Katedra informatiky
Přírodovědecká fakulta
Univerzita Palackého v Olomouci

DIPLOMOVÁ PRÁCE

Klasifikace autorství

Srovnání přístupů a algoritmů pro posuzování autorství

2015

Michal Peřinka

Vedoucí práce: Jan Konečný

Studijní obor: Informatika, prezenční forma

Bibliografické údaje

Autor:	Michal Peřinka
Název práce:	Klasifikace autorství (Srovnání přístupů a algoritmů pro posuzování autorství)
Typ práce:	diplomová práce
Pracoviště:	Katedra informatiky, Přírodovědecká fakulta, Univerzita Palackého v Olomouci
Rok obhajoby:	2015
Studijní obor:	Informatika, prezenční forma
Vedoucí práce:	Jan Konečný
Počet stran:	68
Přílohy:	1 CD/DVD
Jazyk práce:	český

Bibliograhic info

Author:	Michal Peřinka
Title:	Authorship classification (Comparison of ways and algorithms in authorship classification)
Thesis type:	master thesis
Department:	Department of Computer Science, Faculty of Science, Palacký University Olomouc
Year of defense:	2015
Study field:	Computer Science, full-time form
Supervisor:	Jan Konečný
Page count:	68
Supplements:	1 CD/DVD
Thesis language:	Czech

Anotace

V dnešní době obrovského množství informací je třeba tyto informace třídit a zpracovávat automatizací. Služby jako internet přináší k uživateli mnoho textů, na jejichž autenticitu se nedá spolehnout, a tudíž se uživatel nemůže spolehnout ani na informace v textu obsažené. Proto je třeba takové texty verifikovat. Jedním z faktorů spolehlivosti je znalost pravého autora textu. Proto se v této práci věnuji možnosti automatizovaného přiřazení autora k textu, případně odhalení nepravosti udávaného autora. Implementuji metodu posouzení autorství na základě vybraných lexikálních vlastností textů, metodu založenou na rozdílu velikosti při komprimaci a metodu založenou na kvantifikaci kořenů slov. Dále implementuji metodu diskriminativního syntaktického stromu, která má největší úspěšnost v nově tvořené kategorii metod založených na vzorech obsažených v syntaktických stromech. Tyto metody založené na různých paradigmatech popisuji v textu a rozebírám na nich základní vlastnosti a principy při analýze textů. Taktéž rozebírám obecné vlastnosti textu, které ovlivňují rozbor a přiřazení autora.

Synopsis

In this age of enourmous amount of information we need to verify and sort these infromation automatically. Services such as internet provide users with many texts. Authenticity of such texts is unreliable thus users can not rely on contained information either. That is why we need to verify such texts. One of the reliability factors is knowledge of original author. Because of that I deal with possibility of automation text - author attribution or uncovering fake author. I implement a method of authorship attribution based on selected lexical text features, a method based on size difference of compressed texts and a method based on word stems quantification. Further, I implement a method of discriminative syntactic tree, which is most successful in new category of methods based on patterns contained in syntactic trees. All these methods based on different paradigms I describe in the thesis text and I examine fundamental features and principles of text analysis. I also examine general text features, which affect text analysis and author attribution.

Klíčová slova: klasifikace autorství; lingvistické vlastnosti textu; přiřazení autorství; větný rozbor

Keywords: authorship classification; lingvistic features; authorship attribution; sentence analysis

Děkuji vedoucímu své diplomové práce Janu Konečnému za vstřícný a věcný postoj k mé činnosti. Také děkuji své rodině a blízkým za trpělivost ve dnech hektických.

Místopřísežně prohlašuji, že jsem celou práci včetně příloh vypracoval/a samostatně a za použití pouze zdrojů citovaných v textu práce a uvedených v seznamu literatury.

datum odevzdání práce podpis autora

Obsah

1 Popis problematiky klasifikace autorství **9**
- 1.1 Historie . . . 9
- 1.2 Obory využití klasifikace textu . . . 9
- 1.3 Stylometrické atributy . . . 10
- 1.4 Základní kategorie stylometrických atributů . . . 11
- 1.5 Metody pro přiřazení autora . . . 13
- 1.6 Komprimační metoda . . . 16
  - 1.6.1 Obecný popis . . . 16
  - 1.6.2 Výběr algoritmu . . . 16
  - 1.6.3 Výhody metody . . . 17
- 1.7 SVM - Support Vector Machine . . . 17
  - 1.7.1 Obecný popis . . . 17
- 1.8 SVM s kořeny slov . . . 19
- 1.9 Naivní metoda . . . 22
  - 1.9.1 Obecný popis . . . 22
  - 1.9.2 výpočet . . . 23
- 1.10 Metoda diskriminačního syntaktického stromu . . . 24
  - 1.10.1 Úvod . . . 24
  - 1.10.2 Odbočka k NLP a porovnávaným metodám . . . 25
  - 1.10.3 Popis struktury a příklad . . . 26
  - 1.10.4 Teoretické srovnání a problémy . . . 27
  - 1.10.5 Vlastnosti vhodného syntaktického stromu . . . 28
  - 1.10.6 Konstrukce vhodného syntaktického stromu . . . 29
  - 1.10.7 Vlastnosti k-ee stromu vzhledem k obsaženým informacím 30
  - 1.10.8 Upravená metoda sekvenčního pokrytí . . . 34
  - 1.10.9 Přímé získání k-ee stromů . . . 34
- 1.11 Náhled na filologické vlastnosti textu . . . 35
  - 1.11.1 Typické zdroje . . . 36
  - 1.11.2 Nedostatky typických zdrojů . . . 36

2 Řešení a návrh implementace **37**
- 2.1 Implementace . . . 37
  - 2.1.1 Platforma . . . 37
  - 2.1.2 Vlastní triviální metoda . . . 38
  - 2.1.3 Kompresní metoda . . . 40
  - 2.1.4 SVM s kořeny slov . . . 41
  - 2.1.5 Metoda diskriminačního syntaktického stromu . . . 42

3 Výsledky měření **43**
- 3.1 Měření . . . 43
  - 3.1.1 3 autoři . . . 45
  - 3.1.2 4 autoři . . . 46
- 3.2 Diskuze . . . 47

4 Uživatelská příručka **48**
4.1 Soubory . . . 48
4.2 Práce s aplikací . . . 49

5 Programátorská příručka **51**
5.1 AddAuthor . . . 51
5.2 Třída Article . . . 52
5.3 Třída Author . . . 52
5.4 Třída Compression_p . . . 53
5.5 Třída Dstma_p . . . 53
5.6 Form MainW . . . 55
5.7 Třída methodCenter . . . 56
5.8 Třída naive_p . . . 57
5.9 Form result . . . 58
5.10 Form stats . . . 58
5.11 Třída strDb . . . 58
5.12 Třída strint . . . 59
5.13 Třída SVM_tools . . . 59
5.14 Třída SVM_p . . . 60
5.15 Třída syntNode . . . 61
5.16 Třída syntTree . . . 61
5.17 Interface ISVM . . . 62

Závěr **63**

Conclusions **64**

A Obsah přiloženého CD/DVD **65**

Literatura **66**

Seznam obrázků

1 Lineární SVM. 18
2 Příklad syntaktického stromu. 26
3 Příklad upraveného syntaktického stromu. 27
4 Příklad Pattern-Growth přístupu. 29
5 Ukázka stavby vzorů 1. 30
6 Ukázka stavby vzorů 2. 31
7 Ukázka stavby vzorů 3. 31
8 Binned information gain 32
9 Příručka - program po přidání autorů 49
10 Příručka - vlastnosti textu 50
11 Příručka - nabídka možností 50
12 Příručka - přidání autora 51

Seznam tabulek

1 Příklady jader pro SVM. 19
2 Symboly pro algoritmus 35
3 Výsledky měření pro autora A. 45
4 Výsledky měření pro autora B. 45
5 Výsledky měření pro autora C. 45
6 Výsledky měření pro 3 autory v tabulce. 45
7 Výsledky měření pro autora A. 46
8 Výsledky měření pro autora B. 46
9 Výsledky měření pro autora C. 46
10 Výsledky měření pro autora D. 46
11 Výsledky měření pro 4 autory v tabulce. 47

Seznam vět

1 Věta (Indukovaný strom) 28
2 Věta (Vložená hrana) 28
3 Věta (Frekvence výskytu vzoru) 29
4 Věta (Binned conditional entropy a Binned information gain) . . . 30
5 Lemma (Vzory) 32
Důkaz 32
6 Lemma (Distribuce frekvencí) 33
Důkaz 33
7 Věta (Existence meze) 33
Důkaz 33
8 Lemma (IG stromové struktury) 34
Důkaz 34

9 Věta (Branch-and-Bound ořezání) 34
Důkaz 34
10 Věta (Branch-and-Bound ořezání) 35
Důkaz 35

Seznam zdrojových kódů

1 Důležité atributy třídy Author 52
2 Důležité metody třídy Author 53
3 Důležité atributy třídy Compression_p 53
4 Důležité metody třídy Compression_p 53
5 Důležité atributy třídy Dstma_p 54
6 Důležité metody třídy Dstma_p 54
7 Důležité atributy třídy MainW 55
8 Důležité metody třídy MainW 55
9 Důležité atributy třídy methodCenter 56
10 Důležité metody třídy methodCenter 56
11 Důležité atributy třídy naive_p 57
12 Důležité metody třídy naive_p 58
13 Důležité atributy třídy SVM_tools 59
14 Důležité metody třídy SVM_tools 59
15 Důležité atributy třídy SVM_p 60
16 Důležité metody třídy SVM_p 60
17 Důležité atributy třídy syntNode 61
18 Důležité metody třídy syntNode 61
19 Důležité atributy třídy syntTree 61
20 Důležité metody třídy syntTree 62
21 Důležité metody interface ISVM 62

1 Popis problematiky klasifikace autorství

V první kapitole se věnuji úvodu do problematiky zpracování klasifikace autorství. Dále popisuji obecně druhy metod pro analýzu textu a některé konkrétní významné metody, které implementuji a srovnávám v dalších částech textu.

