custom Search Engine bude využíván pro nalezení kandidátů jako iniciálních bodů sběru crawleru. Ideou je usnadnění vývojáři nalezení API pro konkrétní oblast zájmu. Jestliže se tedy dá tato oblast zájmu popsat slovy, využijí se výsledky vyhledávače právě na tyto slova.

2. Vytvoření trénovací množiny dokumentů $\mathcal{D}_T$.

   a) Web API dokumentace $\{\mathcal{A}_T : \delta \in \mathcal{D}_T \wedge \text{„}\delta \text{ je API dokumentací“}\}$.

   b) Ne-API-dokumentace $\mathcal{D}_T \backslash \mathcal{A}_T$.

   Pro klasifikaci dokumentů je využito strojového učení na trénovací sadě dokumentů. Dokumenty typu Web API dokumentace budou čerpány z webu ProgrammableWeb. Ostatní dokumenty budou vybírány z výsledků Google CSE, které nejsou dokumentací. Celá trénovací množina je vytvořena a jednotlivé dokumenty klasifikovány ručně jednou osobou (autorem).

3. Slovní analýza dokumentů $\mathcal{A}_T$ pro vhodnou reprezentaci kvůli klasifikaci.

   Klasifikační algoritmy pracují s vektory rysů. Je proto vhodné dokumenty reprezentovat také tak. V tomto případě budou rysy slova a jejich hodnoty obsažnost slov v dokumentu. Tyto společná slova se naleznou napříč všemi dokumenty, ale s výjimkou např. stopslov[3]. Dokument pak bude reprezentován binárním vektorem, kde i-tá položka je 1, právě když četnost i-tého slova překročila stanovenou prahovou hodnotu.

4. Implementace crawleru, integrace komponent (Nutch, Solr, Google CSE, Weka) a klasifikace dokumentů.

   Crawler bude úkol zpracovávat po etapách. Nejprve zajistí výchozí URL dokumentů z vyhledávače. Po té pomocí Nutch nacrawluje část webového prostoru a zaindexuje do Solr. Nakonec pomocí zvoleného klasifikačního algoritmu aplikuje znalost z testovací sady na zajištěných dokumentech které označí výsledkem – zda je či není dokument API dokumentací a s jakou pravděpodobností.

5. Testování, kvantitativní a kvalitativní hodnocení.

   Crawler bude otestován z různých pohledů, jak široký prostor dokáže zpracovat, v jak velké oblasti se pohybuje a jestli přináší spolehlivé výsledky. Bude vyhodnocena kvalita klasifikace podle měr Přesnosti a Úplnosti za různých podmínek. Podmínky mohou být různé reprezentace jednotlivých dokumentů i různé algoritmy pro strojové učení.

[3]Seznam slov, které se vyskytují v přirozeném jazyce často a nenesou žádnou významovou informaci. Typicky jsou to předložky, spojky, členy atp.

Kapitola 2

Analýza

V této kapitole vymezím pojmy a základní stavební bloky potřebné pro hlubší pochopení parciálních částí programu. Zadefinuji a podrobněji popíši problém automatizace identifikace webového API a jakých bude využito postupů při řešení tohoto problému.

2.1 Vymezení pojmů

2.1.1 Crawler

Webový crawler, někdy též nazýván *spider* nebo *ant* je internetový bot, který systematicky prochází stránky WWW. Typicky za účelem indexování nalezeného obsahu. Často bývají základem pro internetové vyhledávače, které využívají index postavený nad nasbíranými dokumenty pro urychlení vyhledávání.

Crawler začíná s počátečními URL, které má navštívit. Těm se říká seedy (ang. *seeds*). Pro každé URL z těchto seedů identifikuje hypertextové odkazy a přidá si je do seznamu adres k navštívení – seznam je označován jako *crawl frontier*. Tyto adresy jsou dále rekurzivně navštěvovány podle množiny pravidel a pokud crawler dokumenty indexuje, což je většinový případ, uloží zároveň obsah dokumentu do svého úložiště. Ikdyž se tak obsah URL může v čase měnit, crawler si uchovává svojí lokální kopii – *snapshot*.

Při navštěvování dokumentů se bot potýká s problémem navštívení unikátního obsahu pouze jednou. Existují téměř nekonečné kombinace URL parametrů, ale přitom všechny tyto URL směřují na stejný obsah. Proto musí crawler pečlivě zvolit, jaké URL navštíví a to ve velmi omezeném časovém úseku. Na druhou stranu v případě velkého množství linků dochází k jejich uspořádání podle priority. Z tohoto seznamu je pak zpracována pouze část označena za důležitou. Důležitost v oblasti stránek se hodnotí většinou podle počtu linků nebo návštěv.

2.1.1.1 Politika crawlování

Každý slušně naučený crawler by měl respektovat obecné politiky crawlování. Ty se dají podle [2] rozdělit na:

- Politika výběru (ang. *selection policy*). Uvádí, jaké stránky se mohou stáhnout.
- Politika znovunavštívení (ang. *re-visit policy*). Uvádí, kdy je možné kontrolovat provedené změny stránek.
- Zdvořilostní politika (ang. *Politeness policy*). Uvádí, jak se vyhnout přetížení WWW stránek.
- Politika paralelizace (ang. *Parallelization policy*). Uvádí, jak koordinovat distribuované crawlování.

Politika výběru. Mějme celkovou velikost Webu, pak i ty největší vyhledávače pokrývají svým indexem pouze 40-70% indexovatelného webu podle studie z r. 2009 [3] a odhadu z letošního roku circa 67% [4]. Protože se tak crawler vždy dostane jen k části webových stránek, je žádoucí tuto část zúžit právě na nejvíce relevantní dokumenty a ne pouze náhodný vzorek.

Jak bylo již zmíneno, jako metrika hodnocení důležitosti stránek vychází z hodnotící funkce, zohledňující popularitu stránky ve smyslu počtu linků, návštěv i na základě její URL (např. náležitost k určité doméně).

Politika znovunavštívení. Web se ve své podstatě velmi rychle mění a vyvýjí a celé jeho nacrawlování by mohlo trval i měsíce. Po tak dlouhé době by mnohé obsahy nashromážnědých dokumentů byly zastaralé, upravené či smazané. Je tedy nutné dokumenty pravidelně obměňovat. K tomu se často používají funkce čerstvosti a stáří.

- Čerstvost stránky p je binární funkcí času t:

$$F_p(t) = \begin{cases} 1 & \text{pokud } p \text{ je identický s lokální kopií} \\ 0 & \text{jinak} \end{cases}$$

- Stáří stránky indikuje jak moc je lokální kopie stránky zastaralá. Stáří stránky p v čase t je definováno jako:

$$A_p(t) = \begin{cases} 0 & \text{pokud } p \text{ je změněna v čase } t \\ t - \text{čas změny } p & \text{jinak} \end{cases}$$

Cílem crawleru je tak minimalizace času, po kterou jsou stažené stránky zastaralé. Problém se dá modelovat jako systém hromadné obsluhy s několika vstupními frontami a jedním odbavovacím serverem.

Existují dvě známé metody.

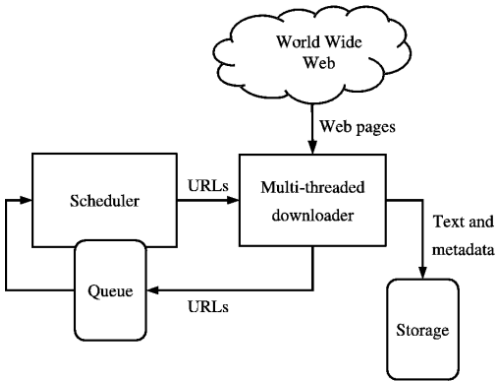
- Rovnoměrná politika. Všechny stránky jsou obměňovány ve stejných intervalech.

- Úměrná politika. Stránky měnící se častěji jsou znovunavšťíveny častěji.

Zdvořilostní politika. Většina serverů byla koncipována na běžný provoz a navštětování lidmi, ale takový automatizovaný crawler dokáže stránku navštívit mnohem častěji v krátkém časovém úseku, což může vést ke snížení výkonnosti serveru a zvýšení doby odezvy pro ostatní uživatele.

Částečným řešením pro tento problém je Protokol pro zákázání přístupu robotům (ang. *Robots exclusion protocol*), jehož instrukce jsou umístěny na serveru v souboru robots.txt. Nestandardizovanou částí protokolu jsou informace o tzv. *crawl delay*, která určuje, po jaké době je možné znovu přistoupit ke zdroji. Crawl delay je respektována většími crawlery od Google, MSN, Yahoo! apod. Tyto intervaly se jinak pohybují od 10 vteřiny do 3–4 minut.

Politika paralelizace. Paralelní crawler je crawler využívající více procesů ve stejný okamžik. Cílem je maximalizovat využití dostupného pásma připojení a vyhnout se stahování jedné stránky více procesy naráz. V tomto případě musí jednotlivé procesy sdílet všechny objevné URL.



Obrázek 2.1: Architektura obecného crawleru

2.1.1.2 Identifikace crawleru

Každý crawler by se měl identifikovat svým označením, které se objevuje v HTTP požadavcích v User-agent hlavičce. Tato informace je na serverech logována a je tak přehled o tom, jaké crawlery a jak často stránku navštívily.

2.1.2 Zaměřený crawler

Zaměřený crawler (ang. *Focused crawler*) je crawler, který shromažďuje webové stránky odpovídající určitým kritériím, podle kterého volí pořadí ve svém

crawl frontier. Kritéria bývají například izolace na jedné doméně či stránky spadající do určité kategorie.

První fází býva vytvoření klasifikátorů pomocí trénovací sady obsahující označené položky informací o tom, do jaké spadají kategorie. Tento klasifikátor je pak využit při brouzdání webovým prostorem a jsou sbírány pouze dokumenty, které klasifikátor předpověděl jako žádoucí.

Hlavním nedostatkem těchto crawlerů je, že jako hlavní metriku v hodnocení stránek získávají z podobnosti s danou kategorií, tj. hodnocení klasifikátoru. Nevyužívá se tak žádná další metrika jako je například PageRank, což nemusí vést k žádoucím výsledkům.

2.1.3 Webové API

Jedná se o aplikační programovací rozhraní pro webový server a klient. V obecném pohledu je dvojí:

- Server API, jež je dostupné přes web a požadavky na toto rozhraní obstarává aplikace běžící serveru. Typicky dochází k výměně dat ve formátu XML nebo JSON a komunikace probíhá přes protokol HTTP.
- Klientské API, které napomáhá rozšířit funkce HTTP klienta (např. prohlížeče) formou pluginů.

Dále v textu se myslí serverové API, není-li řečeno jinak.

Narozdíl od webových služeb, definovaných konzorciem W3C, aby k nim mohlo být přistupováno automaticky nebo částěčně automaticky pomocí WSDL a UDDI a komunikace je založena na XML, webové API standardizováno nikterak není a v jednotlivých implementacích se nedájí přehlédnout odlišnosti.

V dnešní době existuje podle [5] několik trendů:

- RESTful
- Styl RPC
- Hybridní

2.1.3.1 Architektura REST / RESTful

REST je architektonický styl pro distribuované systémy založené na hypermédiích[4]. K tomu využívá omezení a principů z ostatních síťových architktur a kombinuje je s omezením na Webu a požadavky na jednotném rozhraní [6].

Základní principy RESTu dle [6] jsou

[4]Rozšíření hypertextu o grafiku, zvuk a video.

- Klient-server (ang. *Client-server*). Oddelění uživatelského rozhraní od práce s daty, vede ke zvýšení portability klientské části.
- Bezestavost (ang. *Stateless*). Komunikace mezi klientem a serverem již není vázána kontextem předchozích zpráv. Každá zpráva obsahuje vše nutné k obsluze požadavku serverem.
- Kešovatelnost (ang. *Cacheable*). Ve světe WWW může klient odpovědi kešovat, proto i REST musí toto respektovat a označovat svoje odpovědi příslušnou dobou platnosti.
- Vícevrstvý systém (ang. *Layered system*). Pro lepší škálovatelnost může požadavek od klienta k serveru cestovat přes několik prostředníků, každý přitom interaguje pouze s přímými sousedy.
- Jednotné rozhraní (ang. *Uniform interface*). Jednotné rozhraní rozlišuje REST a ostatní webově založené architektury a vede k celkovému zjednodušení systému a i komunikace se stává přehlednější. Implementace se tak mohou přirozeně oddělit a vyvíjet samostatně. Tento princip klade čtyři omezení na jednotná rozhraní:
  - Identifikace zdrojů. Každý zdroj (ang. *resource*) je identifikován svým URI.
  - Manipulace skrze reprezentace zdrojů. Klient nemá přístup ke zdroji jako takovému, nýbrž pouze k jeho reprezentaci/reprezentacím ve formě HTML, XML, či JSON. Včetně metadat o zdroji.
  - Samopopisné zprávy. Každá zpráva obsahuje informace o postupu, jak zprávu zpracovat.
  - Hypermedia jako aplikační stav (*HATEOAS*). Klient přechází mezi stavy aplikace přes hypermédia v odpovědích.

Rozdíl mezi obecnou architekturou REST a RESTful je ten, že RESTful je implementována na protokolu HTTP využívající k modifikaci zdrojů nativní hlavičky protokolu (HEAD, GET, POST atd.).

Ukázky požadavku

```
HTTP GET http://url/.../News/85412
HTTP DELETE http://url/.../News/85412
HTTP GET http://url/.../News?category=foreign
HTTP POST http://url/.../News Author=John&Date=today&...
```

2.1.3.2 RPC API

V kontrastu s REST se RPC styl webových API zásadně liší v (ne)používání HTTP metod pro práci se zdroji. Namísto toho jsou definovány vlastní operace

na specifikovaných URI. Například framework ASP.NET automaticky mapuje sloveso v URL na požadovanou metodu kontroleru obsluhující API.

RPC API tak vystavuje svojí funkcionalitu skrze rozhraní podobnější spíše programovovacímu jazyku, což je zcela odlišný způsob od mechanizmu založeném na službách orientovaných na zdroje.

Ukázky požadavku[5]

```
HTTP GET http://url/.../getNews/85412
HTTP POST http://url/.../deleteNews/85412
HTTP GET http://url/.../getNews/foreign
HTTP POST http://url/.../createNews Author=John&Date=today&...
```

2.1.3.3 Hybridní API

Jak již jméno naznačuje, jedná se o směs RESTful a RPC přístupu. Hybridní API definuje svoje vlastní operace, které se invokují nikoliv příslušnou HTTP metodou, ale na základě informace obsažené ve zprávě. Hybridní API tak může definovat například `getNews` na metodě POST a `createNews` skrze metodu GET.

Využití takovéhoto API může být problematické, protože není garantována bezpečnost manipulace s daty ve smyslu jejich neúmyslného sepnutí – v případě, že by byla operace mazání implementovaná na HTTP metodě GET, mohl by crawler při běžné práci značnou část zdrojů neúmyslně smazat při rutinním zkoumání webu.

Ukázky požadavku

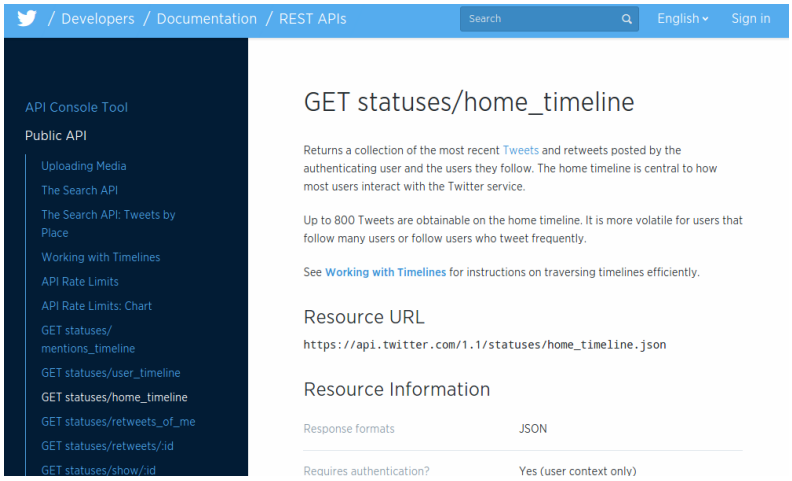
```
HTTP GET http://url/endpoint?method=getJson&params=85412
HTTP GET http://url/endpoint?method=create&params=John;...
HTTP POST http://url/endpoint?method=delete&params=85412
```

Studie z roku 2010 [5], která se zabývala výzkumem typů webových API a který byl proveden na API rozhraních z repozitáře ProgrammableWeb (který už tehdy jevil známky nedokonalosti ohledně zastaralých dat), byly jednotlivé typy aplikačních rozhraní v takovémto zastoupení:

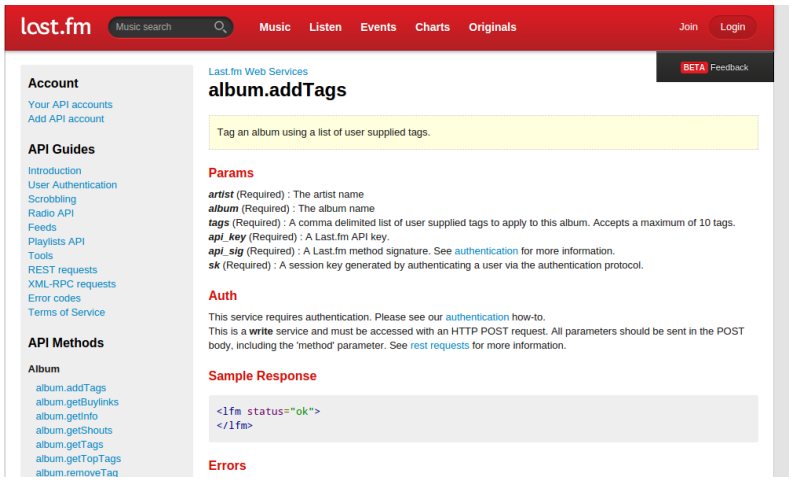
Typ API	Zastoupení [%]
RPC	47,8
RESTful	32,4
Hybridní	19,8

Při letmém ověřování, neboť to není hlavní náplní této práce, se všechny API zmíněné ve studii u hybridních a RPC posunuly směrem k RESTful architektuře. Některé plně, některé částečně. Je tedy důvod se domnívat, že Web API dnes je tvořeno zejména REST/RESTful API.

[5] Může se lišit v různých implementacích.



(a) http://dev.twitter.com



(b) http://last.fm/api

Obrázek 2.2: Ukázky web API dokumentací

2.1.3.4 Web API dokumentace

Nyní už víme, jak vypadá Webové API a jak se k němu přistupuje. Samotná jeho existence by ale byla zbytečná, kdybys nikdo nevěděl o jeho existenci a jak jej využít. Proto každý, kdo chce poskytnout svoje API veřejnosti, vytvoří dokument, který popisuje, jak může klient k API přistupovat, jak je volat, jak vypadají požadavky, odpovědi serveru apod. Oblast dokumentace web API trpí nedostatkem existence široce používaného standardu a většinou se tak jedná o HTML dokument v takové podobě, jak vyhovuje poskytovateli služby.

Problém identifikace. Na rozdíl od webových služeb využívajících proto-

kolu SOAP [7], jejichž popis volání je popsán ve standardizovaném formátu WSDL [8], pro dokumentace webových API žádný masově používaný standard není – ačkoliv jsou k tomu navrženy popisovací jazyky jako WADL [9]. Tyto standardy by značně napomohly řešení tohoto problému, nicméně poskytovatelé služeb je neimplementují a dokumentace jsou popsány nestrukturovaným textem v HTML stránce určené ke konzumaci člověkem, nikoliv strojem. Na tento fakt se zaměřuje návrh o anotování existujících HTML stránek hRest [10], který je založen na mikroformátech. I tato možnost zatím stále čeká na svoje využití v široké veřejnosti.

Zatím tedy stále nejrozšířenější podobou web API dokumentace je popis v přirozeném jazyce na HTML stránce. Aby toho nebylo málo, každý poskytovatel má dokumentaci napsanou ve zcela jiném sledu a do různé úrovně detailu – některé zobrazují ukázky komunikace, jak vypadá ukázkový požadavek, jak příslušná odpověď, často chybí chybové kódy, které mohou přijít v odpovědi a klient neví, jak na ně reagovat nebo chybí typy hodnot, jakých mohou položky nabývat apod.

Z popisu problému patrné, že jen zjistit, zda dokument popisuje webové API, je poměrně náročný úkol. V části 2.2 budu zjišťovat, jak se s tímto problémem potýkali ostatní.

Problém nalezení. Jak bylo zmíněno, dalším problémem je samotné nalezení těchto API. Vývojář hledající nějaké API za účelem jeho využití ve své aplikaci má možnosti dvě:

1. Přímé využití API – API bylo přímo doporučeno nebo využito v minulosti, nebo

2. Využití některého z veřejných API repozirářů – např. ProgrammableWeb [1], nebo

3. použít Internetový vyhledávač – např Google[6].