1.1 Historie

První vědecké metody k určení autorství neznámého (anonymního) textu byly vyvíjeny na počátku 19. století. Tehdy však spíše než o přiřazení autora šlo o vyvrácení původu daného textu, zvláště časového období jeho vzniku. U nás to byl například známý Rukopis Zelenohorský, vydávaný za dílo spadající datem svého vzniku do období počátku středověku. Chemickým rozborem však bylo prokázáno, že jde o falzum. Pokud však pracujeme s texty v digitální podobě, musíme se zaměřit buď na metadata textů nebo přímo na posuzování obsahu. Metadata lze poměrně snadno podvrhnout. Proto stojí za pozornost vývoj metod zabývajících se stylistikou a obsahem textu. První takové metody vznikaly již dlouho před prvním digitálním záznamem.

V roce 1964 byly použity nové metody pro určení autora takzvaných Federálních listů (The Federalist Papers) [16]. Tyto listy jsou souborem 77 esejí z let 1787 a 1788, které vyšly v novinách The Independent Journal a The New York Packet. Tyto eseje byly podepsány všechny shodně jménem Publius a vyjadřovaly se k nové americké ústavě. Na základě větného syntaktického rozboru byli určeni autoři Alexander Hamilton, James Madison a John Jay. Metody syntaktického rozboru věty pak získaly dominanci při posuzování autorství a plagiátorství. Ačkoliv jde o jeden z prvních známých případů, kdy byl vyhledáván autor textu na základě lingvistiky, některé poznatky z tohoto případu se využívají dodnes.

1.2 Obory využití klasifikace textu

Původní využití vzešlo z potřeby určit autora historického literárního textu. Takto například po rozboru Shakespearových her vznikly teorie, že jeho díla ve skutečnosti napsalo více autorů, nebo teorie, že Shakespeare hry vydával pod svým jménem, ale psal je pro něj někdo jiný [17]. První pochybnosti o Shakespearově autorství jsou již z roku 1887, pro testování takovéto specifické třídy textů jsou však potřeba odpovídající množiny srovnávacích materiálů. Problematika autorství her od Williama Shakespeara tak zůstává na úrovni konspiračních teorií. Naopak případ federálních listů ukazuje, že i zpětná analýza může být obecně přijata, je-li podpořena dobrou metodikou.

Již brzy se tyto nově vzniklé metody přenesly na aktuální texty a začaly se používat například pro identifikaci teroristů z anonymních prohlášení, autorů blogů, odhalování pachatelů kriminální činnosti z žádostí o výkupné nebo potvrzování autenticity sebevražedných zpráv [22] [23]. V civilním právu jde o ověřování autorství například při publikování knih, článků a vědeckých prací. V informatice

pak lze podle rozboru zdrojového kódu rozeznat autora viru nebo jiného škodlivého softwaru.

Mezi základní cíle při verifikaci textu mohou patřit: [4]

- Zjištění autora.
- Ověření autora.
- Vytvoření profilu autora a jeho charakteristiky.
- Detekce nekonzistentního textu.
- Zjištění žánru textu.

V této práci se budu některým z těchto témat věnovat. Například zjištění autora určitého textu se obvykle provádí porovnáním vlastností přiřazovaného textu a vlastností textů známých autorů. Porovnání musí proběhnout mezi všemi vybranými autory. Autorovi s největší mírou shody je pak text přiřazen.

Ověření autora probíhá porovnáním daného textu a souboru dalších ověřených textů autora. Pokud se text od souboru textů autora liší stylisticky více, než je stanovená přijatelná mez (daná algoritmem či parametry posuzovatele), pak je autor textu posouzen jako odlišný od autora ostatních textů. Dochází tedy mimo jiné k vytvoření stylistického profilu autora. Stejným způsobem můžeme odhalit také plagiátorství.

Nekonzistence v textu nastává zejména v případě, kdy text netvoří jediný autor nebo kdy autor určitou část textu (volně) přepisuje z textu jiného. Hledání takové nekonzistence je z pohledu automatizace nejnáročnější, protože je nutné text profilovat průběžně po částech. Pokud se střídají části kopírované a autentické často a po malých částech, je odhalení automatizací téměř nemožné. Text jako takový je však nekonzistentní s celkovou tvorbou autora.

Další úkol pro automatizaci může být přiřazení žánru, například pokud vyžadujeme určení webové stránky jako pornografické pro rodičovskou kontrolu. Zatímco pro člověka je přiřadit text k nějakému žánru poměrně jednoduchý úkol, algoritmicky je určování žánru složitější. Problém se řeší podobně jako přiřazování autorství textů. Nejdříve shromáždíme soubor textů, u kterých žánr známe. Poté tyto texty zanalyzujeme. Porovnávaný text patří právě tomuto žánru, pokud má podobné vlastnosti jako zbývající texty. Taková analýza je tím snazší, čím přesněji lze žánr odlišit od jiných žánrů. Neboli čím je žánr specifičtější svým slohem nebo lexikální vybaveností, tím snadněji bude při analýze taková vlastnost identifikována. U zmiňované pornografie to jsou například frekvence výskytu slov s explicitním významem.

1.3 Stylometrické atributy

Pro stylometrickou analýzu textu se využívá obecných lingvistických poznatků. Je potřeba definovat určité jevy ve větě, identifikovat je při větném rozboru a dále

je kvantifikovat vzhledem k typu a délce textu. Takovými jevy jsou libovolné znakové celky a sémantické prvky daného jazyka. Například pokud je text vědeckého charakteru, bude mít odlišnou slovní zásobu než milostný román. Dále se budou lišit délky vět, počet například matematických výrazů, čísel a jiných odborných výrazů. Přesto tyto dva texty mohl napsat jediný autor. Pokud se však typy obou srovnávaných textů shodují, lze je přímo srovnávat například ve frekvenci používání zkratek aj. (viz souhrnný přehled těchto atributů dále v textu). Pokud se posuzuje velikost slovní zásoby nebo jiné podobné atributy, je zapotřebí započítat i faktor délky textu.

Celkem bylo izolováno kolem tisíce stylometrických atributů, avšak nepanuje obecná shoda na tom, které z těchto atributů jsou nejvhodnější pro určení autorství. Obecně se předpokládá, že i ta nejméně významná vlastnost (dle názoru autora) může v jisté situaci hrát rozhodnou roli. Poměrně zajímavé jsou také kulturní rozdíly. Například ve studii [8] jsou popisovány rozdíly autorů kratších textů dle země původu. Takové texty jsou třeba emaily nebo zprávy na sociálních sítích. Asijské kultury mají ve zvyku opisovat problém, než přistoupí přímo k jádru věci. Proto průměrná délka zprávy na novinkové desce s „"kradeným"“ software a hudbou "Euroameričana"v angličtině obsahuje 169 slov, zatímco Asijce v čínštině 807 slov. Pokud by se graficky znázornil tematický postup autora z asijské kultury při psaní konkrétního textu, pak by vypadal tento útvar jako spirála. Uprostřed této spirály by se nacházelo jádro předávané informace a křivky v okolí by byly podružné doplňující informace. Naopak ve vědecky psaném textu by takové znázornění mělo vypadat jako úsečka. Totou spořádání textu pak přímo ovlivňuje jeho strukturu a vlastnosti.

1.4 Základní kategorie stylometrických atributů

Stylometrické atributy lze rozdělit do několika základních kategorii (viz níže, [4]). Tyto kategorie nejsou disjunktní a někdy se mohou překrývat. Přesto se tyto atributy používají při profesionální klasifikaci textů a některé z těchto kategorií jsem použil také při testování algoritmů implementovaných v této práci. Některé stylometrické atributy nemusejí vycházet z aktuálního výpočtu při analýze textu, ale mohou vycházet z obecných charakteristik jazyka nebo z jiných měření. O těchto atributech tedy můžeme uvažovat jako o konstantách (například výskyt 300 nejfrekventovanějších slov v jazyce apod.).

Lexikální atributy:

- Průměrná délka slov a vět.
- Velikost slovní zásoby.
- Skupiny synonym.
- Frekvence opakování slov. V češtině a dalších jazycích vzniká komplikace kvůli skloňování a časování. V těchto jazycích se frekvence opakování slov nahrazuje frekvencí výskytu kořene slova.

- Frekvence používání nejběžnějších slov daného jazyka.
- $n-gramy$ slov, tedy například opakující se kořeny slov. $n-gram$ označuje n stejných písmen vedle sebe.
- $n-gramy$ vět (totéž jako v předchozím bodě, ale aplikováno na celou větu, tedy n slov jdoucích po sobě).
- Opakování gramatických chyb.

Znakové atributy:

- Používání čísel a číslic a jejich přesnost na desetinná místa.
- Používání speciálních znaků a symbolů ($ místo dolar nebo znaku X místo zkratky "vs."...).

Syntaktické atributy:

- "Part of speech"- kategorie pojímaná dle různých autorů různě. Někdy se označuje jako "slovní smog", tedy pro význam věty nepotřebná slova, jindy se považuje pouze za určitou předem definovanou část věty, jako například v [1].
- Částice (lingvistické, "ať, kéž, nechť...").
- Fráze a opakování frází (například: "Tak podívejte se,..."nebo "Jsem hluboce přesvědčen...").
- Opakování větných konstrukcí (např. věty stavěné na stejném způsobovém slovesu, stejná rozvitost věty apod.).
- Syntaktické větné chyby (například chybějící velké písmeno za tečkou či interpunkce kolem uvozovek, interpunkce v souvětí...).

Sémantické atributy:

- Používání synonym.
- Používání sémantických návazností (například metafory: "křišťálově čistý"nebo "ořezat rozpočet až na kost"...).

Implementace některých sémantických atributů by přesahovala rozsah této práce, a proto zde všechny sémantické atributy neimplementuji. Například u zmíněné metafory "křišťálově čistý"by bylo nutné hledat obecnou sémantickou vazbu mezi křišťálem a čistotou. To je teoreticky možné dvěma způsoby. Prvním by bylo vytvoření seznamu všech možných přirovnání. Takový seznam by ovšem byl příliš veliký. Druhá možnost by spočívala v přiřazení hlavních "metaforických"atributů ke každému slovu. Tak by třeba atributem slova "křišťál"byla mimo jiné "čistota". Vytvoření takového seznamu by bylo méně ambiciozní, přesto by byl seznam stále

obrovský. Naopak například pro synonyma již existují slovníky (thesaury), které lze využít.

Aplikačně specifické parametry se liší podle daného jazyka, a to nejen u jazyka přirozeného. Lze totiž vytvářet pravidla pro jevy jazyka uměle vytvořeného, například programovacího jazyka. Tato pravidla pak berou na zřetel povinné syntaktické jevy daného jazyka. Analýza potom upozaďuje slova typu "if, while, foreach...", a preferuje naopak pojmenování proměnných, formátování a konstrukce programu. Zcela jasně umí určit "příkazovou vybavenost"programátora, která je paralelou slovní zásoby u přirozeného jazyka.

1.5 Metody pro přiřazení autora

Metod používaných pro přiřazení jména autora k textu je mnoho a liší se leckdy pouze v detailech. Vstupem jsou tedy texty se jmény autorů (učební texty) a nepodepsaný text. Výstupem je původně nepodepsaný text s přiřazeným nejpravděpodobnějším autorem. Podepsané texty slouží jako zdroj atributů pro porovnání. Atributy se pak mohou porovnávat na základě určitého významového poměru, nebo slouží jako vstup pro strojové učení.

Základní (triviální) metodou je posuzování každého textu zvlášť a hledání přímé shody dle předchozích atributů [4]. Tento postup má však svá úskalí, zvláště pokud má autor více textů různých stylů. Ideální by v tomto případě tedy bylo, kdyby každý text obsahoval metadata o svém zaměření (stylu). Obvykle však texty taková metadata neobsahují. V případě, že jsou články různého stylu, je triviální přístup nepřesný. Pokud dochází ke srovnávání textů stejného žánru a podobné délky, dává takové srovnání přijatelné výsledky.

Prvním "přípravným"postupem je profilově založená metoda [4]. Všechny texty od jednoho autora jsou sloučeny v jediný text. Tento na první pohled triviální postup je nutný zejména v případě, že se učební texty od posuzovaného textu tematicky či stylově velice liší, nebo pokud je množina učebních textů stylisticky značně nekonzistentní. Díky sloučení všech textů se individuální rozdíly vyplývající ze stylistických rozdílů smažou nebo se alespoň sníží, což je výhodnější, než kdyby se z každého textu vytvářela vlastní statistika.

Pravděpodobnostní metoda je sumarizací všech lexikálních atributů, kdy se každému atributu, který je použit, přidává individuální váha [27]. Atributy s větším významem tak mají větší vliv na výslednou pravděpodobnost při přiřazování autorovi.

Kompresní metody používají principy využívané při kompresi [24]. Nejdříve se všechny texty jednoho autora spojí v jediný text. Algoritmus sám o sobě není příliš komplikovaný. Spojené texty se prostě zkomprimují běžnými metodami. Poté se k původním textům přidá posuzovaný text a opět se tento celek zkomprimuje. Takto se postupuje s texty všech autorů. Ten autor, který má mezi svými původními texty a svými texty po přidání posuzovaného textu nejmenší poměrný přírůstek, je pravděpodobným autorem textu. Pro kompresi se používají nejčastěji slovníkové metody RAR, LWZ a jiné slovníkové algoritmy. Dobře popsaný je

Prediction By Partial Matching (PPM) algoritmus [24], který používá RAR. Je možné používat ale i algoritmy, které dávají do slovníku nikoliv pouze sekvence znaků, ale i sekvence slov. Takový přístup je pochopitelně mnohem rychlejší. Protože tato metoda je teoreticky poměrně účinná, implementoval jsem i ji. Věnoval jsem se mimo jiné i výběru komprimačního algoritmu.

Metoda společných $n-gramů$ a variace na $n-gramy$ se zaměřuje na všechny $n-gramy$ (části slov o n znacích), a to zleva [4]. Přestože má tato metoda občas velice dobré výsledky, trpí syndromem nevyváženosti tříd při strojovém učení z textů. Pokud má jeden autor znatelně více textů k posouzení než ostatní autoři a text k posouzení je krátký, pak je autor s mnoha texty často vyloučen z množiny možných autorů posuzovaného textu. Má totiž mnohem více unikátních n-gramů, než kolik $n-gramů$ se objevuje v posuzovaném textu. Tento problém lze však obejít podle Zipfova zákona, viz dále v textu. U textů se nejčastěji posuzuje $n-tice$ písmen ve slově. Touto metodou se účinně měří například četnost předpon, přípon a koncovek ve slovech. Konkrétní provedení je popsané v dalších kapitolách.

Velice používaná metoda instancí je vlastně rozšířením triviálního přístupu [26]. Metoda vyžaduje více textů. Pokud nejsou k dispozici různé texty, metoda si vstupní text sama rozdělí (například podle kapitol v knize). Tím se liší od triviálního postupu. Všechny instance (texty) by měly být stejně dlouhé. Přesnost odhadu se podle studií zvyšuje s velikostí standardizované délky bloků textu přímo úměrně. Odhad ideální velikosti bloku textu tedy není zbytečný proces. Bloky textu o 200 slovech dávají horší výsledky než bloky textu o 1000 slovech. Na těchto instancích textu pak metoda zkouší všechny výše popsané stylometrické vlastnosti textu a ukládá si je do tvaru vektoru. Takový vektor reprezentuje celý text a poté celého autora.

Vektorové metody obecně reprezentují vlastnosti textu ve tvaru vektoru vytvořeného pro strojové učení[4], [1]. Různé algoritmy pak přistupují k této reprezentaci jinak a různě ohodnocují důležitost položek ve vektoru. Vektory například skládají do matic a z nich počítají diskriminanty. Vektory se mohou skládat na základě rozhodovacích stromů, neuronových sítí nebo genetických algoritmů aj. Jako vektor lze ale například vyjádřit počet výskytů syntaktických vzorů. Tyto algoritmy jsou však většinou aplikované na konkrétní problémy jako již výše zmiňované programovací jazyky a podobně. Vektorové pojetí umožňuje částečně omezit nevyváženost v množství zdrojů jednotlivých autorů. Také lze částečně zlepšit výsledky započítáním jednoho textu dvakrát.