Přímé využití neskýtá žádný problém, API je nalezeno, stejně jako příslušná dokumentace, kterou bude muset vývojář nastudovat, pokud ji už nezná a může ji rovnou využít.

Při využití možnosti 2 se vývojáři naskýtá možnost využít přes 13 000 API, které ProgrammableWeb k tomuto podle vlastních informací obsahuje. Web všechna API organizuje do skupin a přiřazuje jim kategorie, podle kterých je možno skupiny pak vyhledávat či filtrovat. Vložení vlastního API do takového repozitáře probíhá vyplněním přidávacího formuláře a je přitom spoléháno na spolehlivost člověka, který API přidává. Výsledkem jsou pak často záznamy nepoužitelné, kdy endpoint[7] odkazuje na domovskou stránku provozovatele.

[6] http://google.com

[7] URI, kde dochází ke zpracování požadavku při volání spužby.

Takovéto chyby znesnadňují automatickou identifikaci API, natož pak jeho využití v mashupech, aniž by vývojář nemusel brouzdat skrz webové stránky provozotavele.

Možnost 3 je univerzální řešení pro nalezení API, která v nejhorším případě spadne na stejnou úroveň, jako špatné záznamy v repozitáři. Filtrování API podle kategorií zde sice nenajdeme, zato je k dispozici plně vybavený vyhledávač, na jaký je člověk zvyklý z každodenního života. Pak už stačí jen dohledat i k API dokumentaci z nabízených odkazů ve výsledku dotazu. Nezanedbatelný fakt je, že by vyhledávač při shodě s vyhledávaným dotazem měl ve výsledcích obsáhnout i záznamy z API repozitářů.

Tato úvaha podněcuje myšlenku využití výsledků z vyhledávače na uživatelem specifikované heslo pro kandidátní body crawleru – jakési referenční webové stránky, které crawler prozkoumá do určité hloubky, nalezne API dokumentace a vrátí je uživateli.

2.2 Rešerše

Tato sekce je určena rešerši stávajících řešení a postupů, které souvisí s problémem automatického nalezení a identifikace Web API. Z průzkumu dostupné literatury je patrné, že se jedná o poměrně nový problém a jeho řešení jsou stále pokusy, nikoliv osvědčené postupy a je stále snaha nalézt nějakou schůdnou cestu.

Úroveň i postupy jednotlivých řešení se velmi různí a pouze některé řeší problém v celé šíři. Vynechám přitom manuální práci s repozitáři webových API, která byla dostatečně popsána v předchozí sekci.

2.2.1 Mica: vyhledávací nástroj pro API komponenty a příklady

První nalezenou zmínkou o problému nalezení správného API je z roku 2008 v článku o webovém vyhledávači Mica (Making interfaces clear and accesible), který byl tehdy ještě ve fázi prototypu a jeho úkolem bylo obohatit standardní všeúčelové webové vyhledávače o funkce, které usnadní vyhledávání popisu API a příkladů [11].

V tomto případě se ještě nejednalo o webová API, ale o API obecná, např. metody a použití tříd v Java SDK. Přesto si tvůrci vyhledávače uvědomili, že se API i v tomto případě rychle mění a rozrůstají, že ani zkušení programátoři si nepamatují vše. Ani kdyby byla dokumentace popsaná na jednom místě, vyhledávání po nějakém částečně obecném pojmu by vrátilo stovky jednotlivých dokumentů.

Autoři článku si při pozorování programátorů všimli, že webové vyhledávače jsou nezbytnou součástí vývoje a jejich použití vede ke zrychlení učícího procesu. To zejména z toho důvodu, že záběr výsledků z vyhledávače by byl

Obrázek 2.3: API vyhledávač Mica (screenshot)

Zdroj: [11]

natolik široký, že pojmul i dokumentece, fóra, diskuse, ukázky kódu nebo kompletní zdrojové kódy pri příslušné úkony. Tomu napomáhal i ranking výsledků a jejich seřazení podle relevance ke hledanému výsledku.

Ovšem stinnou stránkou používání vyhledávače bylo, že se mezi výsledky objevovaly i nerelevantní dokumenty. Buď byl obsah dokumentu na jiné úrovni detailu, než programátor potřeboval, anebo vůbec nesouvisel s programováním. Programátoři byli frustrováni tím, že je neuspokojilo prvních pár výsledků vyhledávače a zkoušeli tak jiná a jiná slova. To ale není překvapivé, vzhledem k tomu, že všechny webové stránky jsou porovnávány stejným mechanizmem a tento typ stránek není nijak favorizovaný.

2.2.1.1 Princip vyhledávače Mica

Řešením měl být prototyp vyhledávacího stroje, který měl pomoci programátorům efektivněji nalézt relevantní výsledky. Mica v dokumentech hledá termíny spojené s programováním a pak heuristikou založenou na frekvenci výskytu termínů identifikuje relevantní dokumenty.

Z pozorování programátorů při práci bylo zjištěno, že hlavním zdrojem pro vyhledávání je Google a své odpovědi nacházeli na stránkách oficiálních dokumentací, tutoriály, s ukázkovým kódem a diskuzními fóry s otázkami a odpovědmi. Vyhledávač Mica byl tímto chováním inspirován a k nalezení počáteční sady výsledků na specifikovaný dotaz využívá Google vyhledávací API.

Při práci s Google vyhledávačem přitom Mica analyzoval pouze prvních

10 výsledků. Další už nepřinášely žádné znatelné zlepšení [11]. Výstupem pak byly jak klasický úryvek stránek, na jaký jsme od Google zvyklí dnes, tak další informace ukazoval Java třídy a metody související s vyhledávaným výrazem, jak je vidět v levém části panelu vyhledávače na obrázku 2.3. Slova v dokumentech byly považovány za důležité pouze tehdy, když se shodovaly s některým názvem třídy, metody či interface z knihoven Java SDK. Seznam těchto klíčových slov, na které se vyhledávač zaměřil, byl vytvořen předem z anotací JavaDoc přímo ze zdrojového kódu Javy.

Prototyp Mica byl navržen primárně pro Javu, nicméně byla možná i rozšíření pro ostatní programovací jazyky. Dále Mica nabízí zvýrazňování klíčových slo v ranking dokumentů, to už je ale pro tuto práci není relevantní. Nyní je projekt zřejmě pozastaven a k dispozici není ani prototyp, ani další verze vyhledávače.

2.2.2 Automatická identifikace Web API

Odrazovou studií pro tuto práci se stalo pojednání o problému automatické identifikace Web API, kde byly položeny základy pro vyhledávací stroj pro API dokumentace. Výsledkem byl natrénovaný model, který na zkušebním datasetu dosahoval přesnosti okolo 75% a úplnosti okolo 80% [12].

Práce byla zaměřena na webový crawler, kombinující techniky crowdsourcingu a algoritmů pro strojové učení. Infrastruktura tohoto poloautomatizovaného crawleru se skládá ze dvou částí:

- Webová aplikace pro tvorbu zkušebních dat.
- Klasifikační model.

Webová aplikace sloužila jako hlavní nástroj při tvorbě trénovacího datasetu. Aplikace čerpala API dokumentace z webového repozitáře ProgrammableWeb a nabízela jednoduché uživatelské rozhraní. Do iframe v okně prohlížece byl načten dokument, který byl v ProgrammableWeb označen jako dokumentace webového API, uživatel pak dle svého nejlepšího vědomí potvrdil správné zařezení dokumentu do kategorie API dokumentace, nebo se o dokumentaci nejedná anebo se nedalo přesně určit.

Aplikace byla vpuštěna mezi lidi a výsledkem byla sada 1 553 URL, 43% z celkového obsahu repozitáře. Z toho bylo 40,18% vyhodnoceno jako API dokumentace, 59,82% žádné API nedokumentovalo. Při zpracování se muselo 318 URL přeskočit z důvodu nefunkčnosti služeb na cílové URL nebo nebylo určení příslušnosti jednoznačné.

Stažené dokumenty byla poté zbaveny HTML značek a byl tak získán čistý text dokumentů. K tomu bylo využito HTML knihovny Tidy[8]. Stejně tak došlo k odstranění informace o jednotlivých HTML značkách, které mohly

[8] http://tidy.sourceforge.net/

vést k lepším výsledkům a stejně tak byly odstraněny všechny Javascriptové kódy. Obsah stránek generovaný Javascriptem tak zůstane nedostupný. Obsah byl tokenizován a text převeden na malá písmena.

Vzniklý korpus z předchozího části byl pak využit pro natrénování identifikačního stroje. K tomu bylo použito jak Naive-Bayes (NB) klasifikátoru implementaovaného ve Weka, tak implementaci SVM z balíku libSVM[9].

V době studie neexistoval žádný porovnatelný stroj, který by řešil něco podobného, proto jako výchozí porovnání bylo založeno na heuristice obsahu klíčových slov v dokumentu. Pro experiment byly zvoleny slova: *api, method, operation, input, output, parameter, get, post, put, delete, append, url*, o kterých „je známo, že se objevují často“, a jako optimální prahová hodnota pro počet výskytů byla 3. Pro vyhodnocení byla použita 5-fold křížová validace [12].

Poznámka: Součástí studie byla prezentace, která poukazovala na nedostatky repozitáře ProgrammableWeb tím, že srovnala výsledky z vyhledávače Google na heslo „music api“ a výsledky z repozitáře v kategorii „music“ – srovnatelné množství výsledků z obou vyhledávání API dokumentací bylo a některé ne [13].

2.2.3 Crawler pro webové služby

Možná nejblíže tomuto tématu je Crawler pro webové služby, prezentovaný v článku z roku 2009 [14]. Jednalo se o zaměřený crawler, postavený na open-source crawleru Heritrix[10], a jeho zaměření je směrováno na služby všech typů, hlavně soubory typu WSDL ale i na stránky, které neformálně popisovaly RESTful sužby.

Strategie pro WSDL. Začátek spočíval v inicializaci seedů pro crawler. Ty byly získány poloautomatickým způsobem – některé byly získány z veřejných repozitářů (ProgrammableWeb, XMethods[11]), zbytek utvořily vybrané URL z již nacrawlovaných dokumentů.

Hledání pak bylo koncentrováno na dokumenty popisující služby, což jsou zejména textové dokumenty. Dokumenty ostatních typů jako jsou obrázky, audio a video byly ignorovány. Konkrétně braly v úvahu dokumenty typu HTML, XML a PDF a ostatní textově založené dokumenty, které by mohly popisovat API.

U XML dokumentů bylo během crawlovacícho procesu zjišťováno, zda se jedná o validní WSDL popis a jestli se odkazuje na validní veřejné endpointy. Prvním krokem byla analýza inlinků a outlinků WSDL dokumentů, které mohou vést na další informace související se službou. Crawler při své práci postupuje pouze po outlincích, zjištění inlinků tak může proběhnout až v post-processingové analýze nacrawlovaného grafu.

[9] http://www.csie.ntu.edu.tw/~cjlin/libsvm/
[10] http://crawler.archive.org
[11] http://www.xmethods.net

Tak byly získány možná-s-WSDL-související dokumenty. Konečné rozhodnutí padne ve fázi post-processingu pomocí vektoru termů dokumentů a jejich podobnosti.

Crawl frontier zpočátku bral všechna URL se stejnou prioritou, následně byly podle negativních vlastností některé penalizovány (např. obsahovaly příliš mnoho subdomén, více jak jeden query string, více jak jeden fragment) a ve finále byla některým URL zvýšena priorita, v případě, že obsahovaly „?wsdl“, „ws“, „service“, nebo „api“. Relevantní URl tak byly zpracovány nejdříve.

Strategie pro Web API. Autoři crawleru zkoušeli přístupy dva. Základem byly:

1. Automatická klasifikace, model strojového učení s učitelem.

2. Frekvence termů dokumentu.

K *automatické klasifikaci* využili algoritmu strojového učení, konkrétně algoritmu Suppport Vector Machine (SVM). Jako trénovací sadu si opatřily ručně sesbíraný pozitivní dataset (obsahující pouze API) z repozitáře ProgrammableWeb. Klasifikace dokumentů byla zapojena do samotného procesu crawlování. Jako nástroj pro klasifikaci byl použit nástroj RapidMiner[12].

Druhý postup byl založen na *frekvenci termů* v dokumentu, jenž měl odstranit nešvary automatické SVM klasifikace. Hlavním nedostatkem měl být fakt, že SVM bralo v potaz pouze text a nikterak HTML strukturu a mark-up. Nový postup bral v potaz také URL dokumentu a také syntaktickou stavbu dokumentu, tj. větší počet camel-case slov, než ostatní náhodné stránky (např. `getDocument`), nebo větší počet interních než externích odkazů a ukázky příkladu volání.

Pro rozpoznání typu dokumentu byly vytvořeny tři sledované indikátory.

- Indikátor *Dokumentace* hledal v URL slova „dev“, „doc“, „help“, „wiki“ a další, zajišťoval počet outlinků a slov ve velbloudí notaci

- Indikátor *API* bral v potaz slova obsažená v URL a/nebo samotném obsahu („api“, „developer“, „lib“, „code“, „service“ apod.) a větší počet slov psaných velbloudí notací.

- Indikátor *Webových vlastností* bral v potaz slova a slovní spojení „rest“, „web service“, „api“ v URL a/nebo samotném obsahu dokumentu a hledá větší počet vnitřních odkazů (do stejné domény).

Každý z těchto indikátorů se posuzoval zvlášť a vzinklo z toho skóre dokumentu, které, když dosáhlo stanoveného prahu, tak byl dokument označen jako Web API.

[12] https://rapidminer.com/

Dostupnost sesbíraných dokumentů. Sesbírané dokumenty byly ve finální fázi nahrány do RDF triplestore[13], kde byly zpřístupněné pomocí SPARQL[14] endpointu k dotazování. RDF trojice byly pak využity k zanesení souvislostí mezi dokumenty.

Ačkoliv by výsledky testování crawleru byly pro tuto práci jistě přínosné, nepodařilo se mi je dohledat. V dokumentu je jen podrobnější popis teoretické části crawleru a klasifikační fáze a informace k dotazování přes triplestore [15].

2.2.4 Automatická detekce Web API dokumentací z Webu s modelem Feature LDA

V neposlední řadě zmíním práci [16], kde je pro automatickou detekci Web API dokumentací využíváno modelu feaLDA. FeaLDA (feature LDA) je vylepšený LDA model (latent Dirichlet allocation), který vytváří generativní pravděpodobnostní model korpusu dokumentů [17].

LDA je tzv. *topic model*, který na dokumenty pohlíží jako na směs odlišných témat (topic). Pro klasifikaci využívá třídy, např. „TRAVEL_related“, „API_related“ apod. a každé z těchto témat má pravděpobonost generovat slova ze své třídy. V případě cestování to mohou být „car“, „road“, „adventure“ apod., v případě API „api“, „endpoint“, „delete“, „post“ apod. Pravděpodobnosti náležitosti slova a tématu jsou získány z předem označeného korpusu. Některá slova samozřejmě mohou náležet více tématům se stejnou pravěpodobností.

Zásadní rozdíl mezi LDA a feaLDA tvoří rovnováha mezi slovy dokumentu. Zatímco v LDA jsou všechny slova stejně vyvážená, feaLDA umožňuje určit množinu znaků (features), kterým dává větší váhu. V tomto případě by to byly právě slova jako „api“, „endpoint“, „delete“, „post“ a podobně.

Podrobný popis fungování algoritmů v této práci vynechám (nicméně je možné si je projít v [17] a [16]) a dále se budu věnovat rozsáhlejšímu pojmutí automatického identifikace API dokumentací v článku.

Experimentální postup a vyhodnocení. Postup se skládá ze tří tradičních kroků pro zpracování dokumentu:

1. Získání trénovacích dat. Data byla získána z repozitáře ProgrammableWeb, z odkazů „API Homepage“ v profilové stránce webu repozitáře. Z celkového počtu 1 553 získaných API, registrovaných na ProgrammableWeb, zbylo po vyloučení neplatných URL 1 547, z nichž 622 webových stránek bylo API dokumentací a 925 ne.

2. Preprocessing. V této fázi byly získané dokumenty zbaveny HTML značek pomocí knihovny HTML Tidy. Stejně tak byly přehlíženy tagy `<script>`,

[13] Databáze určená pro RDF data. Data uložená ve formě předmět–predikát–objekt.
[14] Jazyk pro dotazování nad RDF daty.

protože „nemají žádný význam pro klasifikaci“ [16]. V dalším kroku byly odstraněny nealfanumerické řetězce a všechny znaky byly převedeny na malá písmena. Byla odstraněna stopslova a výsledek byl zpracován Porter stemming algoritmem[15].

3. Klasifikace dokumentů.

Výsledky feaLDA byly testovány na zkušební sadě dokumentů a porovnány s algoritmy Naive Bayes, maximální entropií (MaxEnt) a SVM. FeaLDA bylo testováno na rozdílném počtu témat $T \in \{1, 2, 3..20\}$.

2.2.5 Google CSE pro obohacení informací nalezených entit

Na přímo k API a dokumentacím, ale k ukázce využití Google Custom Search Engine se vztahoval přístup k obohacení zpravodajského videa o informace nalezené na webu. Projekt pro LinkedTV[16], který měl za cíl automaticky generovat obsah pro tzv. *second screen*[17] na základě informací v textovém popisu videa.

Technikou NER (Named Entity Recognition – rozpoznávání pojmenovaných entit) byly získány klíčové entity, které byly předmětem zprávy a zbývalo dodělat rozšíření této techniky o získání relevantních informací. Autoři k tomuto využívali vyhledávače Google CSE, upravený specifické potřeby aplikace. Pro nadpisové informace pak tvořila aplikace dotazy ve formě „The *entity* case“. Výsledkem z CSE byl seznam URL, na kterých bylo možné najít další popisy o zpravodajské události.

Využití Google CSE se v předběžných testech osvědčila a pozorovateli byly tak nabídnuty další relevantní informace k právě prohlíženému videu, stejně tak, jako informace, které nejsou explicitně zmíněné v popisu videa, ale které v kontextu s videem souvisí. A kontext byl získán právě použitím vyhledávače [18].

2.2.6 Shrnutí poznatků z rešeršního průzkumu

Poznatky bych kategorizoval podle fází finálního crawleru – *Automatické získání* a *Automatická identifikace* API dokumentací.

2.2.6.1 Automatické získání

Automatické získání API dokumentace souvisí s problémem nalezení (sekce 2.1.3.4). Z průzkumu je patrné, že tato část byla většinou ingorována a za-

[15]Stemming je longvistický morfologický proces slova, při kterém se převede do kořenového tvaru (stem)

[16]https://www.linktv.org/

[17]Second screen, někdy také „2nd screen“, je označení pro propojení mobilních zařízení a rozšíření zážitků ze sledování TV, časopisů, novin atd.

Tabulka 2.1: Srovnání výsledků přístupů z rešeršního průzkumu

	Přesnost [%]	
Metoda	Min	Max
Naive Bayes	78,6	78,9
SVM	79,0	79,0
feaLDA	75,9	80,5
Frekv. slov	70,2	70,2

měřeny byly pouze na poznání Web API. Jediný implmentovaný crawler byl v 2.2.3, který sice necrawloval celý web, nicméně byl značně zaměřen na repozitář ProgrammableWeb a jehož autoři si uvědomili, že většina vložených URL není ve webovém grafu až tak daleko od samotné API dokumentace, ale ukazují třeba jen na domovskou stránku poskytovatele.

Častým zdrojem, ať už použitým nebo zmíněným, se stal vyhledávač/vyhledávače založené na Google. Všeúčelový klasický Google vyhledávač je jistě dobře obecně použitelný, často byl ale kritizován za svojí obecnost. A nalezení specifického druhu dokumentu, jakým je Web API dokumentace, se příliš nehodí. Naopak se nabízí využít jeho služby Google CSE.

Google CSE umožní vytvořit vyhledávač, který bude používat stejný index jako klasický Google vyhledávač, ale je možné jej customizovat podle potřeb. Jeho počínání v hledání relevantních informací o videozáznamu (2.2.5) bylo značně působivé a pokusím se tak vytvořit podobný, jen jinak upravený vyhledávač pro účel nalezení relvenantních záznamů o specifickém druhu webových API dokumentací.

2.2.6.2 Automatická identifikace

Přístupy pro klasifikaci webových API se objevovaly dva: *strojové učení* a *heuristika založená na frekvenci klíčových slov v dokumentu*. Algoritmům pro strojové učení dominovala metoda učení s učitelem a vždy docházelo k vytvoření trénovacích dat. Nejčastěji používanými algoritmy byly Naive-Bayes a Support vector machine. Heuristiky se v teorii lišili od sebe daleko více, ale vždy se jednalo o postupy vytvořené na *identifikaci klíčových slov*, specifických pro API dokumentace obsažených v samotném obsahu nebo URL stránky, případně další drobnosti jako počítání s různými druhy linků apod. Výsledky a porovnání metod jsou shromážděny v tabluce 2.1.