Podobnostní metoda je jednoduchá metoda, která má však velice přijatelné výsledky. Nejznámější je provedení "Delta"[25]. Z textů se vybere 150 nejpoužívanějších slov, nebo alternativně 150 nejpoužívanějších slov v daném jazyce. Spočítá se průměrné používání těchto slov v textech a tento průměr je považován za základ. Pro každý text se poté vypočítá odchylka od průměru, takzvané z-skóre (tj. standardizované skóre v Gaussově rozložení s průměrem 0 a směrodatnou odchylkou 1). Pokud se v textu tato slova používají nadprůměrně, z-skóre je kladné, pokud se používají podprůměrně, pak je z-skóre záporné. Posuzova-

nému textu se poté taktéž spočítá z-skóre. Autor textu s numericky nejbližším absolutním skóre je posouzen jako autor zkoumaného textu.

K této metodě existuje mnoho rozšíření a zobecnění. Dokonce vznikly snahy matematicky podchytit poměrnou úspěšnost tohoto řešení, proto vznikly různé grafové a osové modely (Laplaceova distribuce) [21]. S dalšími pokusy se metoda zpřesňovala za pomoci zvětšení množství slov při výpočtu frekvence opakování (až 500 slov). Přesnost dále roste, pokud se z textu účelově odstraňují jména a jiná slova, která jsou pro jeden text specifická a velice častá, takže jsou pro vyhodnocení textu jako celku matoucí. V beletrii jde tedy například o vyřazení vlastních jmen, v odborné práci o turbodmychadlech je naopak vhodné z analýzy odstranit slovo "turbodmychadlo"(a skloňované tvary v případě češtiny). Nevýhodou této metody je, že vyžaduje několik lineárních průchodů texty anebo průchody databází slov. Při vhodné implementaci se však tyto průchody omezí, například pokud jsme schopni přiřadit slova do hashovacích tabulek bez konfliktů.

Tyto přístupy se mohou kombinovat a obvykle se také kombinují. Pokud se metod používá více, je možné je implementovat paralelně, čímž se rychlost výpočtu zkrátí. Některé analýzy však mohou navazovat na jiné, čímž dochází k částečné serializaci. Možnosti uvádím u implementace dále v této práci.

Další důležitou skupinou metod analýzy textu jsou různé metody pracující pouze s částí slov. Pokud totiž známe strukturu daného jazyka, dá se očekávat pozice význačných slov na určitém místě. Tak v češtině například najdeme téměř vždy předmět věty za slovesem. Pokud známe strukturu jazyka, dají se využít tyto znalosti k získání podstatných slov (sloveso, předmět, podmět . . .). Taková slova nazýváme funkční slova. Tyto se zařadí do různých struktur a nebo se dají použít při triviální analýze, viz předchozí metody. Tyto metody se považují za nejpřesnější a nejrychlejší. Problém nastává při získávání větných rozborů. Předmět věty se sice většinou nachází za slovesem, ale nemusí být vždy vyjádřený. V tom případě ho nástroj pro výběr neobjeví, nebo za něj chybně označí jiné slovo. Vyhodnocení sice probíhá rychle, protože takto vzniklý model je malý, ale jeho získání je řádově pomalejší než při prosté práci s textem.

Pokud ovšem známe gramatiku přirozeného jazyka, lze sestavit i zjednodušený obecný model stavby věty. Tento postup velice připomíná klasický formální popis gramatiky. Nevýhoda u přirozeného jazyka nicméně je, že s libovolnou změnou gramatika značně narůstá. Představme si situaci, kdy vzniká veliké ucelené souvětí s mnoha větami vedlejšími a v každé větě se pak objevuje mnoho větných prvků. To vše v jazyce rozvitějším než v libovolném programovacím jazyce. Dále musí metoda popisovat nepřesnosti a chyby v jazyce či případné výjimky (např. pokud má fráze jinou stavbu věty nebo se v závislosti na stavbě věty mění tvar slova). Proto se pracuje se zjednodušenými přepisovacími pravidly, které nepokrývají vše (buď vynechávají to, co nepoznají, a nebo pracují pouze se zkrácenými větami tvořenými funkčními slovy).

Velmi mladou metodou je hledání vzorů ve větných syntaktických rozborech. Větný rozbor se od předchozích rozborů liší v tom, že nevynechá žádné slovo

ve větě a každému slovu přiřadí svou značku. Rozbor také neprobíhá pouze na základě polohy ve větě, ale i podle typu slova (podstatné jméno, přídavné jméno, zájmeno, sloveso...) a kontextu. Každá věta má svůj jasně definovaný větný rozbor na například podmět, přísudek, předmět, příslovečná určení času, místa apod. Vytvoření takového větného rozboru trvá poměrně dlouho. Jediný způsob, jak sestavit syntaktický rozbor, je skrze strojové učení. Taková práce nezačíná u informatiků, ale u jazykovědců. Ti provedou větný rozbor na velkém množství vět, označí slova v těchto větách značkami, které popisují význam slov ve větě. Na těchto datech se poté spustí učení např. neuronové sítě. Ta by poté měla být schopna provést větný rozbor na dalších, neznámých textech. Bohužel je tento proces pro větší množství dat značně pomalý, jak ukazuji i v implementační části této práce. Druhým problémem je veliká chybovost výstupu. Když se poté hledají vzory na špatně označkovaném textu, kvalita přiřazení textu s tím úměrně klesá. Nejvyšší udávaná přesnost bývá 95 % správného označkování. V běžném použití však správnost určení klesá až k 70 % v závislosti na komplexnosti textu a implementaci systému pro zpracování přirozeného jazyka (natural language processing - NLP). Zástupcem této metody je metoda diskriminačního syntaktického stromu [1].

1.6 Komprimační metoda

1.6.1 Obecný popis

Komprimační metoda je principiálně velice jednoduchá. Popsal jsem ji z části již v obecném přehledu metod. Zde ji rozepíši více do detailu a zároveň zde udělám již menší srovnání s dalšími metodami.

Algoritmus si v prvním kroku načte všechny texty všech známých autorů a poté využije komprimační algoritmus k tomu, aby takovou množinu všech textů autora zkomprimoval do jednoho celku. Komprimuje se vždy celý soubor textů autora, protože tento postup je citlivý na malé nebo extrémně malé vstupy (řekněme, že vstupem autora by bylo 30 slov). Tím má nevýhodu oproti některým metodám, kterým malé vstupy nevadí. Poté se zjistí paměťová velikost takového výstupu. Poté se přidá k textům autora nepřiřazený text a znovu stejným algoritmem se tyto texty zkomprimují. Výsledná velikost se odečte od velikosti výstupu bez přiřazeného článku. Vzniklý rozdíl je pak tím menší, čím větší je pravděpodobnost autorství, neboť algoritmus využívá pravidelností vstupu k tomu, aby výsledek zmenšil co nejvíce.

1.6.2 Výběr algoritmu

V článcích popisujících tuto metodu se používaly především slovníkové algoritmy, např. [24]. Zajímalo mě, který algoritmus by byl nejefektivnější jak v úspěšnosti přiřazení, tak z časového hlediska. Problém většiny studií na téma přiřazení autorství totiž je, že se v maximální míře soustředí pouze na přesné určení, nikoliv na časové nároky. Já jsem přihlížel i k rychlosti. Hledal jsem proto i u to-

hoto postupu algoritmus, který by byl rychlý a zároveň přesný. Abych takových algoritmů prošel co nejvíce, použil jsem knihovnu algoritmů, pro podrobnější popis viz implementace. Srovnával jsem ZIP komprese a RAR komprese s algoritmy LZW (Lempel–Ziv–Welch) aj. Jako výsledný algoritmus jsem použil LZMA (Lempel–Ziv–Markov chain algorithm) [10], který používá mimo jiné opensource formát ".7z". Nabízel se ještě algoritmus LZMA2, ale ani po několika testech jsem nezměřil zásadní rozdíl mezi LZMA a LZMA2 (průběžné benchmarky jsem prováděl na několika spojených textech z Bible, každých o velikosti cca 4.4 MB čistého textu 1 byte na znak).

LZMA algoritmus je slovníkový, kompresní algoritmus podobný LZ77, ale s lepším poměrem komprese než bzip2[10]. Velikost slovníku může dosahovat až 4 GB. Rozdíl oproti původnímu algoritmu spočívá v proměnné velikosti bloků, kdy algoritmus nepostupuje po jednom bytu, ale může si nastavovat různě velké bitové pole. Dalším vylepšením je proměnná délka mezer mezi opakujícími se shodami. Tím, že dovoluje relativně veliký slovník, není potřeba jej často (nebo vůbec někdy) zresetovat a tvořit znovu. Zkomprimovaný blok se jmenuje paket a má popisující hlavičku, která určuje jeho další výskyt na základě předchozích výskytů. Konkrétní struktura se nachází v [10]. Vzhledem k tomu, že pro určení autorství není potřeba dekomprese, nemá význam ji na tomto místě rozebírat. Za zmínku však stojí úroveň komprese, která se může měnit. Zde jsem ale dospěl ke zjištění, že nastavení komprese nemá na výsledek vliv. Na větším datasetu by se výsledek nicméně mohl projevit.