Kroky pro identifikaci se vždy skládaly z

1. Preprocessing nashromážděného datasetu.

   Trénovací data ve většině případů pocházela z veřejného repozitáře webových API ProgrammableWeb, odkud byla manuálně/poloautomaticky získána a člověkam ohodnocena.

Z HTML stránek byly odstraněny HTML značky a to včetně Javascriptu. Ztratila se tak informace nejen o struktuře dokumentu, která by mohla vést k lepším výsledkům, ale zanedbáním Javascriptu se přišlo i o obsah, jenž byl skriptem generován.

2. Natrénování algoritmů/Implementace heuristik.

3. Vyhodnocení.

Prezentované výsledky přesnosti algoritmů a postupů se vztahovaly vždy na stejný dataset, jaký byl využit k natrénovaní, případně na data, který bezprostředně souvisela z obsahem repozitáře ProgrammableWeb. Je otázkou, jak se tyto přístupy osvědčí na reálných datech.

2.3 Strojové učení

Strojové učení je vědecká disciplína, zabývající se zkoumáním a tvorbou algoritmů, které se dokáží *učit* ze vstupních dat.

Učení v tomto případě je jakýmsi nástrojem k úpravě algoritmu za běhu takovým způsobem, aby v budoucnu byl schopen z nově naučených poznatků vyvodit nějaké závěry a ty pak případně aplikovat na nová data.

Jako člověk a obdobně jiní myslící živočichové ukládá tyto poznatky do paměti podle svého uvážení a vhodné kategorizace. Této v paměti vytvořené struktuře se říká *model*. Tento model je pak v pozdější fázi, když je učící fáze na zkušebních příkladech ukončena, použit v aplikování těchto vědomostí na nová data. To jest jednou ze základních součásti data miningu a základní myšlenkou je snaha o nalezení strukturálních vzorů (ang. *structural patterns*) ve vstupních datech.

2.3.1 Demonstrace na ukázkovém datasetu

Pro lepší představu o strojovém učení uvedu jednoduchý příklad s počasím, na kterém budu demonstrovat požadavky a aplikaci strojového učení.

V tabluce 2.2 jsou znázorněna vstupní data. Data prezentují rozhodnutí na základě nějakých zkušeností z praxe, zda je vhodné hrát venku nějakou dále nespecifikovanou hru za daných venkovních podmínek jako je počasí, teplota, vlhkost a přítomnost větru.

V globálním pohledu se data v tabulce dají označit za *dataset*, její jednotlivé řádky pak prezentují *instance* tohoto datasetu. Tyto instance jsou reprezentovány množinou *atributů* a jejich hodnot. V tomto případě jsou tedy atributy čtyři: *počasí*, *teplota*, *vlhkost* a *vítr*. Výsledná hodnota dle těchto vstupních hodnot je ve sloupci *Hrát si?*.

Atribut počasí nabývá hodnot „slunečno“, „zataženo“ a „deštivo“, atribut teplota „teplo“, „střední“, „chladno“, vlhkost může být „vysoká“ nebo „v normálu“, vítr může být, pak „ano“, nebo „ne“. Atribut výstupní hodnoty

Tabulka 2.2: Ukázkový dataset

Počasí	Teplota	Vlhkost	Vítr	Hrát si?
Slunečno	teplo	vysoká	ne	ne
Slunečno	teplo	vysoká	ano	ne
Zataženo	teplo	vysoká	ne	ano
Deštivo	střední	vysoká	ne	ano
Deštivo	chladno	v normálu	ne	ano
Deštivo	střední	v normálu	ano	ne
Zataženo	chladno	v normálu	ano	ano
Slunečno	střední	vysoká	ne	ne
Slunečno	chladno	v normálu	ne	ano

Tabulka byla autorem vytvořena pro účely této práce na základě ukázkových datasetů dodaných se software Weka [19].

nabývá hodnot „ano“ a „ne“. To dává dohromady $3 \cdot 3 \cdot 2 \cdot 2 = 36$ možných kombinací, z nichž v tabulce je pouze 9.

V tomto příkladě nabývají atributy nabávají hodnot z množiny všech hodnot. Tomuto typu atributů se říká nominální. Opakem tomu jsou numerické atributy, ty v tomto příkladu ale neuvažuji.

Nyní, co by se dalo z té tabulky odvodit – podle kterých znaků je možné soudit nejlepší výstupní hodnotu. Některá tyto pravidla by mohla například být:

KP1: *Pokud* je slunečno, teplo a vysoká vlhkost, *potom* se hrát nebude.
KP2: *Pokud* je deštivo a není vítr, *potom* se hrát bude.
KP3: *Pokud* je teplota střední a vlhkost v normálu, *potom* se hrát bude.
KP4: *Pokud* je zataženo, *potom* se hrát bude.
KP5: *Jinak* se hrát bude.

Tento seznam pravidel je aplikován dle zapsaného pořadí, tj. prvně je aplikováno první pravidlo, pak druhé atd. Takto sepsanému seznamu pravidel se říká rozhodovací seznam *decision list* a pravidla v něm obsažená jsou klasifikační pravidla (ang. *classification rules*).

Pořadí těchto pravidel musí být takto striktně dodržováno a je možné, že aplikovány samy o sobě by mohly vést k chybně klasifikovaným výsledkům.

Stejně tak je ekvivalentní vytvoření jakýchkoliv pravidel přímo z tabulky [20]. Takto vzinklá pravidla se nazývají asociační pravidla (ang. *association rules*).

AP1: *Pokud* je chladno, *potom* je vlhkost v normálu.
AP2: *Pokud* je teplota střední a vlhkost vysoká, *potom* není vítr.
AP3: *Pokud* se hraje hra a není vítr, *potom* je slunečno a vysoká vlhkost.
AP4: *Pokud* je zataženo, *potom* je hraje.

A to jsou pouze některá z asociačních pravidel, která jsou navíc 100% korektní. Dalo by se jich vytvořit daleko více a stále s dostatečně vysokou pravděpodobností. Na rozdíl od klasifikačních pravidel dokáží asociační pravidla „předpovídat“ ostatní atributy, nejen jednu specifickou třídu. A dokonce se nemusí jednat o pouze jeden atribut, pravidlo **AP3:** má na pravé straně atributy dva a dokáže tedy předpovědět oba.

Na podobném principu, ale za použití jiných modelů a struktur, je založena celá řada těchto ML (machine learning) algoritmů. Některé využívají sady pravidel a statistických znalostí trénovací množiny, jiné stromové struktury, které se větví na základě entropie hodnot či jiných požadavků.

V dnešní době existují programy, které umožňují použití nebo vyzkoušení těchto technik. Mezi nejznámější a nejrozšířenější patří Weka a RapidMiner.

Kapitola 3

Návrh

V této kapitole popíši požadavky na zaměřený crawler a jeho návrh. Jakým způsobem je v různých detailech navržen a jak spolu všechny doposud popsané části interagují.

3.1 Požadavky

Obecné požadavky byly zmíněné již v úvodu práce. Nyní je formálně popíši a specifikuji, aby byly jasné možnosti crawleru.

3.1.1 Funkční požadavky

1. Nalezení web API dokumentací – crawler bude vyhledávat dokumenty odpovídající vstupnímu vyhledávacímu heslu či frázi. Nejsou brány v úvahu všechny na Internetu dostupné dokumenty, ale pouze n nejvíce relevantních výsledků z vyhledávače.

2. Konfigurace crawleru – vlastnosti, chování crawleru a jeho komponent je možné uživatelem konfigurovat. Konfigurovatelné parametry jsou:

   a) Parametry pro zdroj kandidátních URL:

      i. Identifikátor vyhledávače. Určuje, jaký vyhledávač bude použit.

      ii. Aplikační klíč – nutný pro využití služeb Google a tedy i vyhledávače.

   b) Parametry pro crawlování:

      i. Hloubka crawlování (viz 4.2).

      ii. Filtrování podle URI.

      iii. Stažený počet dokumentů v jenom kole.

   c) Parametry pro strojové učení a klasifikaci:

i. Algoritmus strojového učení.
ii. Konfigurační řetězec algoritmu.
iii. Zdroj trénovacího datasetu v syrovém nebo ARFF formátu.

3. Poskytování informací – základní informace o nalezených dokumentech jsou uživateli dostupné, včetně informace o tom, zda každý dokument je, nebo neni dokumentací webového API a s jakou pravděpodobností.

3.1.2 Nefunkční požadavky

1. Crawler bude napsán v jazyce Java.
2. Sběr dokumentů bude postaven na open-source crawleru Apache Nutch.
3. Indexace nashromážděných dat bude uložena v Apache Solr.
4. Pro klasifikaci budou využity algoritmy implementované v software Weka.

3.2 Koncept

V této sekci popíši high-level koncept crawleru, který pomůže přiblížit celkové fungování a kooperaci jednotlivých komponent.

Tento pohled jsem je zachycen na obrázku 3.1. Schéma obsahuje dvojí typy kroků:

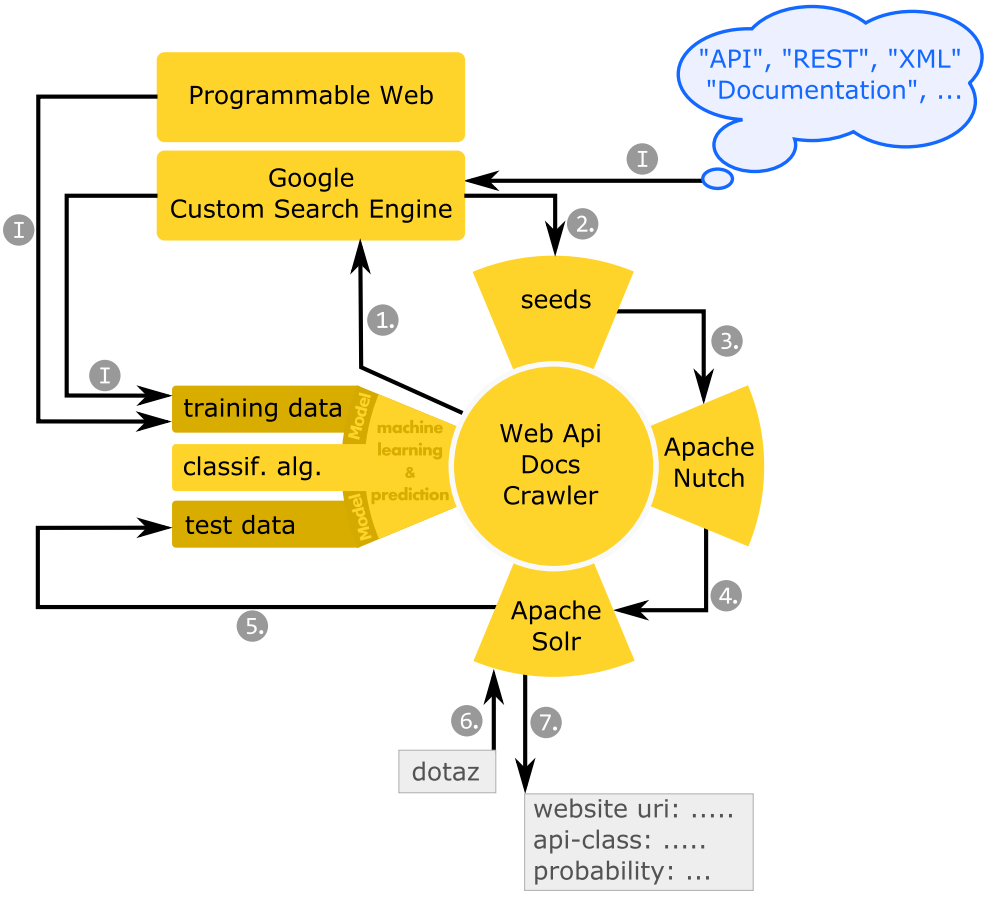
- kroky označené „I“ jsou inicializační a jsou provedeny manuálně před jakýmkoliv spuštěním crawleru, nicméně jedná se o potřebné prerekvizity.
- kroky označené čísly 1 – 7 jsou pro samotné crawlování, klasifikaci a dotazování na výsledky.

3.3 Inicializace

V inicializační části je potřeba vytvořit základ pro algoritmy strojového učení, tedy zkušební sadu dokumentů a zdroj pro kandidátní dokumenty na Webu. Jedná se tedy o

1. Google Custom Search Engine a
2. trénovací dataset.

Trénovací dataset se bude skládat z dokumentů, které jsou Web API a z dokumentů, které API nejsou, ale přesto se s nimi může crawlel setkat. Proto je výhodné, i vzhledem k tvorbě CSE, prvně získat Web API dokumentace,



Obrázek 3.1: Koncept crawleru

z těch získat představu o obsahu slov v tomto typu dokumentů. Ty následně použít na vytvoření CSE a nakonec použít nový CSE na nalezení dokumentů určité kategorie a z těchto výsledků nacházet dokumenty, které API nejsou.

3.3.1 Trénovací dataset

Trénovací dataset bude základem pro algoritmy strojového učení a bude obsahovat Web API dokumentace, stejně jako dokumenty ostatních typů. Ke sběru API dokumentací použiji největší repozitář webových API ProgrammableWeb [1].

Vyjdu ze seznamu API podle popularity využití. To pomůže zaměřit se na mezi lidmi populární API, se kterými se dobře pracuje, tj. jsou přehledné nebo jinak potřebné.

Až bude vytvořený CSE vyhledávač, využiji jej a kategorie API z ProgrammableWeb k nalezení relevantních výsledků na Internetu. Tím získám tipy

na dokumenty, se kterými se crawler může setkat. Ty se stanou zdrojem pro ne-API dokumenty.

3.4 Sběr dat

Za předpokladu, že již máme kompletní trénovací dataset a vyhledávač, může se crawler víceméně pustit do práce. Tu jsem rozdělil do tří fází:

1. Získání seedů z vyhledávače.
2. Sběr dokumentů.
3. Klasifikace dokumentů do tříd *API dokumentace* a *ostatní*.

3.4.1 Získání seedů z vyhledávače

Předpoklady:

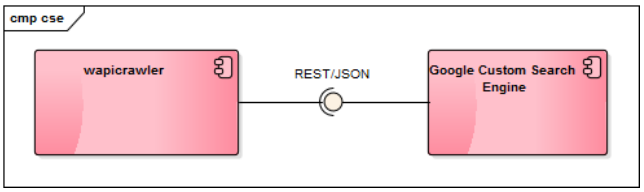
1. Je k dispozici Google CSE vyhledávač s identifikátorem *cx*.
2. Je k dispozici identifikátor *cx*.
3. Je k dispozici Google API klíč *key*.
4. Požadovná kategorie API, reprezentovaná dotazem *q*.

Tato fáze má za úkol nashromáždit kandidátní URL, které budou sloužit jako seedy pro crawlovací část.

Crawler vyšle HTTP požadavek na endpoint Google CSE API s obsaženými parametry 1 – 3 a dotazem *q*, obsahujícím požadované heslo. Vyhledávač zpracuje odpověď, detekuje případné chyby a vrátí odpověď ve formátu JSON.

Crawler odpověď zpracuje – buď došlo k chybě (neplatný klíč aplikace, vyčerpán limit požadavků, ...) nebo je počet nalezených kandidátů nulový – v tom případě crawler nemá jak dál pokračovat a končí. V opačném případě načte (příp. se prolistuje skrz další požadavky a odpovědi) požadované množství k_n kandidátních URL.

Ty uloží v souboru *seeds.txt* (jedna URL na řádek). Tento soubor bude vstupem pro další fázi.



Obrázek 3.2: Návrhový model: získání seedů

3.4.2 Sběr dokumentů

Předpoklady:

1. Je k dispozici soubor *seeds.txt*, obsahující k_n kandidátních URL.
2. Je k dispozici Apache Nutch 1.9.
3. Je k dispozici běžící instance Apache Solr a jeho URL.

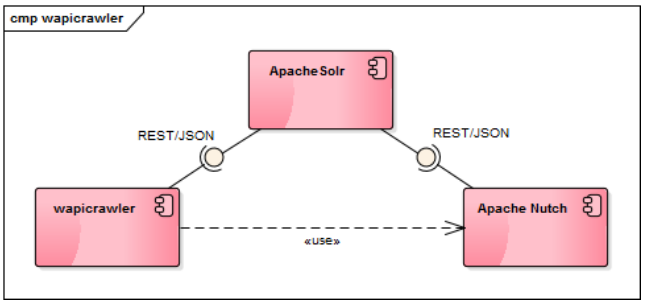
Tato fáze zajistí nashromáždění dokumentů v okolí daném URL kandidáty a dalšími parametry crawlování.

Nejprve dojde ke konfiguraci Apache Nutch podle parametrů a dalších vstupních omezení. Pak je spuštěn crawlovací proces Apache Nutch s využitím jeho spouštěcích skriptů.

První část je zaměřena čistě na sběr webových dokumentů a je celá v režii crawleru Apache Nutch. Tomu bude injektován seznam URL kandidátů jakožto seedů a určí se výstupní adresář, kam se mají nashromáždit nasbírané dokumenty. Až Nutch skončí, budou v cílovém adresáři nashromážděny webové dokumenty v binární podobě.

Druhou části je indexace. Tedy proces převedení dokumentů ve formátu, se kterým pracuje Nutch, do vyhledávací platformy Apache Solr. Od tohoto momentu jsou všechny nasbírané dokumenty dostupné v Solr a je možné se na ně dotazovat podle 4.1.3.3.

A protože jsou fáze na sobě nezávislé a dochází k indexaci všech dokumentů nehledě na jejich typ, musí být kam pak označit typ dokumentu. K tomu dobře poslouží úprava schématu Solr přidáním dalších polí pro třídu dokumentu, tedy 0/1 (zda je či není API) a pravděpodobnost $[0, 1]$, s jakou do dané třídy patří.



Obrázek 3.3: Návrhový model: sběru dokumentů

3.4.3 Klasifikace dokumentů

Předpoklady:

1. Je k dispozici běžící instance Apache Solr a jeho URL.

V této fázi předpokládám, že již je v Solr zaindexovaná řada dokumentů, ale zatím není známa jejich příslušnost do kýžených kategorií. A tento nedostatek právě řeší tato fáze. Využívat k tomu bude algoritmy a pomocné třídy pro strojové učení, většinou dostupné se softwarem Weka.

Nejprve je potřeba si uvědomit, jak fungují algoritmy strojového učení. To je náležitě popsáno v sekci 2.3. Odtud víme, že dokument, jakožto instance, je nějak reprezentován. To budu označovat za *Datový model instance*. Pokaždé se bude jednat o nějaký seznam atributů, jde ale o to, jak se tato množina sestrojí. Pomocí Weka lze datový model instance vytvořit dvěma způsoby:

DM1 Klasickou metodou zvolení m atributů.

Každý atribut má příslušnou doménovou množinu hodnot, kterých může nabývat. Když je číselný, tak čísel různých typu, nebo výčtový, a pak ze specifikované množiny.

DM2 Zvolení množiny atributů přesně odpovídající obsahu dokumentu.

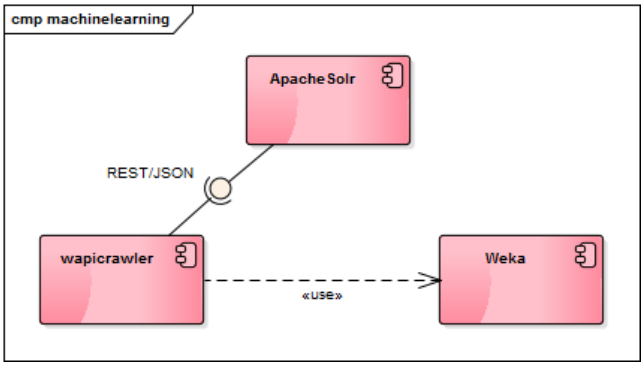
Software Weka umožňuje reprezentaci dokumentu vytvořit tak, aby co nejvíce odpovídala obsahu dokumentu. Toho je docíleno použitím filtru, konkrétně se jedná o `StringToWordVector`. Název filtru napovídá, že nyní nebude mít dokument d atributů m, ale (většinou) o mnoho více – v nejhorším případě $|d|$, tedy počet slov v dokumentu. To se často používá při zpracování textů psaných v přirozeném jazyce, tzv. NLP (natural language processing).

> *„Experience shows that no single machine learning scheme is appropriate to all data mining problems. The universal learner is an idealistic fantasy. [...] real datasets vary, and to obtain accurate models the bias of the learning algorithm must match the structure of the domain. Data mining is an experimental science.“* [20]

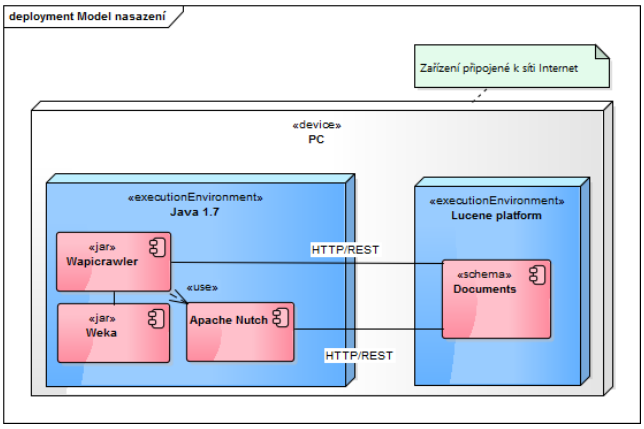
Protože data mining je experimentální věda [20], bude crawler umožňovat zpracování a kategorizaci dokumentů na základě obou zmíněných datových modelů instancí. Stejně tak bude podporovat několik algoritmů pro strojové učení pro klasifikaci.

Použití obou datových modelů instancí je v tomto případě různé. Zatímco při **DM1** je potřeba analyzovat dokumenty popisující webové API a určit vhodnou množinu popisných znaků, využití **DM2** si vyžaduje postupné zpracování dokumentů. StringtoWordsVector vytvoří množinu rysů pro každý dataset jinou. To je ale neslučitelné s fungováním algoritmů strojového učení implementovaných ve Weka, protože rysy trénovacího datasetu musí mít stejný typ a

stejnou délku jako v testovacím datasetu. Naštěstí i to se dá ve Weka vyřešit. Více k tomuto, a jak jsem postupoval při nalezení rysů pro webové API, je k nalezení v kapitole Realizace.



Obrázek 3.4: Návrhový model: klasifikace



Obrázek 3.5: Model nasazení

Každý z vyjmenovaných algoritmů bude možné konfigurovat dle jeho parametrů. Zvolil jsem jednoduchý, pro uživatele příjemný, způsob, který toto umožní a tím je konfigurační řetězec. Ten je možné buď podle základní syntaxe sestrojit, nebo přímo zkopírovat z nástroje Weka. Při manuálním testování algoritmů, např. ve Weka Explorer, je algoritmus parametrizován pomocí uživatelské interakce skrze přehledná dialogová okna s potřebnými popisky. Výsledkem je pak řetězec, který je uživateli hned přístupný. Zkušený uživatel tak může konfigurovat algoritmus přímo zde, což může být rychlejší. A přesně tento způsob pro zkušené uživatele bude implementován též v crawleru.

3.5 Dotazování

Všechny dokumenty jsou dostupné v databázi Solr již po crawlovací fázi, nicméně až v tuto chvíli je každý dokument označen příslušnou 0|1 třídou do které patří. V obrázku 3.1 tomu odpovídají kroky 6 a 7.

Ačkoliv crawler nebude nabízet rozhraní pro přístup k API dokumentacím, bude možné se na ně dotazovat skrze rozhraní Solr za využití klasických dotazovacích prostředů popsaných v 4.1.3.3.

Kupříkladu pro zajištění veškerých obsahů API dokumentů a jejich pravděpodobnost příslušnosti jako API, jestliže pole `api-class` nabývá hodnot 0 resp. 1, pokud není, resp. je API dokumentace a pole `api-prob` obsahuje příslušnost k třídě. Dotaz na Solr endpoint by vypadal takto (před URL kódováním): `http://localhost:8983/solr/select?q=api-class:1&fl=document,api-prob`

Kapitola 4

Realizace

Kapitola v první částí popisuje, jaké technologie, programy a knihovny crawler používá a proč byly zvoleny. Zbytek je věnován popisu konkrétním implementačním detailům, které popisují, jak bylo postupováno při realizaci, jaká jsou nastavení a propojení komponent apod.

4.1 Použité technologie

4.1.1 Java

Jako programovací jazyk jsem zvolil jazyk Java (konkrétně verzi Java SE 7), se kterým se budu moci soustředit na vývoj a zpřehlednění celkového kódu za využití OOP. Zároveň jsou v něm napsány využívané knihovny, takže je bude možné je bez přepisování použít.

Java je programovací jazyk a výpočetní platforma poprvé vydaná společností Sun Microsystems v roce 1995 a v dnešní době je k nalezení téměř na každém kroku. Je využita všude v přenosných počítačích, datacentrech, herních konzolích, vědeckých superpočítačích, mobilních telefonech i internetu, velká část webových stránek nebude plně fungovat, pokud na zařízení bude Java chybět.

Primární cíle jazyka podle [21] jsou

- Jednoduchost, objektová orientace a povědomost.

  Základní charakteristikou programovacícho jazyka Java je jednoduchost. Tak mohou být programy psány bez rozsáhlého programovacího školení. To vede k tomu, že programátoři mohou být produktivní od samého začátku.

  Celý jazyk je postaven na objektovém paradigmatu. V době před principy OOP se kód stával natolik neudržitelný, funkce systémů se stávaly čím dál více složitější a rozmach architektury klient-server a výměny zpráv přímo volal po oddělení částí systému v samostatně fungující části.

Ačkoliv bylo C++ zavrhnuto jako implementační jazyk pro Javu, styl programování je velmi podobný, jen byly odstraněny některé zbytečné nešvary C++, což vedlo ke vzniku Javy jako jazyka, který si z C++ odnáší rysy objektově orientované rysy a využití tříd, má podobnou syntax a je jednodušší.

- Robustnost a bezpečnost.

  Java je navržena pro tvorbu velmi spolehlivého software. Pro to nabízí řadu nástrojů, které umožňují kód kontrolovat v několika stádiích vývoje.

  Přispívá k tomu i sjednocení ve vytváření objektů pomocí klíčového slova *new*, není zde žádná ukazatelová aritmetika jako v C++, a přibyl automatický Garbage collector, který se sám staral o uvolnění nepotřebných referencí do paměti, což pomohlo drasticky zredukovat chyby v produkčním software.

  Technologie Java má v sobě zakomponované vlastnosti znemožňující nabourání aplikace zvenčí nebo jich využít k vytvoření a šíření virů.

- Nezávislost na architektuře a portabilita.

  Návrháři jazyka mysleli i na bezpečné nasazení programů v heterogenních síťových prostředích, kde budou moci programy fungovat nezávisle na hardwarové architektuře, stejně jako by měly být schopné běžet na různých operačních systémech a komunikovat s programy pomocí programovacích rozhraní. K tomuto řešení byl navržen Java Compiler, který generoval tzv. bajtkód (ang. *bytecode*) – na architektuře nezávislý zkompilovaný kód, který bylo možné spustit na více hardwarových a softwarových platformách.

  Portabilitě napomáhá i to, že jsou striktně definované základní typy. Veškeré chování je tak sjednoceno napříč různými prostředími.

  Architektonicky nezávislé a portabilní prostředí, ve kterém programy spouštěny se nazývá *Java virtual machine* (JVM). Jedná se o abstraktní virtuální stroj, kde mohou kompilátory spouštět zkompilovaný kód. Implementace JVM jsou specifické pro různé HW a SW platformy, ale využívá k tomu specifikaci rozhraní POSIX[18].

- Vysoký výkon.

  Java svého výkonu dosahuje zavedením takového schématu, aby mohl interpreter věnovat veškerý svůj procesorový čas na běh programu bez potřeby kontrolovat run-time prostředí. Přispívá tomu i Garbage collector, který běží na pozadí jako proces s nízkou prioritou.

- Interpretovanost, využití vláken a dynamika.

[18] Sada specifikací IEEE pro jednotné aplikační rozhraní.

Java Intepreter může vykonávat bajtkód přímo na zařízení, na kterém byl Interpreter a JVM naportovány.

Vícevláknovost je požadavek na moderní aplikace pro maximální využití procesoru. K tomu jsou k dispozici high-level knihovny, jejichž funkce byly napsány tak, aby byly thread-safe a nedocházelo k tzv. race-conditions (k nechtěnému přepisování sdílených prostředků jednotlivými vlákny).

Dynamika je zajištěna ve fázi linkováni, kdy jsou přilinkovány pouze potřebné třídy, nebo na požadavek v případě knihoven. Linkování může probíhat i mezi vzdálenými systémy.

4.1.2 Apache Nutch

Jako součást, která bude mít na starosti crawlování jsem vyšel ze zadáním doporučeného crawleru Nutch [22] od nadace Apache Software Fundation.

Apache Nutch je vysoce škálovatelný a rozšiřitelný open-source webový crawler, napsán v jazyce Java. Vyšel z projektu Apache Lucene[19] a nyní je rozvětven do dvou částí:

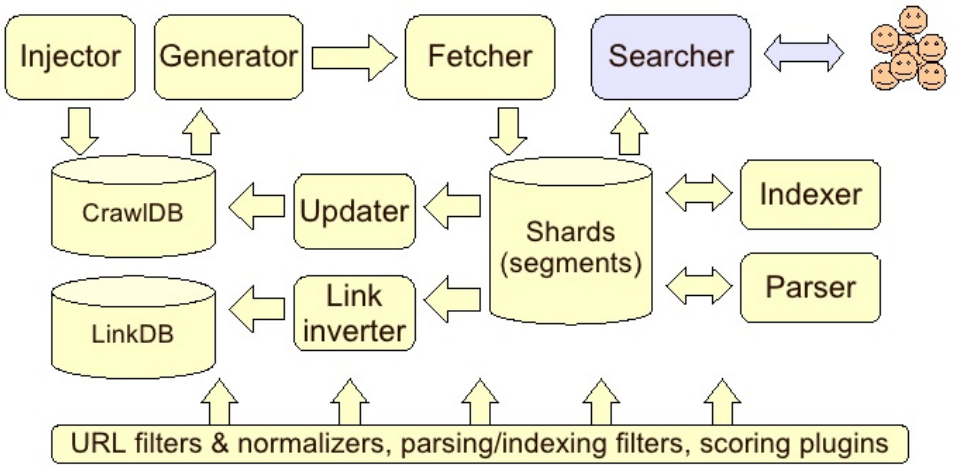
- Nutch 1.x. Původní projekt, lety prověřený crawler umožňující dostatečně přesnou konfiguraci. Je schopen běhu na frameworku Apache Hadoop[20], nicméně samozřejmě funguje i na jednom počítači.
- Nutch 2.x. Alternativní větev, která vychází z verze 1.x, ale kromě drobných změn v kódu a vnitřním API se liší zejména v abstrakci úložiště, které v této verzi využívá software Apache Gora[21] pro zpracování objektu do persistetního stavu. Je pak možné jako databázi využít řadu NoSQL řešení.

Podle [23], pokud crawleru Nutch člověk plně rozumí, je schopen si vytvořit vlastní vyhledávač, jakým je např. Google. Nutch dělá vše od crawlování, parsování (zajišťuje Apache Tika[22]) a indexování (v základním nastavení zajišťuje Apache Solr[23]). Je rozšiřitelný formou pluginů, škálovatelný a využívá vlastní hodnotící filtr (ang. *scoring filter*), který zajišťuje Java třída ScoringFilter, která se využívá k honocení priority nalezených výsledků.

4.1.2.1 Anatomie Nutch

Po úspěšném nainstalování crawleru Nutch jsou k dispozici následující adresáře:

[19] http://lucene.apache.org
[20] http://hadoop.apache.org
[21] http://gora.apache.org
[22] http://tika.apache.org
[23] http://lucene.apache.org/solr



Obrázek 4.1: Apache Nutch: stavební bloky

Zdroj: [24]

- Adresář `build` obsahuje všechny potřebné závislosti ve formátu JAR, které si Apache Nutch stáhl při fázi sestavení.
- Adresář `conf` obsahuje všechny konfigurační soubory, kterými se dá crawler crawler nastavit.
- Adresář `docs` obsahuje dokumentaci k celému projektu, včetně všech aktuálně stažených pluginů.
- Adresář `ivy` obsahuje potřebné konfigurační soubory, pokud chceme např. přidat novou závislost či zakomponovat plugin.
- Adresář `runtime` obsahuje všechny potřebné skripty pro úspěšné spuštění crawleru.
- Adresář `src` obsahuje všechny stažené Java třídy, se kterými byl Apache Nutch zkompilován.

Na obrázku 4.1 jsou vidět základní bloky a celkový náhled na architekturu crawleru Apache Nutch. Jejich funkce jsou:

- `CrawlDB` uchovává informace o všech známých URL. To zahrnuje
  - rozvrh zajištění stránky (ang. *fetch schedule*)

  - stav zajištění stránky (ang. *fetch status*),
  - podpis stránky (ang. *page signature*) a
  - metadata.

- `LinkDB`. Pro každou cílovou URL se zde uchovávají informace o odkazech ve formě seznamu zdrojových URL a cílové adresy odkazu.
- `Shards`, neboli „segmenty“ jsou úložištěm pro
  - syrový obsah stránky (ang. *raw page content*),
  - parsovaný obsah stránky, vč. metadat a nalezených outlinků[24], a
  - čistý text stránky (ang. *plain text*).

Pluginy. Zmíněná flexbilita crawleru spočívá v jeho založení na pluginech, které se starají o veškeré parsování, indexování a vyhledávání, které Nutch v průběhu provádí [25].

Každý plugin rozšiřuje jeden nebo více rozšiřitelných bodů, tzv. *extension-points*. Extension-points jsou například IndexWriter (pro zapisování dat do úložiště, může být Solr, CSV, ...), Parser (pluginy zodpovědné za extrakci dat z URL), Protocol (pro získání dokumentů přes odlišné protokoly – ftp, http, https, ...) a další.

K tomu, jaké pluginy Nutch aktuální používá slouží v konfiguračním nastavení položka `plugin.includes`. Při tvorbě vlastního pluginu je nutné též dodat příslušné direktivy do adresáře `plugin`, souboru `build.xml`. O konfiguraci crawleru pojednává následující sekce 4.1.2.2.

4.1.2.2 Konfigurace

Samotný crawler se dá konfigurovat pomocí předpřipravených konfiguračních souborů ve formátu XML, případně pouhý text. Souborů je celá řada, nejdůležitější jsou:

```
apache-nutch-1.9-src
└── runtime
    └── local
        └── config
            ├── nutch-default.xml ..... Implicitní konfigurace
            ├── nutch-site.xml ..... Explicitní konfigurace
            ├── schema.xml ..... Schéma pro indexaci
            └── regex-urlfilter.txt ..... Filtrace URL reg. výrazy
```

[24] Outlink-em dokumentu d se rozumí odkaz směřující z dokumentu d do libovolného dokumentu $x \neq d$

Soubor `nutch-default.xml` obsahuje implicitní nastavení crawleru, se kterým by se *nemělo* vůbec manipulovat. Veškeré potřebné změny se zanáší do `nutch-site.xml`, které přepíší ty z `nutch-default`. Explicitní konfigurace `nutch-site.xml` má stejnou strukturu jako implicitní, tj. jak je vidět na ukázce 4.1.

Jedinou nutnou podmínkou pro úspěšné spuštění crawleru je identifikace crawleru. To se udělá nastavením vlastnosti `http.agent.name` v souboru `nutch-site.xml` na požadovaný název.

```
<configuration>
    <property>
      <name>http.timeout</name>
      <value>10000</value>
      <description>The default network timeout, in milliseconds.
      </description>
    </property>
    <property>
      <name>store.ip.address</name>
      <value>false</value>
      <description>Enables us to capture the specific IP address
      (InetSocketAddress) of the host which we connect to via
      the given protocol. Currently supported is protocol-ftp and
      http.
      </description>
    </property>
     ...
</configuration>
```

Výpis 4.1: Ukázka konfiguračního souboru nutch-site.xml

Nutch umožňuje filtrovat URL na základě regulárních výrazů. K tomu slouží `regex-urlfilter.txt`, který obsahuje na každém řádku (kromě komentářů začínajících `#`) jeden regulární výraz, který má na začátku ještě speciální znaménko – `+` nebo `-`, podle toho, jestli se má URL akceptovat či zahodit.

Výrazy jsou čteny odshora dolů a první regulární výraz v pořadí, který dokáže akceptovat danou URL, se tak aplikuje. Záleží tak na pořadí výrazů.

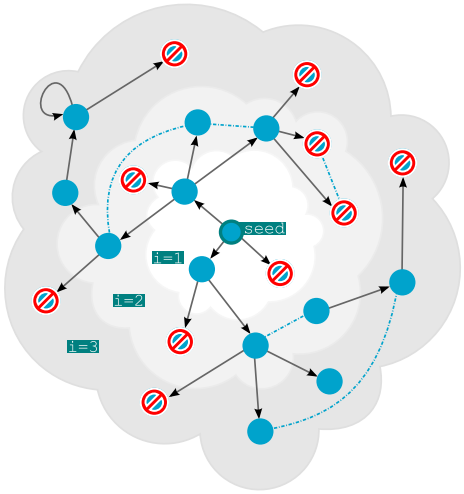
Například záznam
`-^(file|ftp|mailto):`
zahodí všechna URL začínající `file:`, `ftp:`, nebo `mailto:`.

Soubor `schema.xml` znázorňuje schéma, podle kterého jsou dokumenty indexovány. Implicitně jde o schéma pro Apache Solr a jeho využití bude diskutováno v samostatné sekci o Apache Solr 4.1.3.2.

4.1.2.3 Crawlovací proces

Jedno kolo (*round*) crawlování spočívá v následujících krocích:

1. Injektování URL (ang. *Inject*).
2. Tvorba segmentů (ang. *Generate segments*).
3. Získání dokumentů (ang. *Fetch URLs*).
4. Invertování odkazů (ang. *Invert links*).



Obrázek 4.2: Apache Nutch. Hloubka crawlování a volba uzlů.

Zdroj: Autor této práce na motiv z [24].

Uzly jsou jednotlivá URL. Černé šipky jsou odkazy. Modře tečkovaně spojené uzly jsou totožné (např. liší se pouze URL fragment). Červeně škrtlé jsou ignorované uzly (např. byly vyfiltrovány RegExpFiltrem nebo neměly dostatečné skóre). Hodnoty `i` (1–3) označují hloubku od seedu.

5. Zaindexování výsledků (ang. *Index*).

V případě, že chceme crawlovat do větší hloubky, opakují se kroky 2–4. Na vnímání hloubky crawlování z pohledu Nutch je možné vidět na obrázku 4.2.

Verze Nutch 1.X má k dispozici dvě cesty[25], jakými lze spustit crawler. Jsou to skripty

- `bin/nutch` a
- `bin/crawl`.

První skript, nyní deprekovaný, dokázal pouze spouštět výše napsané kroky 1–5 po jednom. Na druhou stranu `bin/crawl` automatizuje použití těchto kroků. V samotném skriptu se dají též nastavit další parametry, jako počet *slaves* a úloh pro Hadoop clustery, max počet URL, který se stáhne a počet vláken na proces. Použití je následující:

```
bin/crawl <seedDir> <crawlDir> <solrURL> <numberOfRounds>
```

kde

[25] V době psaní tohoto dokumentu byla v projektu implementována ještě 3. možnost, kterou je využití REST API crawleru.

Tabulka 4.1: Možnosti skriptu bin/nutch

Příkaz	Akce
`inject crawldb urls`	Injektuje `urls` do `crawldb`
`generate crawldb segments`	Vytvoří seznam URL k získání
`fetch segments/seg`	Získá dokumenty ze segmentu `seg`
`parse segments/seg`	Zparsuje dokumenty ze segmentu `seg`
`updatedb crawldb segments/seg`	Aktualizuje databázi s novými dok.
`invertlinks linkdb -dir segments`	Otočení směru odkazů
`solrindex solrUrl crawldb`	Zaindexuje `crawldb` do Solr na `solrUrl`

- `<seeds>` je adresář se seedy, obsahující textové soubory (alespoň jeden) a každý z nich má na každém řádku jednu URL, nebo přímo soubor s URL,
- `<crawlDir>` je cílový adresář, kam se má uložit získaný obsah,
- `<solrURL>` je URL adresa Apache Solr, kam se mají výsledky zaindexovat a
- `<numberOfRounds>` je počet kol/hloubka prohledávání, viz 4.2.

Pokud by ale byla zapotřebí větší granularita, než jen jednorázové spuštění crawleru, je třeba využít skriptu `bin/nutch`, jehož některé funkce jsem popsal v tabulce 4.1.

4.1.3 Apache Solr

Apache Solr jsem zvolil jako databázi pro nahledané výsledky. Hlavním důvodem byla snadná integrace s crawlerem Apache Nutch a také to, že je vhodnou vyhledávací platformou, která nabízí rozhraní pro zkušené uživatele stejně tak, jako REST API, které je možné využít jinou aplikací.

Apache Solr je open-source enterprise vyhledávací platforma, která vznikla z projektu Apache Lucene. Je napsaná v Javě a součástí je samostatný server pro full-textové vyhledávání.