1.6.3 Výhody metody

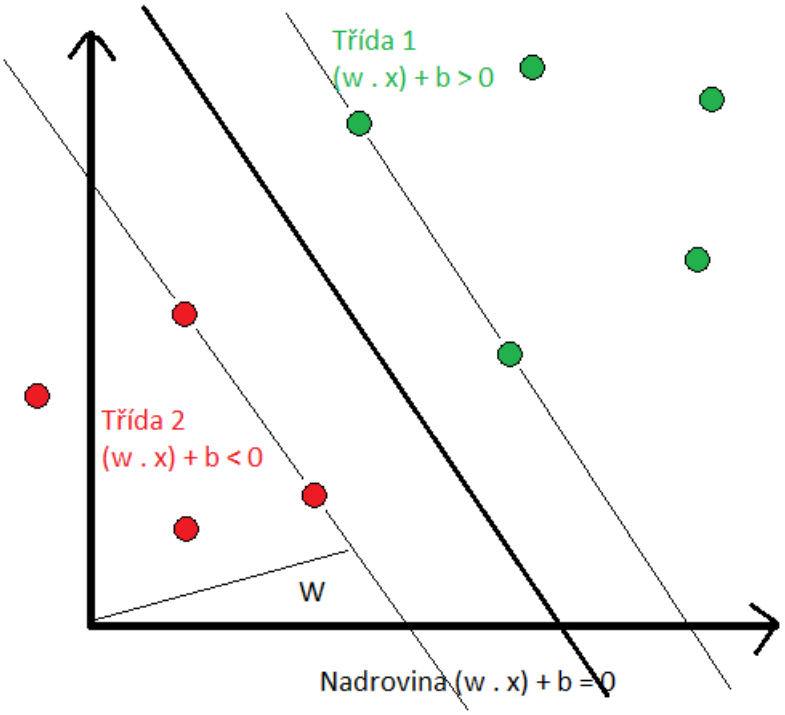
Výhoda této metody pro případné praktické užití spočívá v tom, že je extrémně rychlá, snadno pochopitelná a v případě pokročilých kompresních algoritmů je i poměrně účinná. Do výzkumu komprimačních metod se vkládá značné úsilí, a tím roste i jejich užití pro tuto tematiku. Cíl je totiž společný. Čím větší podobnost, tím větší komprese. Kompresní algoritmus totiž pokrývá veškeré možné lexikální vlastnosti, a to aniž by se jim vlastně explicitně věnoval. Podobnost se hledá na úrovni bitů a bloku bitů, čímž se v podstatě věnuje nejen celým slovům, ale zároveň i kořenům slov, $n - gramům$ apod. Zároveň ale mimoděk provádí jakousi syntaktickou kompresi. Struktura věty se pochopitelně také určitým způsobem opakuje. Obsah může být jiný, ale opakuje-li se nějaká fráze často, kompresní metoda na rozdíl od lexikálních metod tuto frázi postihne celou v bloku, nikoliv jen slovo od slova.

1.7 SVM - Support Vector Machine

1.7.1 Obecný popis

SVM je metoda strojového učení [3]. Jde o často využívanou metodu z několika důvodu. Je schopná vyhodnocovat naprosto jakékoliv konkrétně dané vlastnosti textů (stylometrické i jiné). To znamená, že metoda nemusí používat pouze vy-

braná slova nebo často frekventovaná slova, ale lze použít celý vstup se všemi slovy. SVM dále poměrně dobře zvládá situaci, kdy jeden autor píše texty různého žánru, protože zohledňuje i kontext. Záleží pouze na tom, co SVM získá jako vstup ve tvaru vektoru. SVM je tak rychlá, že pracuje v reálném čase, a lze tak detekovat například spam v elektronické poště [3].



Obrázek 1: Rozdělení pozitivních a negativních výsledků nadrovinou.

V jednoduché lineární formě nedělá SVM nic jiného, než že rozděluje přímkou (nebo plochou v případě 3D modelu) pozitivní a negativní výsledky s co největším rozdílem (okrajem) [5], tak jak je vidět například na obrázku 1. Forma výsledku je ve tvaru $u = w \cdot x + b$, tedy vektor, kde w je normálový vektor k rozdělující nadrovině (čáře, rovině), x je vstupní vektor. Rozdíl je pak definován jako největší možná plocha od dělící nadroviny k nejbližšímu negativnímu a pozitivnímu výsledku. Rozdělení je pochopitelně binární, vznikají tím tedy dvě třídy (pozitivní, negativní; ačkoliv konkrétně detekce autorství může v některých případech vyžadovat více tříd, tento problém dále rozeberu později v kapitole o diskriminačním syntaktickém stromu). Dále existují problémy, jejíž instance nejdou rozdělit přímkou, ale pouze křivkou. Výpočet největšího možného dělení je NP problém, na který lze aplikovat optimalizační algoritmus (maximalizační):

$$maximalizuj \frac{1}{2}||w||^2 \, vzhledem \, k \, y_i(w \cdot x_i + b) \geq 1, \forall i,$$

kde x_i je i-tá trénovací množina a $y_i \in \{-1, 1\}$ je výstup pro i-tý vstup. Samotná nadrovina je tedy určena pouze instancemi problému, tím je daný okraj nadroviny, podpůrné vektory.

SVM je koncipováno podle principu "structural risk minimization" [5] z teorie strojového učení. Hledá se tedy model, podle kterého bude garantovaná nejnižší možná chyba výsledku pro neznámou nebo nepoužitou množinu příkladů. Důsledkem je, že se SVM nesmí přeučit na konkrétní data, ale musí správně posuzovat i ta neznámá, avšak odpovídající data.

SVM lze rozšířit na nelineární model namapováním na vícerozměrně dimenzionální vlastnost určenou předem. V tomto prostoru se pak sestaví rozdělující nadrovina. Nechť $\phi > \Re^N \rightarrow F$ je zobrazení $x \rightarrow y = \phi(x)$ takové $N \ll dim(F)$. Pak pro dané ϕ je lineární klasifikace $u = s \cdot z + b$ s učebním parametrem s. Algoritmus pracuje pouze se skalárním součinem. Tudíž se vícedimenzionální hodnoty saz se nemusí počítat, pouze skalární součin $k(x, w) := \phi(x) \cdot \phi(w) = s \cdot z$ je vyžadován pro výpočet. V tabulce 1 jsou příklady funkcí pro jádro.

Lineární	$k(x, y) = x^T y + c$
Polynomiální	$k(x, y) = (xy' + c)^d$
Sigmoid	$k(x, y) = \tanh(\kappa(xy') + \theta)$
RBF	$k(x, y) = exp(-\gamma\|\|x - y\|\|^2)$
Gaussovo	$k(x, y) = exp\left(-\frac{\|\|x-z\|\|^2}{2\theta^2}\right)$

Tabulka 1: V tabulce jsou příklady jader pro SVM. Lineární jádro je nejjednodušší funkce. Polynomiální jádro je vhodné pro datasety s normalizovanými daty. Gaussovo jádro je vhodné naopak pro data, o nichž nevíme téměř nic před samotným posuzováním. RBF lze použít při analýze signálů jako například obrázků (detekce obličejů, hledání tvarů). Při posuzování autorství se sice používá RBF, ale autoři diskriminačního syntaktického stromu použili lineární jádro [1].

Vektor x je zařazen do třídy 1, pokud rozhodovací funkce nabude větší hodnoty než 0.
$h(x) = signum\left(\sum_{i=1}^{n} \alpha_i \gamma_i k(x_i, x) + b\right)$
Pouze instance α_i korespondující s podpůrnými vektory x_i jsou rozdílné od 0 (instance na hraně). Pokud přepíšeme $h(x) = signum(w \cdot x)$ a vybereme nějaké $\gamma > 1 - \delta$ nad množinou l trénovací množiny, pak
$Error \leq v + \sqrt{\frac{c}{l}\left(\frac{R^2\Lambda^2}{\gamma}\log^2 l + \log\frac{1}{\delta}\right)}$
, kde $||x|| \leq R, ||w|| \leq \Lambda$. ν je část trénovací množiny, kde rozdíl je $y_i(wx_i) < \gamma$ a c je velká konstanta. Vstupy x mají stejně velkou délku.

1.8 SVM s kořeny slov

Vzhledem k tomu, že při implementaci budu používat SVM tak, jak ho implementoval T. Joachims [3], budu se zde věnovat vlastnostem jeho provedení (tzv. SVM^{LIGHT}). Obecně však platí většina vlastností pro jinak zavedené SVM nebo pro ty z něj vycházející. Pro všechna SVM platí, že nevstupuje jako učící množina dat přímo text. I v angličtině se používají k analýze kořeny slov. Kořeny

slov odstraňují problémy způsobené skloňováním a časováním sloves. Mimochodem autoři textu [2], kteří rovněž ve své srovnávací práci používali SVM, jsou původem německé národnosti a své testy prováděli na německy psaných textech. I proto bylo dle mého názoru jejich zájmem zapojit kořeny slov, neboť němčina je tvarově bohatší než angličtina. Dále toto provedení SVM^{LIGHT} vzhledem k efektivitě může potlačit existenci raritních kořenů. Tyto metody odvozené od SVM mají potvrzený výkonový benefit oproti naivním algoritmům Bayesovým (avšak sekvenčním) a v podstatě stejný kvalitativní výsledek [2] (u úloh tohoto typu lze stěží říci, že půjde o každý jeden příklad). SVM dává také prokazatelně lepší výsledky oproti Rocchio algoritmu [3] (pro popis Rocchio algoritmu viz [6]) a algoritmům postaveným na základě rozhodovacích stromů [2].