Mezi jeho hlavní vlastnosti mimo jiné patří:

- Rozhraní pro standardní formáty XML, JSON, HTTP.
- Optimalizace pro velké množství dokumentů.
- indexace „v téměř reálném čase“.
- Vestavené administrátorské rozhraní.

```
<fieldType name="url" class="solr.TextField"
    positionIncrementGap="100">
    <analyzer>
        <tokenizer class="solr.StandardTokenizerFactory"/>
        <filter class="solr.LowerCaseFilterFactory"/>
        <filter class="solr.WordDelimiterFilterFactory" generateWordParts="1"
            generateNumberParts="1"/>
    </analyzer>
</fieldType>
```

Výpis 4.2: Ukázka Solr fieldType

```
<field name="url" type="url" stored="true" indexed="true"/>
<field name="anchor" type="string" stored="true" indexed="true"
            multiValued="true"/>
```

Výpis 4.3: Ukázka Solr field

4.1.3.1 Inicializace

Apache Solr po stažení obsahuje předpřiravenou strukturu a spustitelné soubory, takže je možné jej ihned začít využívat. V adresáři `example` se nachází JAR `start.jar`, který spustí Jetty Servlet.

Po spuštění následujícího příkazu je Solr připraven k použití a na adrese (implicitně) `http://localhost:8983/solr/` by pak měla platforma běžet. Na stejné adrese se přes prohlížeč dostaneme k uživatelskému rozhraní.

```
cd solr-4.10.1/example
java -jar start.jar
```

4.1.3.2 Konfigurace

Solr umožňuje širokosáhlé nastavení v adresářích a podadresářích složky `example`. Pro účely této práce ale stačí konfigurace schémat. Schéma je k nalezení v `example/solr/collection1/conf`, soubor `schema.xml`.

Tento soubor obsahuje informace o tom, která pole bude zaindexovaný dokument mít, jak se s nimi bude pracovat v případě přidání dalších dokumentů nebo když na ně bude dotazováno.

Uvnitř dokumentu v sekci `<types>`, se nachází elementy typu `<fieldType>` a `<field>`. `<fieldType>` definují typ pole, který je pak možno využít.

Pro aplikaci změn ve schématu je potřeba Solr znovu spustit.

4.1.3.3 Dotazování

Apache Solr využívá pro dotazování syntax převzatou z Apache Lucene. Z předchozí části je známo, že indexované dokumenty mají určitou strukturu, definovanou v `schema.xml`. Tato stuktura definuje každý dokument d_i jako množinu o n prvích, složenou z polí $f_{1...n}$ a jejich hodnot $v_{1...n}$.

$$d = \{(f_1, v_1), (f_2, v_2), \ldots, (f_n, v_n)\}$$

Základní dotaz je pak reprezentován výrazem ve tvaru

$$f_i : x$$

kde $i \in \{1 \ldots n\}$ a x libovolná hodnota, na kterou se chceme dotázat. V případě, že by x mělo obsahovat nějaký speciální znak (např. mezeru), je nutné použít uvozovky: f_i : "$x\ y$".

Tento výraz je možné kombinovat s

- Dalšími výrazy, pojenými logickými spojkami „AND“, „OR“.
- Závorkami „(“ a „)“. K určení priority zpracování.
- Unárním operátorem negace, který je značen „NOT“ nebo „−“ (minus). Výsledek pak neobsahuje dokumenty splňující negovanou formuli.

Pro dekonkretizaci dotazu je možné využít zástupného znaku „*“ (hvězdička), který reprezentuje žádný až libovolný počet libovolných znaků.

4.1.3.4 REST API rozhraní

Pro manipulaci s databází je v Solr implementováno rozhraní REST, skrze které je možné data přidávat, upravovat nebo se dotazovat. Operace CRUD je možné realizovat podle následujících instrukcí.

- **Vkládání** nového dokumentu probíhá vložením dat ve formátu XML, splňující strukturu obsahující veškerá povinná pole a platné hodnoty:

```
<doc>
  <field name="f_1">v_1</field>
  <field name="f_2">v_2</field>
  ...
  <field name="f_n">v_n</field>
</doc>
```

  Endpoint URL: http://localhost:8983/solr/update

  HTTP metoda: POST

- **Dotazování**. Pro dotazování se využívá hodnot v query string, které musí být zakódované pro URL (ang. URL-encoded). Jejichž jednotlivé významy jsou:
  - *q* – dotaz, viz 4.1.3.3.
  - *sort* – řazení, viz 4.1.3.3.
  - *start* – specifikuje offset z výsledku dotazu. Implicitně 0.
  - *rows* – maximální počet dokumentů obsažen v jedné stránce odpovědi na výsledek dotazu. Implicitně 10.

  - *fl* – seznam názvů polí dokumentu, které se mají vrátit ve výsledku, oddělených čárkou, nebo mezerou.
  - *wt* – určí, jaký *writer type* se má použít (v jakém formátu se vrátí výsledek). Možné hodnoty: json, xml.

  Endpoint URL: http://localhost:8983/solr/select

  HTTP metoda: GET

- **Úprava** dokumentu probíhá obdobně jako vkládání, jen součástí upravovaných polí je příslušný modifikátor v atributu `update` elementu `field`. Pole bez modifikátoru jsou brány jako selekční, tj. změní se dokumenty mající stejné hodnoty u specifikovaných polí.

```
<doc>
  <field name="f_1" >v_1</field> <!-- id -->
  <field name="f_2">v_2</field> <!-- dalsi id -->
  <field name="f_i" update="set">v_i</field> <!-- nova
       hodnota -->
  ...
</doc>
```

  K dispozici jsou modifikátory

  - *set* – nastaví pole na požadovanou hodnotu.
  - *add* – přidá hodnotu do vícehodnotového pole.
  - *remove* – odebere hodnotu/y z vícehodnotévho pole.
  - *removeregex* – odebere hodnoty specifikované regulárním výrazem.
  - *inc* – inkrementuje pole o specifikovanou hodnotu.

  Endpoint URL: http://localhost:8983/solr/update

  HTTP metoda: POST

- **Mazání** dokumentu probíhá opět zasláním zprávy ve formátu XML na příslušný endpoint. Obsahem zprávy je selekční dotaz, odpovídající formě z 4.1.3.3. Zpráva musí být strukturu

```
<delete>
    <query>QUERY_TEXT</query>
</delete>
```

  Endpoint URL: http://localhost:8983/solr/update

  HTTP metoda: POST

Pokud byly provedeny změny, je nutné je potrvdit, aby se projevily v dalších požadavcích. To se dá udělat zanesením `<commit/>` do zprávy se změnou, případně zasláním HTTP požadavku GET na adresu Solr endpointu `http://localhost:8983/solr/update?commit=true`.

4.1.3.5 Integrace s Apache Nutch

Apache Nutch [22] je s Apache Solr úzce spojen, protože je jeho hlavní a první implementovanou indexovací a vyhledávací platformou. Nutch již disponuje potřebnými mechanizmy pro komunikaci se Solr, proto samotná integrace není až tak složitá.

1. V první řadě je třeba uzpůsobit schéma tak, aby odpovídalo struktuře od crawleru. Nutch má toto schéma umístěné v `conf/schema.xml` a obsahuje potřebnou strukturu (v případě přidání vlastních pluginů je potřeba tomu přizpůsobit i schéma, indexuje-li plugin další atribut). Tímto schématem je nutno nahradit schéma Solr:

   ```
   cp ${NUTCH_RUNTIME_HOME}/conf/schema.xml
      ${APACHE_SOLR_HOME}/example/solr/collection1/conf/
   ```

2. Protože schéma od Nutch nedrží krok s vývojem Solr, je potřeba u nově zkopírovaného schématu v Solr zakomentovat řádky (53–54), z důvodu již neexistující třídy.

   ```
   <!--   <filter class="solr.EnglishPorterFilterFactory"
       protected="protwords.txt"/> -->
   ```

3. Dále za řádek definující pole pro ID <field name="id"... /> (zhruba řádek 70), přidat pole pro verzi

   ```
   <field name="_version_" type="long" indexed="true"
       stored="true"/>
   ```

4. A nakonec znovu spustit Apache Solr (podle 4.1.3.1), aby nově upravené schéma nabylo platnosti.

Nyní je Nutch schopen posílat nasbírané dokumenty k indexaci.

```
bin/nutch solrindex http://127.0.0.1:8983/solr/
          crawl/crawldb
          -linkdb crawl/linkdb crawl/segments/
```

4.1.4 Weka

Software Weka (Waikato Environment for Knowledge Analysis) obsahuje algoritmy na strojové učení určených pro data mining, použiji pro část strojového učení. Jak bylo zmíněno, možností je více (např. RapidMiner), ale volím Weku, protože s ním mám již nějaké zkušenosti.

V této části projekt Weka v krátkosti představím a nastíním práci s ním.

4.1.4.1 Co je Weka

V první řadě je Weka[26][19] workbench, vizuální nástroj, pro úlohy data miningu, disponující nejmodernějšími algoritmy strojového učení, předzpracování a závěrečné zpracování dat. Podporuje celý proces pro experimentální data mining, včetně přípravy vstupu, vyhodnocení učících schémat a vizualizaci vstupních dat po aplikaci učícího schématu [20].

Weka je napsán v jazyce Java a všechny implementované algoritmy a podpůrné třídy jsou dostupné jako sada knihoven.

Při práci s Weka *Explorer* (název grafického uživatelského rozhraní) nebo i v samotném kódu je určitě dobré znát formát ARFF, se kterým software pracuje.

Ovšem Weka Explorer se hodí pro práci s malými, či středně velkými datasety. První, co při otevření datasetu ke zpracování udělá, je, že jej celý načte do operační paměti. Pro to tvůrci vytvořili další GUI zvané *Knowledge Flow*, které umožňuje pomocí vizuálního nástroje zpracovat velké datasety postupně. Vizuální nástroj obsahuje vytváření a propojování stavebních bloků, skrz které pak data „tečou“ a podle uzlů, které navštíví, se s nimi něco stane.

4.1.4.2 ARFF formát

ARFF [26] (attribute relation file format) je ASCII txtový soubor popisující dataset a jeho instance s atributy, využívaný sofwarem Weka.

```
pocasi,teplota,vlhkost,vitr,hrat

slunecno,teplo,vysoka,ne,ne
slunecno,teplo,vysoka,ano,ne
zatazeno,teplo,vysoka,ne,ano
destivo,stredni,vysoka,ne,ano
destivo,chladno,vnormalu,ne,ano
destivo,stredni,vnormalu,ano,ne
zatazeno,chladno,vnormalu,ano,ano
slunecno,stredni,vysoka,ne,ne
slunecno,chladno,vnormalu,ne,ano
```

```
@relation pocasi

@attribute pocasi {slunecno,zatazeno,destivo}
@attribute teplota {teplo,stredni,chladno}
@attribute vlhkost {vysoka, vnormalu}
@attribute vitr {ano,ne}
@attribute hrat {ano,ne}

@data
slunecno,teplo,vysoka,ne,ne
slunecno,teplo,vysoka,ano,ne
zatazeno,teplo,vysoka,ne,ano
destivo,stredni,vysoka,ne,ano
destivo,chladno,vnormalu,ne,ano
destivo,stredni,vnormalu,ano,ne
zatazeno,chladno,vnormalu,ano,ano
slunecno,stredni,vysoka,ne,ne
slunecno,chladno,vnormalu,ne,ano
```

Rozdíl souboru ve formátu CSV (vlevo), přepsaném do formátu ARFF (vpravo)

Na ukázce je vidět, že dokument ARFF se dá vytvořit celkem jednoduše, z pouhého CSV souboru (který je opět možné snadno vytvořit z kteréhokoliv

[26] http://www.cs.waikato.ac.nz/

tabulkového procesoru). Soubor ARFF pouze doplňuje některé anotace začínající zavináčem (@) a za nimi na stejném řádku nasleduje jejich hodnota. Anotace můžou být:

- *relation*. Udává název datasetu.
- *attribute*. Hodnotou je název atributu a typ. Podporované typy jsou *string*, *numeric*, *date*, případně specifikované možnosti, když jde o nominální atribut, uzavřených v {}, jak je na ukázce.
- *data*. Uvádí začínající seznam instancí datasetu. Jeho obsahem je na každém řádku jedna instance, složená z hodnot atributů oddělených čárkou (,). Pokud hodnota atributu není známa, doplňuje se otazník (?).

4.1.4.3 Stručný přehled algoritmů

Weka nabízí celou řadu algoritmů pro strojové učení a klasifikaci. Několik z nich bude možné využít v crawleru, včetně jejich možné konfigurace. Podle tříd to jsou:

- **Bayesovské**. Konkrétně *NaiveBayes*, je statistické modelování rozhodování tak, že všechny atributy jsou na sobě *nezávislé* a *stejně důležité*.
- **Stromové**.
  - *J4.8* open-source Java implementace algoritmu C4.5. Vytváří rozhodovací strom ang. *decision tree*) podle uzlů s nejmenší entropií.
  - *Random Tree* vytváří strom na základě k náhodně zvolených atributů.
  - *Random Forest* vytváří větší množství rozhodovacích stromů a výsledkem je modus (nejčastější hodnota) vrácená jednotlivými stromy. Tím eliminuje overfitting, který se objevuje u rozhodovacích stromů.
  - *NB-Tree* hybrid mezi Naive Bayes a rozhodovacími stromy – listy stromu jsou Bayesovské klasifikátory.
- **Regresní**. Konkrétně *Logit boost*. Používá pro učení aditivní logistickou regresi.
- **Pravidlové**. Jediný zástupce *Decision table*
- **Funkční**. Ve Weka označení pro třídu klsifikátorů, které je možné zapsat jako matematickou rovnicí (narozdíl od ostatních, např. stromů a s jedinou výjimkou Naive Bayes).
  - *Logistic* založen na lineárním regresním modelu.

– *SVM* (Support vector machine) dokáže řešit binární klasifikační problémy. Cílem je najít nadrovinu mezi klasifikovanými daty takovou, aby vzdálenost roviny od obou skupiny, které rovina odděluje, byla co největší.
– *SMO* (Sequential minimal optimization) je vylepšená verze pro SVM model.

4.1.5 Google CSE

Google Custom Search Engine[27] je služba společnosti Google, která umožňuje vytvoření a použití vyhledávače na základě vlastních požadavků za využití existujícího Google indexu.

V současné době má dvě možnosti na vytvoření. První využívá pouze filtr pro specifikované URL, které jsou v CSE prohledávány. Druhá možnost je založit vyhledávací stroj na klíčových slovech. A protože neznám seznam URL pro stránky k vyhledávání – leda by to byly API repozitáře, na které se nechci omezovat – využiji k vytvoření CSE klíčová slova.

Klíčová slova ovlivní, jaké spektrum domén zahrne vyhledávač do pozdějšího vyhledávání. Je proto vhodné volit slova související s Web API dokumentací, jako např. „REST“, „API“, „Documentation“ apod.

Poté dojde k vytvoření URL vzorů, většinou se jedná o domény, které budou prohledávány při zadání dotazu. Tyto vzory je možné dále rozšířit, modifikovat. Nakonec vznikne seznam URL, který je také možné upravit.

Výsledkem je vyhledávací stroj s jedinečním identifikátorem, který je možné využívat jak v prohlížeči, jako klasický Google vyhledávač, tak pomocí RESTful rozhraní přes HTTP.

Požadavek musí obsahovat:

- API klíč – `key` – klíč identifikující klienta pro využívání služeb Google.
- CSE identifikátor – `cx` – identifikátor vyhledávače.
- Dotaz – `q` – výraz pro specifikaci hledání.

Odpověď může být ve formátu JSON nebo Atom, která obsahuje základní info o dotazu, vyhledávacím stroji, výsledcích a nakonec samotné výsledky. Maximální počet výsledků je ale 10, zbytek je dostupný přes stránkovací systém. Ukázku požadavku a odpovědi je možné vidět v příloze B.

Také samotné používání vyhledávacího stroje je omezené na počet 100 dotazů za den, pokud je využit běžný API klíč. Nicméně pro několik vyhledání to stačí a v případě potřeby je možné klientský účet za peníze vylepšit a počet dotazů bude navýšen.

[27] https://cse.google.com/cse/

4.2 Pozitivní část trénovacích dat a analýza

Tato část realizace popisuje, jak byla získána a analyzována pozitivní část trénovacích dat (dokumenty typu API dokumentace) pro vytvoření datových modelů pro strojové učení.

4.2.1 Získání Web API dokumentací

Pro získání web API dokumentací jsem jako hlavní zdroj použil webové stránky ProgrammableWeb – v současné době největší veřejný repozitář webových API, který k tomuto dni čítá 13 238 záznamů.

Záznam API obsahuje informační údaje, *API Provider* – odkaz na poskytovale služby, *API Homepage* – odkaz na dokumentaci webového API, *Primární kategorii* – kategorie API (analytics, social, atp.), a další.

Vyšel jsem ze seznamu API seřazeného podle oblíbeností uživatelů. Tím se zaměřím na nejvíce používaná API, jejichž oblíbenost může plynout z potřebnosti jejich využití, anebo také z pohodlnosti, s jakou se s nimi pracuje. A protože neexistuje žádný definovaný standard pro psaní API dokumentace, spolehnu se na nepsaný standard, který svým přístupem definují uživatelé programátoři.

Záznamy jsem procházel seřazené sestupně podle oblíbenosti a ručně jsem ke každému záznamu hledal API dokumentaci — východiskem byl odkaz *API Homepage*. Pokud cílový dokument API nebyl, pokusil jsem se ji najít přes odkazy vedoucí z této stránky. Při selhání jsem se API pokusil najít přes vyhledávač Google nějakou API od tohoto poskytovatele – *API Provider*. Pokud ani to nevyšlo, přišel jsem na další záznam. Výsledky jsou číselně prezentované v tabulce 4.2. V kategorii „Ostatní“ jsou dokumenty, které nebyly dostupné, nebo je nebylo možné dohledat a/nebo je nebylo možné vyhodnotit jako web API dokumentaci.

Z celkového počtu získaných web API dokumentací mělo přibližně 12% z nich svůj obsah generovaný Javascriptem. Tento způsob používají zejména větší poskytovalé jako jsou Google či Facebook. Z tohoto důvodu, jelikož jsem neměl dosud zprovozněný crawler, který by toto umožňoval, ručně jsem stáhl veškerý čitelný obsah stránky z prohlížeče do souboru na disku – to si mohu dovolit, protože veškeré techniky pro klasifikaci využívají pouze množinu slov a rovnou bylo docíleno odstranění HTML značek, obrázků apod. K automatickému zpracování takto generovaných dokumentů je zapotřebí jinak sofistikovanější postup, využívající skutečný prohlížeč. Jedním z takovýchto řešení je např. Selenium, které nabízí API pro práci s drivery, kterými je pak možné automaticky ovládat prohlížeč [27]. V tomto případě by pomohl k vykonání Javascriptu na stránce. Tento aumatizovaný postup bude zakomponován v crawleru přesně pro tyto případy.

Při procházení záznamů z ProgrammableWeb byly získány API dokumentace z 26 kategorií. Jmenovitě se jedná o kategorie Advertising, Ana-

Tabulka 4.2: Získané API dokumentace

Zkontrolováno záznamů	API dokumentace	Ostatní
132	100	32

lytics, Bookmarks, Blogging, Calendars, Database, ECommerce, Enterprise, Fitness, Jobs, Language, Mapping, Messaging, Music, Other, Payments, Photos, Social, Search, Storage, Telephony, Tools, Translation, Video, Widgets a Weather.

4.2.2 Analýza API dokumentací

Nyní mám množinu API dokumentací v souborech, jejichž obsahem je čistý text. Nyní je na čase získat podklady pro reprezentaci dokumentů jako instancí pro strojové učení (viz 2.3). V sekci Klasifikace dokumentů byly zmíněny dva datové modely instancí, které budou použity. V tuto chvíli je řeč o **DM1**, který byl použit v [12]. Pro experiment byla použita slova *api, method, operation, input, output, parameter, get, post, put, delete, append, url.* Já provedu analýzu svých zkušebních dat a pokusím se tuto množinu rozšířit o další slova, často se vyskytující v API dokumentacích.