SVM využívá váhy na ohodnocení důležitosti typu slov v textu. Původně se váhy používaly pro omezení velikosti vstupu (vstup byl ořezán o slova váhy menší než δ). Protože je však SVM výkonná metoda, není potřeba její vstup zmenšovat. Slovům se dává tím menší váha, čím častěji se vyskytují mezi texty dokumentů. U takového slova se totiž očekává přílišná obecnost, která neodlišuje daný text od jiných (a tím autora od jiného autora). Používáme termín inverzní frekvence výskytu v dokumentu (idf) $f_{idf}(w_k) = \log \frac{n_d}{n_d(w_k)}$. $n_d(w_k)$ je počet dokumentů v kolekci obsahující slovo w_k a n_k je počet všech dokumentů v kolekci. Inverzní frekvence tedy roste s tím, jak klesá počet výskytů slova v dokumentech a maxima nabývá, pokud se objeví pouze v jednom dokumentu.

Dalším zajímavým fenoménem, který má vliv na analýzu textu a který je nutno zahrnout do výpočtu, je takzvaný Zipfův zákon [7]. Americký lingvista George Kingsley Zipf si povšiml zvláštnosti ve frekvenci výskytu slov. Pokud si vyrobíme tabulku, kde sestupně seřadíme slova dle počtu výskytů, všimneme si určité odvoditelnosti v úbytku počtu výskytu slov. Druhé nejfrekventovanější slovo je dvakrát méně časté, než první slovo. Třetí slovo je pak třikrát méně časté, než nejfrekventovanější slovo atd. Originální formulace Zipfova zákona tak vypadá takto

$$f(r) = \frac{A}{B+r},$$

$f(r)$je frekvence výskytu $r - t\acute{e}ho$ prvku tabulky a A a B jsou parametry textu. Nicméně takto definovaný vzorec neobstojí velice často. Je to mimo jiné způsobeno takzvaným "hapax legomenem". To je zvláštní případ slova nebo slovního spojení, které se objeví pouze jednou v celém textovém souboru. Textovým souborem se nemyslí soubor na disku, ale lingvistický textový soubor, tedy svazek knih jednoho autora či žánru nebo třeba celý jazyk. Velký problém je to u mrtvých jazyků jako např. u latiny nebo starořečtiny. V našem případě je to problém pro Zipfův zákon. Pokud se totiž takových tvarů objeví více, není vzorec plynulý a slova se zanedbat nedají, protože nesou unikátní typ informace. Zipf a jeho kolegové proto upravili Zipfův zákon na formuli

$$f(r) = \left(\frac{A}{B+r}\right)^{\frac{1}{\gamma-1}},$$

tento vzorec (Zipf–Mandelbrotův zákon) pak už platí i při výskytech méně obvyklých slov [20].

Aby se takováto frekvence dala porovnávat mezi texty různé délky, je vhodné normalizovat vstup. Normalizací se nemyslí nic jiného, než přepočítání frekvencí, jaká by byla při stejné délce textu. Pro SVM je tato úprava doporučovaná a provádí se obecně formulí

$$d_i^* = \frac{d_i}{||d_i||_{L_2}},$$

kde $||d||_{L_2} = \left(\sum_{i=1} |d_i|^p\right)^{\frac{1}{p}}$ je p-norma a $||x||_{L_2}$ značí eukleidovskou normu. Tím na kratším textu vznikne efekt přidání slov. Přijímaný vzorec pro příbytek nových slov $|V|$ s rostoucí délkou textu $|N|$ je

$$V = N^C, \;\; 0 < c < 1.$$

Tudíž nárůst frekvence při zvětšení textu z N_1 na N_2 není lineární, ale faktor zvětšení vypadá jako $a\frac{N_1}{N_2}$, $0 < a < 1$.

Nechť $f(r)$ je frekvence distribuce v textu N_1, délky $|N_1|$ a slovní zásobou velikosti V_1. Text N_2 obsahuje N_1, totéž platí pro množiny slovní zásoby. Potom $af(r)$ je frekvence distribuce v textu N_2. Platí

$$|N_1| = \sum_{r=1}^{|V_1|} f(r),$$

a pro druhý text

$$|N_2| = a\sum_{r=1}^{|V_2|} f(r) = \sum_{r=1}^{|V_1|} f(r) + a\sum_{r=|V_1|+1}^{|V_2|} f(r) = a|N_1| + a\sum_{r=|V_1|+1}^{|V_2|}.$$

$$\text{Tudíž platí } ||N_2||_{L_p} = \sum_{r=1}^{|V_1|} f^p(r) + \sum_{r=|V_1|+1}^{|V_2|} f^p(r).$$

Po dosazení do Zipfovy formule můžeme vyjádřit parametr a

$$a = \frac{||N_2||_{L_p}}{||N_1||_{L_p} + \sum_{r=|V_1|+1}^{|V_2|} \left(\frac{A}{B+r}\right)^{\frac{p}{\gamma}}}$$

Pokud $p > \gamma$, pak suma konverguje, jak roste slovní zásoba k nekonečnu. Pokud nerovnost platí naopak, pak diverguje. Empirické testy dávají lepší výsledky v L_2 normě než v L_2 normě při rostoucí γ.

SVM je metoda, která je velice rychlá a jejím vstupem může být libovolný parametr, který se dá převést do tvaru vektoru. Takže jsem se rozhodl použít ho nejdříve s nějakou přímočařejší vlastností, jako je počet výskytů daného kořene slova v textu. Tím se odmazává problém skloňování. SVM se také používá při metodě diskriminačního syntaktického stromu.

1.9 Naivní metoda

1.9.1 Obecný popis

Po prostudování základních vlastností textů jsem se rozhodl spojit více lexikálních vlastností do jednoho algoritmu. Základním problémem je délka výpočtu. Lexikální vlastnosti sice netrpí na požadovaný čas výpočtu ani zdaleka tolik, co vlastnosti syntaktické (viz pozdější kapitola o diskriminačním syntaktickém stromu, zejména implementace a měření), přesto mají složitost alespoň (kn), kde k je počet lexikálních vlastností, které posuzujeme. Pokud bychom se ale rozhodli dát každé vlastnosti jedno vlákno výpočtu, které by bylo nezávislé na ostatních (byl by to modelový případ mnoha čtenářů bez písaře), pak by zpracování mělo dosáhnout časové složitosti $\Omega(n)$.

Nyní si tedy stačí vybrat některé vlastnosti, které budou posuzovány. Vybral jsem ty základní, které byly popsány ve výčtu. Jsou to

- Průměrná délka věty.
- Průměrná délka slova.
- Počet unikátních slov v textu (velikost slovní zásoby).
- Všechny bigramy slov.
- Všechny trigramy slov.
- "Abnormální"slova (nespisovná slova, neidentifikovatelná slova).
- Skupiny synonym.

Při konstrukci a testování jsem ale narazil na několik problémů. Současné slovníky mají problémy pokrýt i některé základní lexikální vlastnosti. Například slovník pro detekci nespisovných (abnormálních) slov často mylně zařadí méně běžné slovo jako nespisovné. Problém nastává také s archaismy. Přestože jsem původně v této práci počítal i s podporou českého jazyka, musel jsem ji kvůli těmto problémům opustit. Je těžké najít platformu, která by podporovala ve stejné kvalitě jak angličtinu, tak češtinu. Chyby slovníkových zdrojů zhoršují výsledky srovnávání. Stejný problém pak nastává u NLP pro větný rozbor užitý v poslední metodě. Abnormálním slovům jsem tedy přiřadil menší význam než některým spolehlivějším vlastnostem.

Další veliký problém způsobuje thesaurus. Thesaurus je slovník synonym a významově podobných slov. Problém s touto funkcí je, že je komplikovanější uplatnit její užití. Autor by měl splňovat to, že zná-li pro nějaký jev dvě slova, která jej popisují, měl by používat stále tyto dva pojmy a maximálně by jich mohlo přibývat v čase. Pokud tedy autor A zná pro jev tři slova a autor B zná pro jev dvě slova, přičemž v porovnávaném textu se používají pro tento jev dvě slova, autorem textu by měl být autor B. První problém spočívá v tom, že někdy se skupiny synonym prolínají, ačkoliv neznamenají totéž (třeba proto,

že dané slovo má dva různé významy a jeho konkrétní význam záleží na kontextu). Thesaurus ale vyhodnotí dvě slova jako synonyma, protože jimi mohou být, a tak vznikne jakási "skupina synonym"s chybnými prvky. Tento problém nastává obzvláště u na slova chudé angličtiny. Tím dochází k deformaci velikosti slovní zásoby a vznikají chyby.