Princip je zřejmý: najít slova často se vyskytujících v API dokumentacích. Pro tento účel vytvořím program, který mi připraví podklady pro následnou analýzu. Program zprcuje všechny doposud sesbírané API dokumentace $\mathcal{A}_T$ pro každé slovo w_i této kolekce zjistí *počet dokumentů*, ve kterých se vyskytuje, tj.:

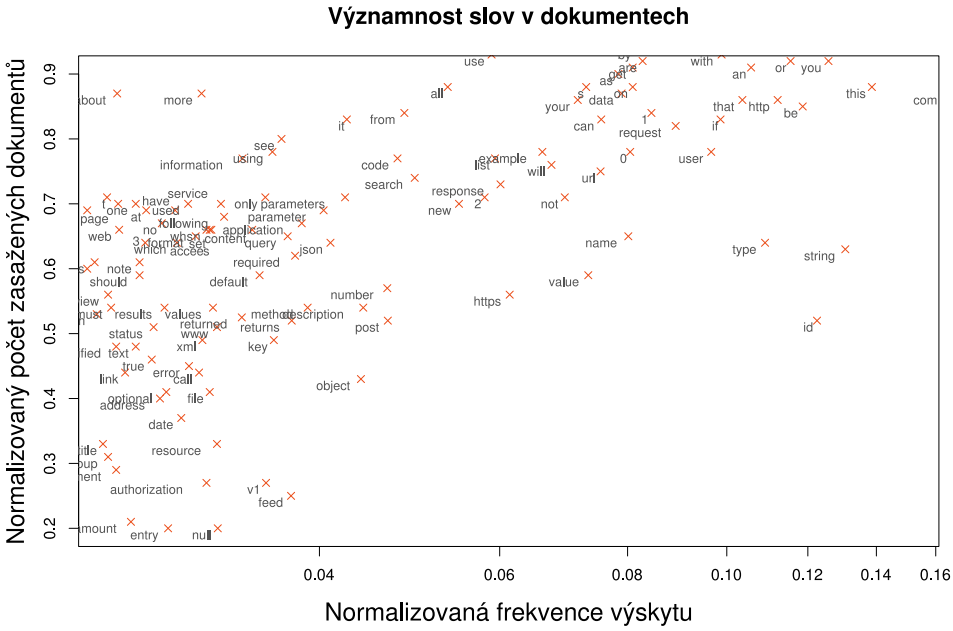
$$|\{\forall d \in \mathcal{A}_T; w_i \in d\}|$$

a *celkový počet výskytů* slova w_i v kolekci:

$$\sum_{d \in \mathcal{A}_T} \sum_{w \in d} |\{w; w = w_i\}|$$

Pokud výsledky zobrazím do grafu, který ukazuje závislost celkového počtu výskytů na počtu dokumentu každého slova, je už snadné vybrat slova související s API. Ty budu vybírat v okolí oblasti výskytu slov, která byla použita v [12]. Graf je možné vidět na obrázku 4.3. Tabulka 4.3 ukazuje podrobnější statistiky o těchto datech.

Výsledkem je tedy následující možina slov: *api, method, operation, input, output, parameter, get, post, put, delete, append, url, data, request, http, https, default, xml, json, example, response, description, required.*



Obrázek 4.3: Analýza slov API dokumentací

Tabulka 4.3: Přehled dat trénovací sady API dokumentací

Počet dokumentů	100
Celkový počet slov	73 562
Celkový počet různých slov	11 379
Medián počtu slov v dokumentu	599
Max. počet slov v dokument	9 290
Min. počet slov v dokumentu	46

4.3 CSE a negativní část trénovacích dat

4.3.1 Tvorba CSE vyhledávače

Nyní, když mám množinu slov reprezentující tyto dokumenty, je na čase vytvořit CSE vyhledávač zaměřený na Web API. V rámci testovací části práce jsem vyzkoušel vytvořit několik různých vyhledávacích strojů z různých klíčových slov. Zhodnocení jednotlivých konfigurací strojů je k nahlédnutí v kapitole 5.1. Pro finální verzi crawleru byl použit vyhledávač vytvořený dle konfigurace 4.4.

Po vytvoření vznikne customizovaný Google vyhledávač, který na základě specifikace omezí své hledání na seznam URL vzorů, které jsou při použití pro-

Tabulka 4.4: Konfigurace Google CSE

Creation from keywords	☑
Keywords	*api, rest, http, json, xml, get, post, documentation*
Suggest more	☐
Add all	☐
Expand keywords	☑
Restrict the URL patterns to	*All Countries*

hledány. Seznam je možné dál upravovat vyškrtnutím nebo přidáním nějakých URL vzorů, ale já ho použil v této podobě. Důležitou informací identifikující tento vyhledávač je jeho identifikátor *Search Engine ID*, které může vypadat např. `004173228268281370859:atvecc__7du`.

4.3.2 Negativní část trénovacího datasetu

Pro správnou funkci algoritmů na strojové učení je podstatné mít trénovací kolekci obsahující data všech tříd – v tomto případě dvě. Získání pozitivní části – API dokumentací – je popsáno v sekci 4.2.1 a o tom, jak se získájí dokumenty negativního charakteru – ne-API dokumentace – popíši nyní.

Crawler bude vycházet prvotně ze seedů získaných z CSE vyhledávače. Určitě tedy přijde do styku s daty, které vyhledávač může vrátit jako výsledek dotazu. Původní myšlenka crawleru byla taková, že uživatel crawleru zadá heslo, dejme tomu, že kategorii API – např. počasí (weather) a crawler nalezne dokumentace API vztahující se k počasí.

Postupně tedy kategorie API dokumentací, získaných v 4.2.1, opět ručně, vyhledávám ve vytvořeném vyhledávači a prvních několik výsledků vyhodnotím, zda-li se opravdu jedná o dokument *nepopisující API* dokumentaci, a stejným způsobem jako v předchozím případě uložím obsah na disk. Tím vznikne kolekce dokumentů, které obsahují čistý text dokumentu, který nepopisuje API dokumentaci. V mém případě je polovina datasetu pozitivní a polovina negativní.

4.4 Inicializace Nutch a Solr

4.4.1 Nutch

Pro implementovanou verzi crawleru byl použit Nutch 1.9, jehož jedinou nutnou podmínkou pro úspěšné spuštění je nastavení jména crawleru v `nutch-site.xml` přidáním vlastnosti

```
<property>
    <name>http.agent.name</name>
    <value>APISpider</value>
</property>
```

Dále se ujistím, že filtrování URL v Nutch ignoruje obrázky, mailto apod.:

```
-^(file|ftp|mailto):
-[?*!@=]
-.*(/[^/]+)/[^/]+\1/[^/]+\1/
+.
```

Nyní po sestaveni Nutch je připraven pro crawlování.

Nastavení pluginů. Jelikož celý proces crawlování v Nutch je realizován pomocí pluginu, v tabulce 4.5 je popsán seznam pluginů, které jsem pro tuto práci použil.

Tabulka 4.5: Použité pluginy pro Nutch

Plugin	Funkce
protocol-selenium	HTTP protokol umožňující spuštění Javascriptu na stránce.
urlfilter-regex	Umožní filtrovat URL podle regulárních výrazů
parse-(html|tika)	Parsuje HTML a získá čistý obsah
index-(basic|anchor)	Přidává nutná pole pro indexaci
indexer-solr	Indexuje data do Solr
scoring-opic	Hodnocení kvality URL
urlnormalizer-(pass|regex|basic)	Pro normalizaci URL

Aktivace pluginu Selenium pro Javascriptově generovaný obsah. V sekci 4.2.2 o analýze API bylo zmíněno, že nemalá část API webových dokumentů generuje svůj obsah Javascriptem. Pro jeho správné vykonání je potřeba prohlížeč. Selenium nabízí ovladače pro různé typy prohlížečů a umožňuje vykonat Javascript na stránce a zpřístupnit tak staticky nedostupný obsah.

Již dlouho byl protokol Selenium ve formě pluginu součástí Nutch verze 2.x. Po hodinách studování architektury Nutch jsem se pustil do implementování protokolu pro Nutch 1.x, ale ke všeobecnému štěstí byl implementován samotnými autory do verze `trunk`.

Pro jeho úspěšné zapojení do verze 1.9 je zapotřebí

1. Opatřit si i Nutch ve verzi trunk[28].

   Verze trunk v době implementace crawleru nebyla schopná provozu, kvůli problémům s indexací, je tedy potřeba ji nainstalovat do verze 1.9.

2. Přemístit plugin a jeho součásti

   ```
   cp -r ${NUTCH_TRUNK_SRC}/src/plugin/lib-selenium
         ${NUTCH_1.9_SRC}/src/plugin/
   ```

[28] http://svn.apache.org/repos/asf/nutch/trunk/

3. Přidat Selenium do závislostí Nutch úpravou souboru /ivy/ivy.xml

```
<!-- NUTCH_1.9_SRC/ivy/ivy.xml -->

<ivy-module version="1.0">
  <dependencies>
    ...
    <!-- begin selenium dependencies -->
    <dependency org="org.seleniumhq.selenium" name="selenium-java" rev="2.42.2" />

    <dependency org="com.opera" name="operadriver" rev="1.5">
      <exclude org="org.seleniumhq.selenium" name="selenium-remote-driver" />
    </dependency>
    <!-- end selenium dependencies -->
  </dependencies>
</ivy-module>
```

4. Informovat o pluginu Ant upravením /src/plugin/build.xml souboru:

```
<!-- NUTCH_1.9_SRC/src/plugin/build.xml -->

<project name="Nutch" default="deploy-core" basedir=".">
  <!-- ================================================== -->
  <!-- Build & deploy all the plugin jars.                -->
  <!-- ================================================== -->
  <target name="deploy">
    ...
    <ant dir="lib-selenium" target="deploy"/>
    <ant dir="protocol-selenium" target="deploy" />
  </target>
      ...
</project>
```

5. A v poslední řadě v konfiguraci Nutch, v souboru nutch-site.xml přidat property `plugin.includes`, která bude vypadat úplně stejně, jako v nutch-default.xml, ale namísto protokolu http bude použit protokol selenium. Konfiguraci Nutch je věnována sekce 4.1.2.2. Výsledek úpravy může takto:

```
<!-- NUTCH_1.9_SRC/conf/nutch-site.xml -->
<property>
    <name>plugin.includes</name>
    <value>protocol-selenium|urlfilter-regex| ... </value>
</property>
```

6. Po sestavení Nutch je plugin funkční.

4.4.2 Solr

Pro nastavení Solr je potřeba jej integrovat s Nutch, což je popsáno v sekci 4.1.3.5. A přidat do schématu pole, které využívá modul pro identifikaci API dokumentací.

Do schématu v Solr tedy přidám do pole `<fields>` nová pole `api-prob` a `api-class`.

```
<fields>
    ...
    <field name="api-prob" type="float" stored="true" indexed="true" default="-1.0"/>
    <field name="api-class" type="string" stored="true" indexed="true" default=""/>
...
```

Solr musí být pro funkci crawleru znovu spuštěn s novým schématem (viz 4.1.3.1).

4.5 Získání kandidátů

Získání URL kandidátů je první fáze, která probíhá již při spuštění crawleru. Vstupem je dotazová fráze q od uživatele. Ta je použita jako dotaz na CSE, který vyhledá dostupné webové stránky a vrátí je ve své odpovědi. Tuto odpověď crawler zpracuje a vybere prvních k_n záznamů.

CSE implementuje rozhraní RESTful, a komunikace probíhá jednoduše a bez problémů. Požadavek na „social“ API by vypadal takto:

```
GET /customsearch/v1
    ?key=AIzaSyAT1M2PUBltxCXPWwsBbuNSIso9UbOJNVg
    &cx=004173228268281370859:atvecc__7du
    &q=social HTTP/1.1
Host: www.googleapis.com
```

kde hodnota *key* je identifikátor uživatele API služeb Google, *cx* je identifikátor CSE a q je dotaz.

Odpověď ve formátu json. Jeho základní struktura je taková:

```
{
 "kind": "customsearch#search",
 "url": {
  "type": "application/json",
  "template": ...
 },
 "queries": {
  "nextPage": [
   {
    "title": "Google Custom Search - social",
    "totalResults": "25",
    "searchTerms": "social",
    "count": 1,
    "startIndex": 2,
    "cx": "013315504628135767172:d6shbtxu-uo"
   }
  ],
  "request": [
   {
    "title": "Google Custom Search - social",
    "totalResults": "25",
    "searchTerms": "social",
    "count": 1,
    "startIndex": 1,
    "cx": "013315504628135767172:d6shbtxu-uo"
   }
  ]
 },
 "context": {
  "title": "CSE",
  "facets": [ ... ]
 },
 "searchInformation": { ... },
 "items": [
  {
   "kind": ... ,
   "title": ... ,
   "htmlTitle": ... ,
   "link": ... ,
   "displayLink": ...,
   "snippet": ... ,
   "htmlSnippet": ... ,
   "formattedUrl": ... ,
   "htmlFormattedUrl": ... ,
   "pagemap": { ... },
   "labels": [ ... ]
```

```
  }
 ]
}
```

Výpis 4.4: Struktura odpovědi od Google CSE

Odpovědí je tedy objekt, obsahující informace o požadavku a jeho výsledku. V tuto chvíli stojí za zmínku

- *queries* – obsahuje informace o aktuálním požadavku a další manipulaci
  - *nextPage* jsou informace o další stránce výsledků, jaká položka je první (hodnota *start*), kolik výsledků je celkem (*totalResults*).
    Obdobně vypadaly informace o předchozí stránce *previousPage*.
  - *request* obsahuje informace o současném požadavku.
- *searchInformation* obsahuje informace o hledání (délka hledání, nalezených výsledků, apod.)
- A konečně *items* je pole, obsahující jednotlivé záznamy pro kandidáty. Ty jsou typu *item*, který reprezentuje jeden výsledek dotazu a jeho součástí jsou zejména URL *link*, titulek *title*, *snippet* pro „náhled“ obsahu výsledku, jako z klasického vyhledávače, *htmlSnippet* je HTML formátovaný.

Protože maximální počet výsledků na stránku v odpovědi je 10, v případě, že uživatel specifikuje větší číslo, je sérií požadavků přes *nextPage* iterováno, dokud není sesbíraných všech k_n kandidátních URL. Obecně se tedy bere $max(k_n, totalResults)$ výsledků.

Pokud nebyl nalezen ani jeden kandidát ($totalResults = 0$), crawler nemá jak pokračovat a končí.

Lehce složitější ukázkový požadavek a odpoveď je k nalezení v příloze B.

4.6 Crawl dokumentů

4.6.1 Sběr

Pro sběr dokumentů v stačí spustit Nutch s potřebnými parametry. Jako seedy dostane URL kandidáty z předchozího kroku, které byly získány z CSE vyhledávače. K tomu využiji skript `bin/crawl`.

```
bin/crawl <seeds> <crawlDir> <solrURL> <numberOfRounds>
```

Kde `seeds` jsou seedy získané z CSE, uložené v souboru `seeds.txt`. `crawlDir` je adresář pro nasbírané dokumenty, v mé implementaci je pod názvem `wapicrawl`, který obsahuje podsložky podle vyhledávací fráze *q*. Parametry `solrUrl` (URL Solr endpoint) a `numberOfRounds` (hloubka crawlování) jsou parametry specifikované uživatelem.

To zajistí nacrawlování dokumentů do hloubky `numberOfRounds` od každého URL kandidáta ze seedů, při reskeptování dalšího nastavení Nutch (např. regexpUrlFilter).

Plugin Tika zajišťuje preprocessing a parsování HTML dokumentu a není použit žádný stemmer ani lemmatizace.

4.6.2 Indexace

Dokumenty jsou nyní v binární podobě v adresáři `wapicrawl/q`. Pro indexaci použiji skript `bin/nutch`

```
bin/nutch solrindex <solrUrl> wapicrawl/<q>/crawldb
```

kde `solrURL` je uživatelem specifikovaný endpoint Solr a `q` dotaz uživatele.

Nyní jsou nasbírané dokumenty zaindexovány v Solr přístupné k dotazování. Zbývá fáze klasifikace dokumentů.

4.7 Automatická identifikace

Poslední fáze, nezávislá od ostatních je fáze klasifikace dokumentů do tříd *API dokumentací* a *ostatní*. V této fázi dojde k natrénování konkrétního algoritmu strojového učení ze specifikované trénovací sady a následně pomocí nich budou klasifikovány dokumenty indexované v Solr spadajících k dotazu uživatele *q*. Výsledky klasifikace budou zaneseny do přidaných polí `api-class`, které bude obsahovat třídu (1 pro API dokumentace, 0 pro ostatní) a `api-prob` nese pravděpodobnost, s jakou do dané třídy patří. Aktualizace hodnot těchto probíhá skrze REST rozhraní Solr.

Jak bylo řečeno v části návrhu, pro identifikaci použiji struktury a algoritmy dodané se softwarem Weka. Seznam podporovaných algoritmů a jejich knihovny jsou uvedeny v tabulce 4.6.

Tabulka 4.6: Podporované klasifikační algoritmy

Algoritmus	Typ	Třída
Naive Bayes	Bayes	`weka.classifiers.bayes.NaiveBayes`
Logistic	Funkční	`weka.classifiers.functions.Logistic`
SMO	Funkční	`weka.classifiers.functions.SMO`
SVM	Funkční	`weka.classifiers.functions.LibSVM`
Decision Table	Pravidla	`weka.classifiers.rules.DecisionTable`
Logit Boost	Regrese	`weka.classifiers.meta.LogitBoost`
NB-Tree	Strom	`weka.classifiers.trees.NBTree`
J48	Strom	`weka.classifiers.trees.J48`
Random Forest	Strom	`weka.classifiers.trees.RandomForest`
Random Tree	Strom	`weka.classifiers.trees.RandomTree`

4.7.1 Různé reprezentace instancí

Crawler podporuje dvě možnosti reprezentování instancí (dokumentů), zmíněných v 3.4.3.

Pro **DM1** je dokument reprezentován jako vektor n atributů. Hodnota těchto atributů je binární a je dána funkcí:

$$\phi(d, w, t) = \begin{cases} 1 & \text{pokud je slovo } w \text{ v dokumentu } d \text{ obsaženo alespoň } t\text{-krát} \\ 0 & \text{jinak} \end{cases} \tag{4.1}$$

kde, w je klíčové slovo z množiny slov, u kterých bylo zjištěno, že se v API dokumentacích objevují často 4.2.2. Porovnávání ignoruje velikost písmen. Prahová hodnota byla zvolena stejná jako v [12]. Tedy 3, kvůli osvědčeným výsledkům, je ale možné jí editovat (viz 4.7).

Druhou reprezentací datového modelu instance **DM2** je aplikován Weka filtr `StringToWordVector` s nastavením

```
-R first-last
-W 1000
-prune-rate -1.0
-N 0
-stemmer weka.core.stemmers.NullStemmer
-M 1
-tokenizer weka.core.tokenizers.WordTokenizer
-delimiters "\" \\r\\n\\t.,;:\\\'\\\"()?!\"
```

Klasifikace je pak prováděna za běhu pomocí `FilteredClassifier`. Protože při aplikaci filtru by vznikla chyba ve Weka z důvodu, že by instance mohly mít rozdílný počet a typ atributů. Pomocná třída FilteredClassifier dokáže tento problém zpracovat.

4.8 Konfigurace crawleru

Konfigurace crawleru a jeho částí je možná přes připravný konfigurační soubor `crawler.properties`, který odpovídá struktuře Java Properties, tedy `klíč=hodnota` na každém řádku. Ačkoliv tedy statické konfigurování crawleru Nutch je ponechána v jeho režii a je možné jej nakonfigurovat zvlášť.

Klíče a jejich významy jsou uvedeny v tabulce 4.7.

4.8.1 Trénovací dataset

Crawler podporuje dvojí podobu trénvacích dat

- *Soubor typu ARFF.* Trénovací instance jsou předány formátem ARFF, který je popsán v sekci 4.1.4.2. Je vyžadováno aby hodnoty atributů instance nabývaly hodnot z množiny $\{y, n\}$. Hodnota „y“, jedná-li se o označení výskytu i-tého klíčového slova vícekrát, než je prahová hodnota, resp „n“ v opačném případě. Atributy se musí pořadím shodovat

Tabulka 4.7: Konfigurovatelné vlastnosti crawleru

Položka	Popis
solr.url	URL Apache Solr endpoint. Např.: *http://localhost:8983/solr/*
nutch.home	Cesta k Apache Nutch. Např.: */path/to/apache-nutch-1.9*
depth	Hloubka crawlování podle 4.2
cse.api-key	Aplikační klíč pro Google API
cse.cx	Identifikátor vyhledávacího stroje Google CSE
num	Počet URL kandidátů vyhledávače
trainingSet	Cesta k trénovacímu datasetu 4.8.1
classifier	Jaký klasifikační algoritmu se má použít. Možnosti: *j48, naiveBayes, svm, smo, randomForest, NBTree, logistic, randomTree, decisionTable, logitBoost*
classifier.parameters	Parametry pro kklasifikační algoritmus ve formě odpovídajícího řetězce z Weka. Např.: *-C 0.265 -M 2*
model	Model reprezentace instance, který se má použít. Buď to *dokument* pro **DM2**. Nebo čárkami oddělený seznam specifických slov pro **DM1**, např.: *api,documentation,rest.*
threshold	V případě **DM1** – práh, při kterém bude hodnota atributu přechází z 0 do 1, viz funkce 4.1

vyznačených v konfiguračním souboru crawleru. Hodnoty klassifikační třídy jsou „yes“, resp. „no“, pokud se jedná, resp. nejedná o API dokumentaci. Tento způsob podporuje pouze reprezentaci instancí **DM1**.

- *Kolekce souborů.* Trénovací sada je vytvořena automaticky podle příslušné konfigurace crawleru. Na specifikované zdrojové cestě očekává dva adresáře
  - Adresář *yes* obsahuje soubory pozitivní části datasetu – API dokumentace.
  - Adresář *no* obsahuje sobory negativní části datasetu – ne-API dokumentace.

Kapitola 5

Vyhodnocení

Kapitola popisuje, jak bylo přistupováno k testování a vyhodnocení jednotlivých částí a aspektů crawleru.