Další problém spočívá v nutnosti projít celý text znovu pro dohledání existence synonyma. Toto lze naštěstí vyřešit dobrou konstrukcí paměťové struktury (více v implementační části). Nicméně ani dobrá implementace se nemůže neprojevit na pomalejším výpočtu této vlastnosti, a proto zde očekávám úzké hrdlo výpočtu.

Další položky jsou počet unikátních slov, počet unikátních kořenů a počet bigramů a trigramů slov, a to vždy vzhledem k délce článků (aby nehrála takovou roli délka zdroje). Délku slova pochopitelně měřím na znaky, zatímco věty na slova. Počítat počet znaků ve větě by nemělo vypovídací hodnotu téměř žádnou.

Podstatná jména a přídavná jména mají v některých jazycích důležitou vlastnost, a totiž že slova se skloňují. Např. slova "krásný", "krásná"a "krásné"jsou tudíž rozeznávána jako tři nezávislá unikátní slova. Podobným problémem je změna při změně slovního druhu. V angličtině je to například "beautiful"a "beautifully". Zřejmě každý, kdo zná slovo "beauty", z něj dokáže odvodit a napsat slova "beautiful"či "beautifully". Proto tato dvě slova nejsou slova unikátní a měla by se počítat jen jednou, protože jejich základ, tedy kořen, je stejný. Jako slova se počítají rozdílně, v kořenech se však jen zvýší frekvence daného kořene.

U trigramů a bigramů je koncept částečně podobný. Angličtina zná a má tabulky pro nejfrekventovanější bigramy i trigramy slov [9]. Jejich význam v textu je ale menší než význam kořenů slov. Bigramy, ale i trigramy jsou většinou příliš malé části slov na to, aby mohly nést význam. Nicméně slova se skládají ze slabik, které už n-gramy pokrývají. A protože slabik není neomezené množství a v každém jazyce jich je řádově kolem tisíce, autoři se mohou odlišovat nejen na základě používaných slov, ale i používaných předpon, spojek, přípon a koncovek.

1.9.2 výpočet

Problém nastal v okamžiku, kdy jsem se snažil získat váhy pro jednotlivé prvky. Změna ohodnocení pro délku textu pro mě byl nejdříve údaj velice důležitý. Laicky jsem očekával, že každý autor většinou píše zhruba stejně dlouhé texty, což se záhy ukázala být chyba. Problémem byla například frekvence výskytu jednotlivých bigramů, protože bigramů se v textu vyskytuje enormní množství. Bigramy a další velké množiny jsou tak zastoupeny pouze kvantitativně (tzn. zaznamenává se pouze počet unikátních bigramů v textu), nikoliv každý bigram se svou frekvencí zvlášť.

Během snahy přiřazovat různé váhy se ukázalo, že pokud jsem vylepšil váhu v jedné vlastnosti, byl sice autor A přiřazen násobně lépe než před změnou, zato autor B tak přišel o mnoho správných přiřazení. Pokud jsem přesto našel kompromis a částečně eliminoval výjimky mezi oběma autory, zcela selhalo přiřazení

autora C.

Tento jev je způsoben dvěma různými důvody:

- Lexikální vlastnosti se obecně těžce balancují vahami, zvláště při nekonzistentnosti psaní autorů.
- Empiricky zjištěné váhy se stávají neaktualizovatelné, pokud se často mění velká část vstupů.

Druhý bod je aplikovatelný spíše na atribuci textů než obecně. Například v grafice při převodech palet jsou empiricky zjištěné hodnoty univerzální a fungují na každý obrázek. Při vytváření vah pro atribuci textu stačilo mírně posunout základní údaj, jako například toleranci odchylky délky textu, a dva autoři náhle začali splývat, protože původní váhy tuto hodnotu silně reflektovaly.

Rozhodl jsem se tedy aplikovat jednotlivé základní vlastnosti a výskyt unikátních slov jako vstup pro SVM. Slova místo kořenů jsem vybral záměrně. Očekávám u nich sice menší úspěšnost než u unikátních kořenů, ty jsem však již jednak implementoval v jiné metodě a navíc mým cílem v této metodě bylo ilustrovat efektivitu triviálního přístupu.

1.10 Metoda diskriminačního syntaktického stromu

1.10.1 Úvod

Všechny předchozí metody si všímaly lexikálních vlastností textů, ze kterých se usuzoval styl psaní autora. Podle těchto kvantifikovaných vlastností různého druhu se pak hledal určitý koeficient podobnosti mezi dvěma texty. Tento přístup je alespoň při triviálním postupu poměrně časově náročný, protože různých stylistických faktorů lze najít řádově tisíce. Tedy za předpokladu, že se budou analyzovat všechny, nikoliv jen vybrané.

Proto autoři Sangkyum Kim, Hyungsul Kim a spol. z University of Illionois přišli s metodou konstrukce nové množiny vlastností, a to do podoby upraveného k-stromu [1]. Zároveň pak ukazují nový postup při vyhledávání dat (data mining) z textu do k-stromu. Dle jejich tvrzení je takovýto způsob ukládání dat efektivnější, než dosud používané množiny syntaktických vlastností. K potvrzení tohoto tvrzení pak přidávají i některé množiny vlastností z jejich vlastních testů. V této práci (v přiloženém software) budu tyto výsledky porovnávat s jinými daty. Nicméně v této části textu se budu věnovat popisu této nové metody. Text zde vychází přímo z originálního článku [1], protože je to jediný zdroj k této metodě.

Tato metoda využívá sestaveného syntaktického stromu, ze kterého se dá určovat typ slova (dle jeho pozice ve větě). To umožňuje zrychlení vyhodnocování takzvaných funkčních slov (slova jako větný předmět, předložky a zájmena). Pomocí těchto slov se dá určovat autorství s podobnou účinností jako při použití jiných metod zahrnující všechna slova. Dále se objevily nové techniky pro přirozené zpracování textu (Natural Language Processing - NLP) a spojování členů

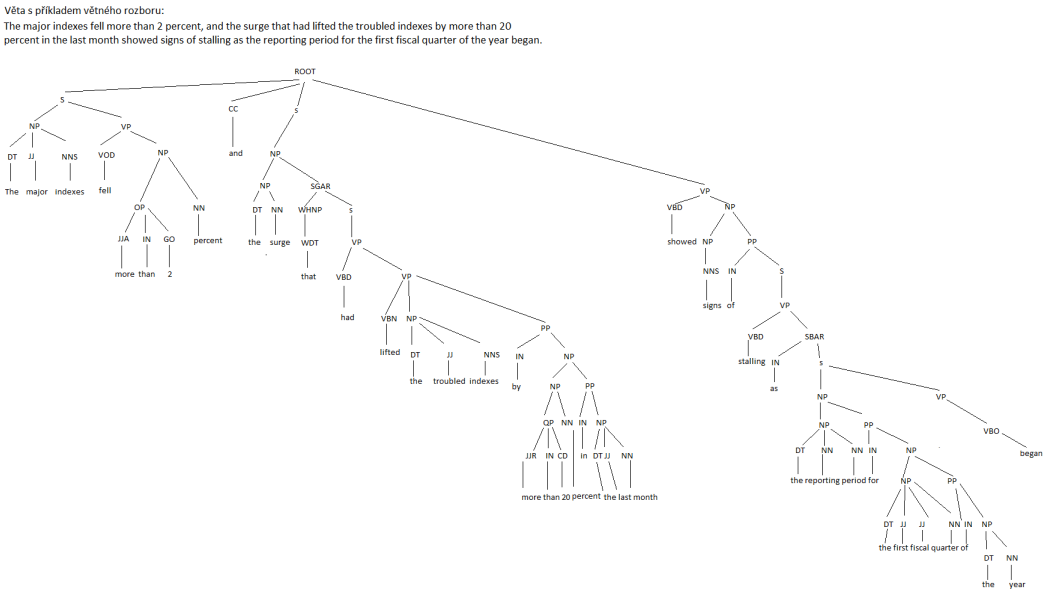
ve větách na jednotlivé lingvistické prvky (na part of speech prvky - zde v pojetí efektivní slova). Ty tvoří syntaktický strom.

1.10.2 Odbočka k NLP a porovnávaným metodám

Nejdříve je potřeba podívat se na tematiku jazykového automatizovaného rozboru a její historii. Konstrukce takového počítače, který by komunikoval s uživatelem pomocí vstupu v přirozeném jazyce, nikoliv v umělém, byl snem každého spisovatele sci-fi, ale také otců prvních počítačů. Koneckonců již Alan Turing v roce 1950 přišel s článkem "Computing Machinery and Intelligence"[18], který se zabýval umělou inteligencí a ve kterém pojem umělé inteligence taktéž definoval. Zde také došlo k prvnímu jasnému poznání problému, že převést obsah věty, nikoliv formu, do algoritmu výpočtu počítače není triviální úloha. Proto člověk rozliší komunikaci s počítačem od komunikace s inteligentní bytostí. Tak vznikl Turingův test, kde se autoři pokouší imitovat vyjadřování člověka počítačem. Protože komunikace probíhá vstupem do konzole a slova nenesou obsahovou značku, ale jsou jen formálně uspořádána dle pravopisných pravidel, objevil se poprvé problém větné analýzy přirozeného jazyka.