5.1 Vyhodnocení konfigurací CSE

CSE vyhledávač je zdrojem zdroj URL kandidátů (seedů) pro crawler. Tato sekce prezentuje porovnání několika CSE vyhledávačů, které se liší ve složení klíčových slov. Pozorovány byly zejména schopnosti nalézt API a rozmanitost výsledků.

Pro tyto účely jsem vytvořil několik testovacích CSE vyhledávačů *WebApisX*. Množiny použitých klíčových slov pro každé CSE je vypsáno v tabulce 5.1

Tabulka 5.1: Testované konfigurace CSE

CSE	Klíčová slova
WebApis1	api, documentation, rest
WebApis2	api, reference, rest, json, http
WebApis3	api, rest, restful, json, xml, web service, description
WebApis4	api, rest, http, json, xml, get, post, documentation
WebApis5	api, rest, http, json, xml, get, post, documentation, reference, service

Následně jsem testoval všechny CSE na kvalitu výsledků z hlediska relevance a pokrytí pro seedy. V tabulce 5.2 jsou uvedeny výsledky tohoto porovnání. Porovnání probíhalo v ohledu nalezení API s ohledem na prozkoumání okolí, tedy pokud výsledek byl svým obsahem alespoň spojen s API, byl označen jako platný. Na druhé straně jsem pozoroval záběr, jaký CSE má. Tento aspekt jsem hodnotil záběr ve smyslu počtu různých domén obsažených ve vý-

sledcích (sloupce Doménová diverzita). Jednotlivé aspekty byly porovnávány pro různé počty testovaných výsledků.

Tabulka 5.2: Srovnání CSE vyhledávačů při různých konfiguracích

	Poměr API [%]			Doménová diverzita			
CSE \Poč. výsledků	10	20	30	10	20	30	40
WebApis1	50	65	63	2	5	6	8
WebApis2	100	96	92	3	5	5	–
WebApis3	60	35	–	5	7	-	–
WebApis4	100	95	76	3	4	9	13
WebApis5	100	95	86	2	4	9	11

Z výsledků je vidět, že je vhodnější větší počet klíčových slov pro specifikaci CSE. Většinou dochází k lepším výsledkům co se týče oblasti API dokumentací a stoupá i záběr domén, který je obsažen.

V žádné konfiguraci nechyběly ve výsledku giganti jako Google (developers.google.com), Yahoo! (developer.yahoo.com). Ve většině případů byl zahrnut i repozitář ProgrammableWeb pro vyhledávanou kategorii.

5.2 Vyhodnocení klasifikace

V této části porovnám klasifikační kvality algoritmů pro strojové učení a jejich modelů na trénovací sadě dokumentů a následně na „ostrých“ testovacích datech.

Testováno bylo 10 algoritmů (Naive Bayes, Logistic, SMO, SVM, Decision Table, Logit Boost, NB-Tree, J48, Random Forest a Random Tree) na obou datových modelech instancí. Jako míry pro měření jsem zvolil pro možnost porovnání stejné, jaké byly použité ve studiích z rešeršní části. Tedy:

1. Přesnost (ang. *precision*), je pravděpodobnost, že relevantní dokumenty jsou obsaženy ve výsledků. Formálně

$$\text{Precision} = \frac{|\{\text{relevantní dokumenty}\}| \cap |\{\text{dokumenty ve výsledku}\}|}{|\{\text{dokumenty ve výsledku}\}|}$$

2. Úplnost (ang. *recall*), je pravděpodobnost, že dokumenty obsaženy ve výsledku jsou relevantní:

$$\text{Recall} = \frac{|\{\text{relevantní dokumenty}\}| \cap |\{\text{dokumenty ve výsledku}\}|}{|\{\text{relevantní dokumenty}\}|}$$

3. F_1 skóre (někdy také *F-test*, *F-measure*), které je vážený průměr precision a recall:

$$F_1 = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}$$

4. Přesnost vyhodnocení (ang. *accuracy*) je celková míra přesnosti klasifikace algoritmu a jedná poměr správných výsledků a celkového počtu výsledků:

$$\text{Přesnost} = \frac{|\{\text{TP}\}| + |\{\text{TN}\}|}{|\{\text{TP}\}| + |\{\text{FP}\}| + |\{\text{FN}\}| + |\{\text{TN}\}|}$$

   Kde TP jsou pravdivě pozitivní výsledky (ang. *true positives*), TN pravdivě negativní výsledky (ang. *true negatives*), FP falešně pozitivní výsledky (ang. *false positives*) a FN falešně negativní výsledky (ang. *false negatives*).

Aby nedošlo ke zmatení čtenáře v důsledku překladu měr, jsou ve zbytku textu pro míry použity označení stejné, jako v rovnicích.

Ačkoliv tento crawler staví klasifikační model pouze jednou, všiml jsem si, že čas potřebný na vytvoření modelu se řádově liší při různých algoritmech a datových modelech instancí. Proto jsem měřil i čas na vytvoření modelu.

5.2.1 Vyhodnocení na trénovací sadě

Algoritmy a modely jsem nejprve otestoval na trénovacím korpusu z kapitoly Realizace. K vyhodnocení jsem použil nástroj Weka Explorer. Výsledky přesnosti jsou zobrazeny v grafu 5.1. Protože se jedná o modely učení s učitelem, pro které je třeba sestavit trénovací model, rozhodl jsem do sledování zahrnout i čas, který je potřebný pro sestavení modelu. Ačkoliv při použití se trénování v ideláním případě provede pouze jednou pro celý běh crawleru. Výsledky časů pro různé konfigurace klasifikace jsou zobrazeny v grafu 5.2.

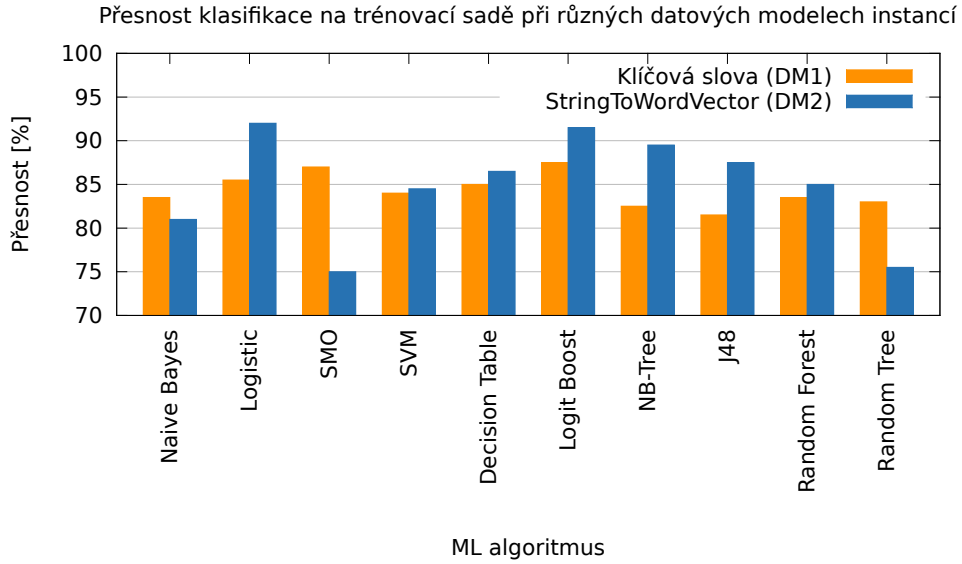
K vyhodnocení byla použita 10-fold vrstvená křížová validace (ang. *stratified cross-validation*), která oproti klasické křížové validaci snižuje výsledný rozptyl.

Při modelování instancí z obsahu klíčových slov (**DM1**) je možné porovnat jejich distribuci a vliv na výslednou třídu. Tato rozložení pro obě sady jsou v příloze C.

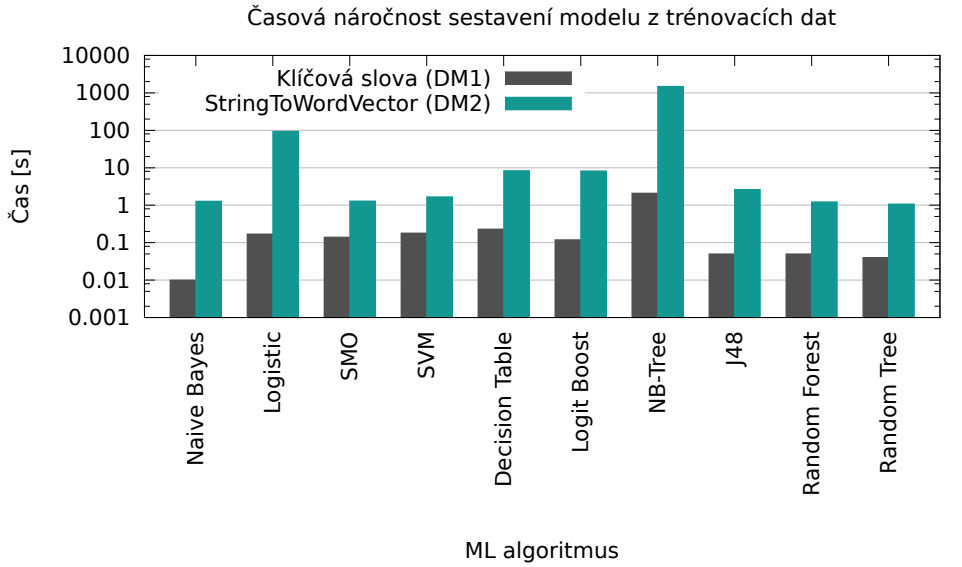
Z naměřených hodnot je patrné, že oba datové modely instancí mají své výhody na určitých klasifikačních algoritmech. Použití filtru StringToWordVector dokáže výsledky některých algortmů zlepšit (např. Logistic o nezanedbatelných 6,5%) a výkon některých naopak zhoršit (např. SMO o 12%). A při aplikování na správný algoritmus je použitím filtru dosaženo srovnatelného výsledku obecně horších algoritmů s výsledky všepoužívaných SVM/SMO.

Z časového hlediska vytvoření modelu naopak jasně použití filtru prohrává oproti ostatním a někdy je až řádově pomalejší (NB-Tree až 1000×).

Tato data je ale třeba brát s lehkou rezervou z důvodů velikostně omezené velikosti trénovací sady a křížové validaci na stejné množině dokumentů. Výsledky se při použití na širší sadě mohou značně lišit.



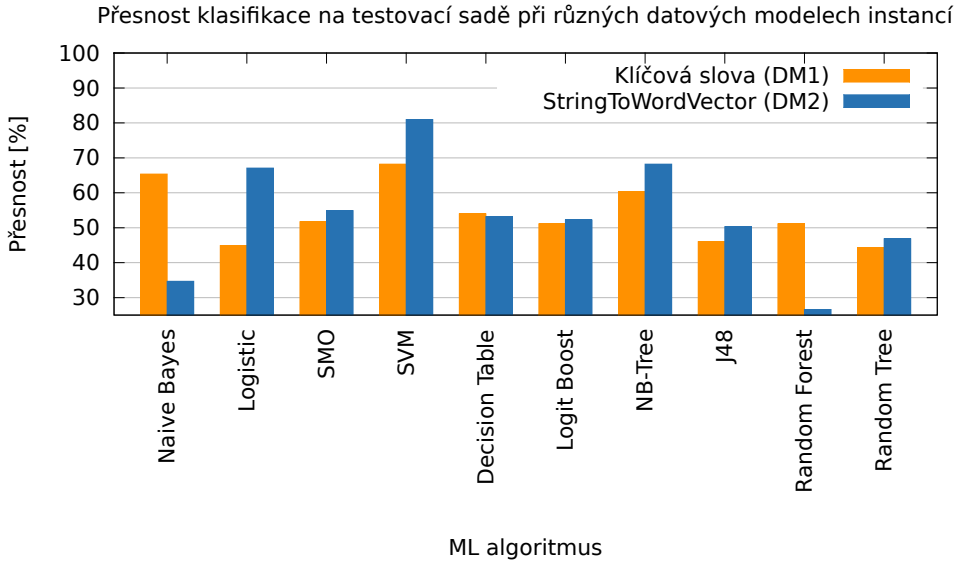
Obrázek 5.1: Přesnost klasifikace na trénovací sadě při různých datových modelech instancí



Obrázek 5.2: Časová náročnost sestavení modelu z trénovacích dat

5.2.2 Vyhodnocení na testovací sadě – crawlerem sesbírané dokumenty

V minulé sekci byly modely otestovány křížovou validací samy na sobě, nyní je ale na čase zjistit, jak si klasifikační část poradí s dokumenty sesbíranými z okolí URL kandidátů. Proto byl crawler spuštěn, aby sesbíral drobnou část webových stránek. Výsledkem byl korpus o 3 628 dokumentech, ze kterých jsem náhodně vybral 6 × 100 dokumentů a ty pak podle svého nejlepšího vědomí a svědomí označil, zda jsou či nejsou Web API dokumentací.



Obrázek 5.3: Přesnost klasifikace na testovací sadě při různých datových modelech instancí

Opět za využití Weka Explorer a výše zmíněného trénovacího datasetu jsem následně otestovat kvality jednotlivých algoritmů na crawlerem sesbíraných dokumentech. Výsledky jsou vidět na grafu 5.3, přesné výsledky pak v tabulce 5.3.

Jak je vidět, tak ačkoliv se algoritmy zdály rovnocené při křížové validaci na trénovacím datasetu, na „skutečných“ datech se objevily hlubší rozdíly. Podle očekávání se přesnost u všech znatelně snížila a někdy dosahuje až podprůměrných výsledků. Naopak nejlépe si počínal SVM při použití filtru StringToWordVector, který dosahuje na 80, 92% přesnosti, což je srovnatelná přesnost, které jsem dosáhl na trénovací sadě a které dosáhli autoři [12] na své testovací sadě.

Tabulka 5.3: Naměřené hodnoty přesnosti ML algoritmů

Algoritmus	DM	Precision	Recall	F1	Přesnost [%]
Naive Bayes	1	0,86	0,65	0,72	65,34
	2	0,84	0,35	0,42	34,68
Logistic	1	0,84	0,45	0,53	44,88
	2	0,86	0,67	0,73	67,05
SMO	1	0,85	0,52	0,60	51,70
	2	0,90	0,55	0,63	54,91
SVM	1	0,85	0,68	0,74	68,18
	2	0,88	0,81	0,84	80,92
Decision table	1	0,82	0,54	0,62	53,97
	2	0,90	0,53	0,61	53,17
Logit Boost	1	0,86	0,51	0,59	51,13
	2	0,89	0,57	0,65	57,22
NB-Tree	1	0,83	0,60	0,68	60,22
	2	0,87	0,68	0,74	68,20
J48	1	0,86	0,46	0,54	46,02
	2	0,85	0,50	0,59	50,28
Random Forest	1	0,83	0,51	0,60	51,13
	2	0,81	0,27	0,32	26,59
Random Tree	1	0,81	0,44	0,53	44,31
	2	0,82	0,47	0,56	46,82

5.3 Vyhodnocení crawlování

Tato část se zaměřuje na kvalitativní testování crawleru. Zejména jak velkou část webu obsáhne, kolik domén je ve výsledcích zahrnuto a o jako poskytovatele se jedná.

Poznámka: Protože se jedná o popis rozsáhlého webového prostoru, který se stále vyvíjí a záleží zejména na dotazu q od uživatele, který silně ovlivňuje výsledky, jsou některé hodnoty uvedné pouze v řádové přesnosti.

Analýza vyspecifikovaného CSE vyhledávače přinesla poznatky, že optimální je využití 30 seedů, které poskytují dostatečnou rozprostřenost URL ve výsledcích, aby byl crawler schopen pokrýt relativně široké spektrum stránek. Tato relativně hluboká hladina výsledků vyhledávače, která je prohledávána by měla crawler navést alespoň na domovskou stránku API. Proto jsem zvolil hloubku crawlování 4. Výsledkem je kompromis, který má za výsledek získání tisíců dokumentů na jednu vyhledávací frázi. Při hloubce 2 a 3 se jedná o stovky, při 4 jde o tisíce a při 5 jsou to už desetitisíce nasbíraných dokumentů. Větší crawlovací hloubku jsem z technických důvodů nebyl schopen zkoumat, nicméně crawler toto nastavení umožňuje.

Testovaná konfigurace crawleru:

```
depth    = 4
cse.cx   = 004173228268281370859:atvecc___7du
num      = 30
```

Při zmíněné konfiguraci crawleru dojde k sesbírání tisíců dokumentů všech kategorií, z nichž je nějakou API dokumentací 10,2%. Crawler dle očekávání probrouzdal jak nezávislé stránky tak repozitáře webových API.

5.3.1 Záběr domén a poskytovatelů

Z výchozích 30 seedů z několika domén (cca 10, viz 5.2) získaných z vyhledávače CSE byl crawler schopen sesbírat tisíce dokumentů z několika stovek domén.

Pro dota q=„social“ tvořily většinový podíl velcí poskytovatelé služeb – Google, Twitter, Tumblr, Yahoo, Flickr, Pinterest, kteří poskytují široké a mezi uživateli rozšířenou nabídku služeb, které svojí povahou dokonce odpovídají dotazu.

Nezanedbatelnou část často tvoří samotné příspěvky uživatelů ze sociálních sítí (Twitter, Youtube), příspěvky v různých diskuzních fórech obsahující ukázky kódu a návody na použití API.

Co mi ve výsledcích chybělo, byly menší poskytovatelé služeb, které se mně buď nepovedlo ve vzorku najít, anebo nebyly ve výsledcích vůbec.

5.3.2 Identifikace problémů a návrhy na zlepšení

Ve výsledcích chyběly menší poskytovatelé služeb. Pro řešení tohoto problému je nejspíše nedostačující použití CSE, který používá index nad omezeným počtem URL vzorů, ve kterých vyhledává. Jediným východiskem se mi jeví použití daleko širšího záběru crawleru, který svými seedy vyjde z daleko širší množiny URL, pro kterou se mi CSE jeví jako omezený. Tento problém by nemuselo vyřešit ani zvýšení hloubky crawlování.

Při větších hloubkách dochází spíše ke crawlování celého webu a slepého pátrání po API dokumentacích, kterých je k nalezení přes specializovaný vyhledávač pouhých 10% a bylo by naivní si myslet, že tak vysoký poměr je aplikovatelný na celý web. Z tohoto hlediska bych se držel stanovené, relativně nízké hloubky crawlování, aby měl bot možnost najít API dokumentaci ze stránek přímo spojených s výsledky z vyhledávače.

Dokumenty, ve kterých se ML algoritmy často pletly, byly často pouhé ukázky kódu, komunikace nebo obecného knihovního API, nejednalo se však o popis webového API. Řešením by mohlo být zaměření se na tento typ dokumentů například v trénovací sadě, nebo je jinak identifikovat a záměrně je označit za nesprávná.

5.4 Shrnutí výsledků

Klasifikace. Pro klasifikační část bylo vybráno a otestováno 10 algoritmů (Naive Bayes, Logistic, SMO, SVM, Decision Table, Logit Boost, NB-Tree, J48, Random Forest a Random Tree), na obou datových modelech instancí (rysy z obsahu klíčových slov a použití Weka filtru StringToWordVector). Sledovány byly časy postavení modelu a výsledky klasifikace měřené ukazateli Precision, Recall, F1 a přesnosti.

V první fázi byly modely otestovány na trénovací sadě dokumentů s poměrem 1:1 Web API dokumentací a ostatních, použita byla 10-fold vrstvená křížová validace. Následně byly modely vystaveny klasifikaci dokumentů sesbíraných crawlerem přímo z webu z okolí výsledků URL kandidátů.

Podle očekávání výsledky na trénovací sadě byly celkově daleko přesnější, než na crawlerem sesbíraných dokumentech. Zatímco na trénovací sadě byly výsledky přesné v rozmezí 75%–92% (nejmenší přenosti dosahoval algoritmus SMO při datovém modelu instance z filtru, nejvyšší naopak alg. Logistic, též při využití filtru), na testovací sadě z crawleru procentuální rozmezí přesnosti klesklo na 28%–81% (nejméně přesným se ukázal alg. Random Forest s filtrem StringToWordVector, naopak nejpřesnějším se ukázal alg. SVM s filtrem).

Je patrné, že datový model instance s použitím filtru StringToWordVector přináší extrémní výsledky v obou směrech (má na svědomí nejlepší, ale i nejhorší výsledky). V celkovém pohledu se ale dá pozorovat, že spíše přesnost zvyšuje. Při trénovací sadě zvýšil přesnost u 7/10 algoritmů a nejlepší zlepšení bylo pozorováno u algoritmu Logistic o 6,5%, na testovací taktéž 7/10, a nejlepší zlepšení opět přinesl u algoritmu Logistic a to o 12%.

Napříč všemi algoritmy bylo dosaženo na trénovací sadě průměrné přesnosti 84%, na testovací sadě pak pouhých 53% a to bez ohledu na datový model instance. 53% není vysoké číslo, nicméně jedná se o experimentální vědu a cílem bylo najít nejvhodnější ML algoritmus pro tento problém. Na trénovací sadě byly rozdíly méně znatelné, na „ostrých“ datech se nejvíce osvědčila kombinace algoritmu SVM a použití filtru při modelování reprezentace instance. Tato kombinace jako jediná na testovací sadě dosahovala přesnosti vyšší 80%, pak následovala následovala kategorie až v rozsahu 60–70%, do které se vešly Naive Bayes s DM1, Logistic s DM2, SVM s DM1 a NB-Tree při libovolné reprezentaci instance.

Crawling. Výsledky části crawlování velmi záleží na nastavení jednotlivých parametrů crawleru, specifikaci CSE vyhledávače a hledané fráze. Při použití nižších hodnot u parametrů hloubky crawlování a top-N výsledků z vehledávače, jako jsem použil při testování crawleru, má za následek prohledání drobné části webového prostoru – při hloubce 4 a 30 URL kandidátů bylo nashromážděno přes 3,5 tis. dokumentů z více jak 400 domén, ze kterých bylo jako API vyhodnoceno 8% jako Web API.

Výsledky, shromážděné při této konfiguraci, obsahovaly zejména větší poskytovatele služeb, jako jsou Google, Twitter, Tumblr, Yahoo!, Flickr nebo

Pinterest, stejně tak jako občasné příspěvky samotných uživatelů ze sociálních sítí a blogů. V testovaném vzorku se nepodařilo najít menší poskytovatele, nicméně při změně konfiguračních parametrů crawleru a jeho přiblížení ke crawlování celého webu, se pak ani tyto menší cíle nalezení nevyhnou (pokud jsou indexované vyhledávačem či na ně vedou odkazy).

Závěr

Současná situace při hledání správného webového API a jeho dokumentace začíná pokulhávat kvůli nedostatečně rozšířeným a obecně použitelným standardům ohledně jejich popisu a prezentace. Toto se snaží suplovat veřejné API repoziráře jako ProgrammableWeb, které shromažďují informace o webových API a jejich dokumentacích. Obsah repozitáře je ale spravován lidmi a bez průběžných aktualizací a ověřování vložených záznamů, proto vzinkají tendence tento proce automatizovat. Dosavadní pokusy byly zaměřeny prozatím na identifikaci dokumentací webových API, které vycházely z lidmi sesbírané sady dokumentů zajištěné z ProgrammableWeb, která pak sloužila pro natrénovaní klasifikačních algoritmů. Je znám jediný zaměřený crawler, který vychází ze záznamů repozitáře a prohledává sousední URL, pro identifikaci taktéž používá supervised-learning modely.

Tato diplomová práce se zabývá řešením automatického nalezení a identifikace Web API dokumentací nezávisle na veřejných repozitářích. Základem je crawler, jehož vstupním parametrem je vyhledávací fráze, podle které se vyhledávačem naleznou relevantní dokumenty, které identifikují oblasti zájmu, které crawler prohledává podrobněji. Všechny sesbírané dokumenty jsou následně klasifikovány určeným algoritmem strojového učení a výsledky jsou pak dostupné uživateli.

Výsledný stroj [28] je napsán v jazyce Java, je postaven na open-source crawleru Apache Nutch, používá Apache Solr jako vyhledávací a indexovací platformou pro nalezené dokumenty, kandidátní URL pro počáteční množinu seedu crawleru jsou získány ze zmíněného vyhledávače Google CSE. Klasifikační část používá pomocné struktury a implementované algoritmy software Weka.

Základem klasifikační části crawleru je reprezentace instancí, a to ve dvojí podobě. Buďto binárním vektorem, kde i-tá položka je 1, pokud četnost i-tého klíčového slova přesahuje prahovou hodnotu (zvolil jsem 3), jinak 0. Jako klíčová slova byla použita množina *api, method, operation, input, output, parameter, get, post, put, delete, append, url, data, request, http, https, default,*

xml, json, example, response, description, required. Druhý způsob reprezentace je založen na reprezentaci dokumentu jako bag-of-words z obsahu.

Pro odstranění fixace na repozitáře webových API byly jako zdroje pro počátek crawlování zvoleny výsledky z customizovaného vyhledávače Google Custom Search Engine, založeného na klíčových slovech *api, rest, http, json, xml, get, post, documentation*, který limitoval indexovaný web na zúžený seznam domén.

Výsledky testování ukázaly, že klasifikační část dosahuje největší přesnosti na nacrawlovaných dokumentech při použití algoritmu SVM, který na datovém modelu instance z klíčových slov určí přesně 69% instancí a s použitím filtru až 81%. Což je srovnatelný výsledek s referenčími pracemi, které ale nepracovaly s dokumenty nacházejícimi se libovolně na webu. Crawlovací část projde na jeden záběr při hloubce 4 s 30 počátečními URL kandidáty z vyhledávače přes 3,5 tis. různých URL, jejichž obsah odpovídá uživatelem zadané frázi. Při této konfiguraci je ve výsledku zahrnuto přes 400 domén, které zahrnují větší a středně velké poskytovatele služeb.

Budoucí práce

Při realizaci a testování jsem zjistil několik nedostatků a nápadů na vylepšení nástroje. Ve výsledcích byly spolehlivě zahrnuti větší poskytovatelé služeb, avšak malý počet menších. K odbourání tohoto nedostatku by stála za zvážení volba alternativního zdroje pro počáteční seznam URL kandidátů, případně přeorientovat crawler na sběr API dokumentací z celého webu, zatímco nyní je vázán na rozsah CSE vyhledávače. Dokumenty by mohly být filtrovány a skórovány již ve crawl-frontier crawleru pomocí heuristiky založené na klíčových slovech, nebo obdobně rychlou metodou, aby nedocházelo ke sbírání celého obsahu webu. Klasifikační část by zajisté mohla lépe využít rozsáhlejší trénovací dataset. Též by si vyžadovala podrobnější testování, které by experimentovalo s různými nastaveními algoritmů a filtrů, které by mohly přispět k ještě lepším výsledkům, než jaké byly v této práci představeny.

Literatura

[1] ProgrammableWeb: *ProgrammableWeb - APIs, Mashups and the Web as Platform [online].* [přístup 2015-03-15]. Dostupné z: `http://www.programmableweb.com/`

[2] Castillo, C.: *Effective Web Crawling.* Dizertační práce, University of Chile, 2004.

[3] Gulli, A.; Signorini, A.: The Indexable Web is More than 11.5 Billion Pages. *WWW '05 Special interest tracks and posters of the 14th international conference on World Wide Web*, 2005: s. 902–903.

[4] de Kunder, M.: *The Size of the Wolrd Wide Web (The Internet) [online].* [přístup 2015-03-15]. Dostupné z: `www.worldwidewebsize.com`

[5] Maleshkova, M.; Pedrinaci, C.; Dominigue, J.: Investigating Web APIs on the World Wide Web. *IEEE 8th European Conference on Web Services (ECOWS)*, 2010: s. 107–114.

[6] Fielding, R. T.: *Representational State Trasnfer (REST).* Dizertační práce, University of California, Irvine, 2000.

[7] WWW Consorcium: *SOAP Version 1.2 Messaging framework [online].* [přístup 2015-04-25]. Dostupné z: `http://www.w3.org/TR/soap12/`

[8] WWW Consorcium: *Web Services Description Language (WSDL) 1.1 [online].* [přístup 2015-04-25]. Dostupné z: `http://www.w3.org/TR/wsdl`

[9] WWW Consorcium: *Web Application Description Language [online].* [přístup 2015-04-27]. Dostupné z: `http://www.w3.org/Submission/wadl/`

[10] Kopecky, J.; Vitvar, T.; Pedrinaci, C.; aj.: RESTful Services with Lightweight Machine-readable Descriptions and Semantic Annotations. In *REST: From Research to Practice*, Springer New York, 2011, s. 473–506.

[11] Stylos, J.; Myers, B. A.: Mica: A Web-Search Tool for Finding API Components and Examples. *VL/HCC 2006. IEEE Symposium on Visual Languages and Human-Centric Computing*, 2006: s. 195–202.

[12] Pedrinaci, C.; Liu, D.; Chenghua; aj.: Harnessing the Crowds for Automating the Identification of Web APIs. *AAAI Spring Symposium: Intelligent Web Services Meet Social Computing*, 2012.

[13] Pedrinaci, C.: *Harnessing the Crowds for Automating the Identification of Web APIs [online]*. Knowledge Media Institute, 2012, [přístup 2015-03-30]. Dostupné z: `http://www.slideshare.net/cpedrinaci/harnessing-the-crowds-for-automating-the-identification-of-web-apis`

[14] Steinmetz, N.; Lausen, H.; Brunner, M.: Web Service Search on Large Scale. In *Service-Oriented Computing*, Lecture Notes in Computer Science, Springer-Verlag Berlin Heidelberg, 2009, s. 437–444.

[15] Steinmetz, N.; Lausen, H.; Brunner, M.; aj.: D5.1.3 – second crawling prototype. 2009.

[16] Lin, C.; He, Y.; Pedrinaci, C.; aj.: Feature LDA: a Supervised Topic Model for Automatic Detection of Web API Documentations from the Web. In *The Semantic Web – ISWC 2012*, Springer Berling Heidelberg, 2012, s. 328–343.

[17] Blei, D. M.; Ng, A. Y.; Jordan, M. I.: Latent Dirichlet Allocation. *Journal of Machine Learning Research*, ročník 3, 2003.

[18] Redondo-García, J. L.; Hildebrand, M.; Romero, L. P.; aj.: Augmenting TV Newscasts via Entity Expansion. In *ESWC Satellite Events*, Springer International Publishing Switzerland, 2014, s. 472–476.

[19] Machine Learning Group at the University of Waikato: *Weka [online]*. 2015, [přístup 2015-03-22]. Dostupné z: `http://www.cs.waikato.ac.nz/`

[20] Witten, I. H.; Frank, E.; Hall, M. A.: *Data Mining – Practical Machine Learning Tools and Techniques*. Elsevier, třetí vydání, 2011, ISBN 978-0-12-374856-0.

[21] Sun Microsystems: *The Java Language Environment [online]*. 1997, [přístup 2015-04-24]. Dostupné z: `http://www.oracle.com/technetwork/java/intro-141325.html`

[22] The Apache Software Foundation: *Apache Nutch – highly extensible and scalable open source web crawler software project [online]*. [přístup 2015-03-21]. Dostupné z: `http://nutch.apache.org`

[23] Shaikh, A. F.; Laliwala, Z.: *Web crawling and Data Mining with Apache Nutch*. Packt publishing, 2013, ISBN 978-1-78328-685-0.

[24] Białecki, A.: *Nutch as a Web data mining platform [online]*. Apache Foundation, 2010, [přístup 2015-03-21]. Dostupné z: `http://www.slideshare.net/abial/nutch-as-a-web-data-mining-platform`

[25] Nagel, S.: *Nutch Wiki: About Plugins [online]*. 2014, [přístup 2014-03-21]. Dostupné z: `https://wiki.apache.org/nutch/AboutPlugins`

[26] Machine Learning Group at the University of Waikato: *Attribute-Relation File Format (ARFF) [online]*. 2008, [přístup 2015-03-22]. Dostupné z: `http://www.cs.waikato.ac.nz/ml/weka/arff.html`

[27] SeleniumHQ: *Selenium – Web Browser Automation [online]*. 2015, [přístup 2015-03-31]. Dostupné z: `http://www.seleniumhq.org/`

[28] Šmolík, J.: *Wapicrawler [online]*. 2015, [přístup 2015-04-24]. Dostupné z: `https://bitbucket.org/smoliji/wapicrawler`

[29] Nagel, S.: *Nutch as a Web data mining platform [online]*. ApacheCon EU, 2014, [přístup 2015-03-21]. Dostupné z: `http://events.linuxfoundation.org/sites/events/files/slides/aceu2014-snagel-web-crawling-nutch.pdf`

[30] The Apache Software Foundation: *Apache Solr [online]*. 2014, [přístup 2015-03-21]. Dostupné z: `http://lucene.apache.org/solr/`

[31] Bennet, M.: *Solr. SchemaXML [online]*. 2014, [přístup 2015-03-21]. Dostupné z: `https://wiki.apache.org/solr/SchemaXml`

[32] Tan, K.: *Discover the Solr Search Server [online]*. 2015, [přístup 2015-03-21]. Dostupné z: `http://www.solrtutorial.com`

[33] Apache Software Foundation: *Apache Solr Reference Guide [online]*. 2015, [přístup 2015-03-22]. Dostupné z: `https://cwiki.apache.org/confluence/display/solr/`

[34] Google: *Custom Search – Using REST to Invoke the API [online]*. 2015, [přístup 2015-03-30]. Dostupné z: `https://developers.google.com/custom-search/json-api/v1/using_rest`

[35] Google: *Google Custom Search Engine [online]*. 2015, [přístup 2015-04-24]. Dostupné z: `https://cse.google.com/cse/`

Příloha **A**

Seznam použitých zkratek

API Application programming interface

CRUD Create, read, update, delete

CSE Custom search engine. Myšlena služba *Google Custom Search Engine*

CSV Comma separated values

DB Database

feaLDA feature LDA

GUI Graphical user interface

HATEOAS Hypermedia as the engine of application state

HTML Hypertext markup language

HTTP Hypertext transfer protocol

IEEE Institute of electrical and electronics engineers

JAR Java archive

JSON Javascript object notation

JVM Java virtual machine

LDA Latent Dirichlet allocation

Mica Making interfaces clear and accessible

ML Machine learning

NLP Natural language processing

OOP Object-oriented programming

PDF Portable document format

RDF Resource description framework

REST Representational state transfer

SMO Sequential minimal optimization

SPARQL SPARQL protocol and RDF query language

SVM Support vector machine

UDDI Universal description, discovery and integration

URI Uniform resource identifier

URL Uniform resource locator

W3C World wide web Consortium

Weka Waikato environment for knowledge analysis

WSDL Web service description language

WWW World wide web

XML Extensible markup language

Příloha **B**

Ukázka komunikace s CSE

Ukázka požadavku a odpovědi z komunikace s CSE vyhledávačem.

B.1 HTTP Požadavek

Ukázkový HTTP požadavek na REST API CSE vyhledávače, kde `key` je API klíč identifikující uživatele služeb Google, `cx` je identifikátor CSE, `start` je počáteční stránka, `num` je počet položek na stránce (max. 10) a `q` je dotaz.

```
GET /customsearch/v1
    ?key=AIzaSyAT1M2PUBltxCXPWwsBbuNSIso9UbOJNVg
    &cx=004173228268281370859:atvecc__7du
    &start=1
    &num=1
    &q=social HTTP/1.1
Host: www.googleapis.com
```

B.2 Odpověď ve formátu JSON

Ovpověď vypadá takto:

```
{
 "kind": "customsearch#search",
 "url": {
  "type": "application/json",
  "template": "https://www.googleapis.com/customsearch/v1?q={searchTerms}
    &num={count?}&start={startIndex?}&lr={language?}&safe={safe?}
    &cx={cx?}&cref={cref?}&sort={sort?}&filter={filter?}&gl={gl?}
    &cr={cr?}&googlehost={googleHost?}&c2coff={disableCnTwTranslation?}
    &hq={hq?}&hl={hl?}&siteSearch={siteSearch?}
    &siteSearchFilter={siteSearchFilter?}&exactTerms={exactTerms?}
    &excludeTerms={excludeTerms?}&linkSite={linkSite?}&orTerms={orTerms?}
    &relatedSite={relatedSite?}&dateRestrict={dateRestrict?}&lowRange={lowRange?}
    &highRange={highRange?}&searchType={searchType}&fileType={fileType?}
    &rights={rights?}&imgSize={imgSize?}&imgType={imgType?}
    &imgColorType={imgColorType?}&imgDominantColor={imgDominantColor?}&alt=json"
 },
 "queries": {
```

```
  "nextPage": [
   {
    "title": "Google Custom Search - social",
    "totalResults": "25",
    "searchTerms": "social",
    "count": 1,
    "startIndex": 2,
    "inputEncoding": "utf8",
    "outputEncoding": "utf8",
    "safe": "off",
    "cx": "013315504628135767172:d6shbtxu-uo"
   }
  ],
  "request": [
   {
    "title": "Google Custom Search - social",
    "totalResults": "25",
    "searchTerms": "social",
    "count": 1,
    "startIndex": 1,
    "inputEncoding": "utf8",
    "outputEncoding": "utf8",
    "safe": "off",
    "cx": "013315504628135767172:d6shbtxu-uo"
   }
  ]
 },
 "context": {
  "title": "CSE",
  "facets": [
   [
    {
     "label": "blogs",
     "anchor": "Blogs",
     "label_with_op": "more:blogs"
    }
   ],
   [
    {
     "label": "docs",
     "anchor": "Docs",
     "label_with_op": "more:docs"
    }
   ],
   [
    {
     "label": "help",
     "anchor": "Help",
     "label_with_op": "more:help"
    }
   ],
   [
    {
     "label": "forum",
     "anchor": "Forum",
     "label_with_op": "more:forum"
    }
   ]
  ]
 },
 "searchInformation": {
  "searchTime": 0.186664,
  "formattedSearchTime": "0.19",
  "totalResults": "25",
  "formattedTotalResults": "25"
 },
 "items": [
  {
   "kind": "customsearch#result",
   "title": "Custom Search vs Google.com - Custom Search Help",
   "htmlTitle": "Custom Search vs Google.com - Custom Search Help",
   "link": "https://support.google.com/customsearch/answer/70392?hl=en",
   "displayLink": "support.google.com",
   "snippet": "Your custom search engine doesn't include Google Web Search features such as
```

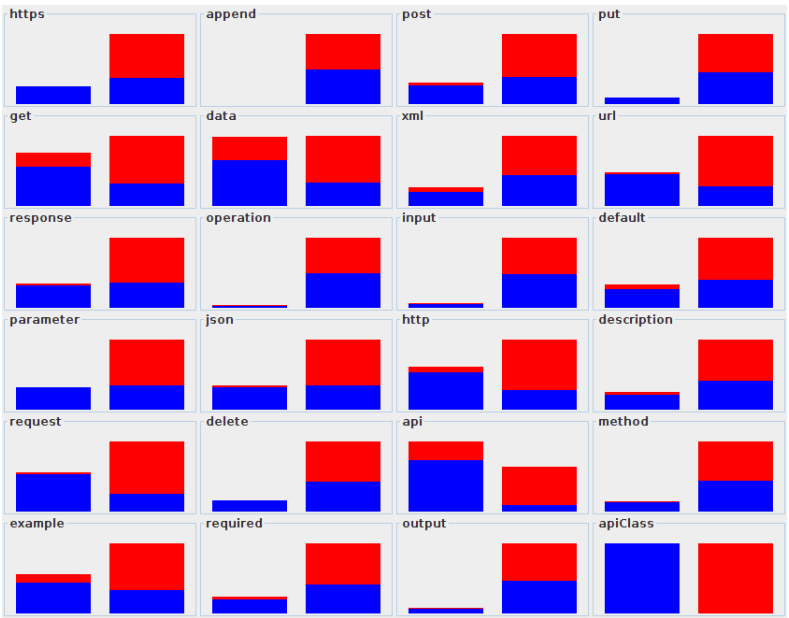
```
        \nOneboxes, real-time results, universal search, social feaures, or personalized ...",
    "htmlSnippet": "Your custom search engine doesn&#39;t include Google Web Search features
        such as \u003cbr\u003e\nOneboxes, real-time results, universal search,
        \u003cb\u003esocial\u003c/b\u003e feaures, or personalized ...",
    "cacheId": "u8vjSnbnFLEJ",
    "formattedUrl": "https://support.google.com/customsearch/answer/70392?hl=en",
    "htmlFormattedUrl": "https://support.google.com/customsearch/answer/70392?hl=en",
    "pagemap": {
     "metatags": [
      {
       "viewport": "width=device-width,initial-scale=1,maximum-scale=1,user-scalable=no"
      }
     ],
     "breadcrumb": [
      {
       "url": "Custom Search Help",
       "title": "Custom Search Help"
      }
     ]
    },
    "labels": [
     {
      "name": "help",
      "displayName": "Help",
      "label_with_op": "more:help"
     }
    ]
   }
  ]
}
```

Příloha C

Vliv klíčových slov na třídu API

Obrázky z aplikace Weka Explorer ukazující rozložení zjištěných klíčových slov pro API dokumentace (viz 4.2.2) mezi třídami *yes – Api dokumentace* a *no – ostatní*.

Vždy levý sloupec ukazuje množství dokumentů $\phi(d, w, t) = 0$ a pravý $\phi(d, w, t) = 1$ a červený barva značí hodnotu třídy *no* a modrá hodnotu *yes*.



Obrázek C.1: Vliv zjištěných klíčových slov na klasifikační třídu API dokumentace (trénovací sada)

Obrázek C.2: Vliv zjištěných klíčových slov na klasifikační třídu API dokumentace (nacrawlovaná sada)

Příloha D

Obsah přiloženého CD

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