V průběhu 60. a 70. let probíhaly spíše teoretické a jednoduché návrhy, jak tento problém řešit. Hlavním problémem byla a dodnes je výpočetní náročnost. Proto až ke konci 80. let vzniklo něco jiného než "napřímo"napsaná pravidla pro program stylu "otázka-odpověď". Teprve v pozdních 80. letech se dostatečně zvýšil výkon počítačů, takže byl dostatečný pro strojové učení. To spolu s formálně rozepsanými pravidly gramatiky dle Chomského umožnilo první pokusy s učením na textech a nebo na tehdy populárním hlasovém vstupu [19].

Velkým pomocníkem pro potřeby strojového učení je překvapivě obrovské množství textů vznikajících díky byrokracii Evropské Unie. Spolu s problémy rozpoznávání hlasu a vytváření větných rozborů textů je zde také problém automatického překladu z jednoho jazyka do druhého. Jak takové použití vypadá, krásně ilustruje web translate.google.com, který taktéž funguje na principu strojového učení, a proto tak špatně překládá například texty z mrtvých jazyků, jako například z latiny, protože se zde již nemá z čeho učit. Ale v Evropské Unii se mluví mnoha jazyky a každý z nich je úředním jazykem EU. Překladatelské práce jsou značně nákladné, a tak motivací EU je, aby vznikl počítačový neomylný překladač, který bude „"zdarma"“. Nejen proto zveřejňuje všechny texty ve všech jazycích. Velké množství identických textů v mnoha jazycích je jako zdroj pro strojové učení překladů nenahraditelné. Protože překlad z jednoho jazyka do druhého probíhá mimo jiné na základě syntaktického (větného) rozboru, je strojové učení překladů také zdrojem pro rozvoj automatizovaných metod větného rozboru, které jsou zásadní pro syntaktické metody určování autorství. Možnosti použití větného rozboru k určení autorství textu však zůstávala dlouho podceňována. První texty dostupné v odborných databázích věnujících se tomuto tématu pocházejí teprve ze začátku našeho tisíciletí.



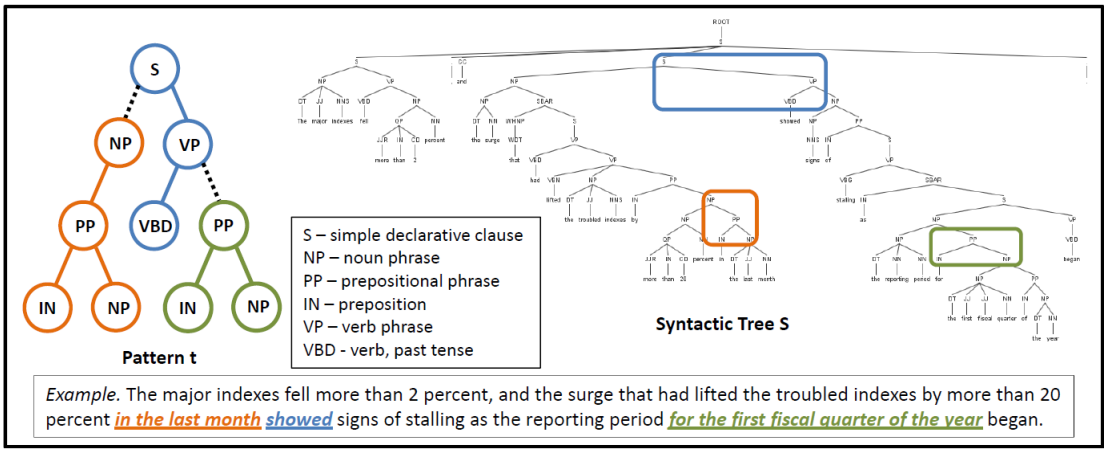
Obrázek 2: Syntaktický strom pro větu, která je uvedena jako příklad v článku popisu metody [1].

1.10.3 Popis struktury a příklad

Syntaktický strom je datová struktura typu k-strom s jedním kořenem. Je vyvažovaný podle popisků part of speech (POS) tagů. Tyto POS tagy nemusí nutně být celá slova, ale libovolné značky nebo řetězce popisující syntaktickou stavbu věty. Takto popsaná věta vložená do stromu se nazývá k-vložená hrana podstromu (k-embedded edge neboli k-ee). Mezi dvěma uzly je vložená hrana, pouze pokud je mezi těmito uzly vztah předek-následník, nikoliv rodič-dítě. Jednoduše řečeno k-ee strom tvoří hrany mezi podstromy syntaktického stromu.

Příklad je na obrázku 3 na straně 27. Syntaktický strom je uveden na větě, která není krátká a není jednoduchá ani svojí větnou stavbou. Ze syntaktického stromu je vidět, že rozeznává ze syntaxe věty funkční vazby slov. Pozná tedy z pozice slova ve větě, co je předmět, co je podmět a co je přísudek. Tato vlastnost se ale liší jazyk od jazyka. To je veliká komplikace oproti jiným metodám (např. metodě slovníkové komprese).

Zatímco ostatní verze syntaktických stromů z jiných prací pouze spojovaly různé nezávislé podstromy (části vět, věty), tento syntaktický strom zachovává původní strukturu velkého souvětí. Na obrázku 3 je vidět takový syntaktický strom s $k == 2$. Věta pochází z *The New York Times*. Na obrázku je zvýrazněn vzor t. Tento vzor se objevuje v článcích autora v 21,2 % vět. Naproti tomu u jeho kolegy se tento vzor stavby věty objevuje pouze v 7,2 % vět. Z tohoto rozdílu je zřejmé, že určení autorství v tomto případě (mezi dvěma potenciálními autory) by nebyl problém. Vzor t je tvořen třemi podstromy spojenými dvěma k-vloženými hranami. Konkrétně tyto hrany jsou vyznačeny čárkovaně (S,NP) a (VP,PP). Autoři sice tvrdí, že je k-ee strom lepší než jiné formy držení



Obrázek 3: Příklad syntaktického stromu převzatý z původního článku. Jedná se o verzi s úpravami od 2. Na syntaktickém stromu je zvýrazněna budoucí struktura k-ee stromu, která je vlevo. Příklad tedy ukazuje 2-ee strom.

syntaktického stromu, na druhou stranu nejlepších výsledků dosáhli s 1-ee stromem a 0-ee strom na tom byl pouze o málo hůře. 0-ee strom funguje, jako kdyby ve větě žádná struktura nebyla. 2-ee strom pak byl z hlediska výpočetního času exponenciálně náročnější.

1.10.4 Teoretické srovnání a problémy

Tak jako vždy při data miningu vzniká problém možného přehlcení informacemi. V tomto případě hrozí získání příliš mnoha vzorů. Proto je potřeba tyto syntaktické stromy zbavit přebytečných, málo často se vyskytujících vzorů. Takovéto málo se vyskytující vzory by mohly způsobit vyřazení i jinak vhodného přiřazení autor - text. Pokud samotný jeden autor používá určitou větnou konstrukci poměrně málo, neznamená to, že když ji daný text obsahuje taktéž málo, že je jeho autorem. Existuje totiž více forem, které nepoužíváme, než těch, které používáme. Další důvod je rychlost výpočtu. Při posuzování podle slov nese vždy největší informaci slovo málo časté. Tvrdit tedy, že málo časté vzory nejsou diskriminační, je dle mého názoru sporné. Zároveň ale nelze popřít, že vzhledem k množství vzorů v textu není možné všechny vzory zachytit a použít. Algoritmus ale stojí na výběru nejčastějších vzorů v textu.

Získávání vzorů obvykle probíhá ve dvou fázích. První fází je vygenerování vzorů, v druhé fázi se jednotlivé vzory otestují na četnost výskytu. Poté se málo frekventované vzory vyřadí. Nicméně jsou i práce zabývající se možností vygenerování vzorů přímo s větším množstvím výskytu. Takový proces je pochopitelně rychlejší. Autoři kriticky přiznávají, že tyto metody zatím nedávají výsledek vhodný pro stavbu k-stromu. Jednofázové algoritmy totiž dávají pouze výsledek výskytu ano-ne. Zde by bylo potřeba určit rovnou i počet výskytů. Algoritmus tedy ve skutečnosti prochází text dvakrát, jednou pro zisk vzorů a podruhé pro počet výskytů.

Při použití této metody pak byly výsledky autorů o 8,23 % v průměru lepší než u jiných metod a podle t-testu byl model autorů lepší na 95 % hladině spolehlivosti. To v podstatě znamená, že v 95 % případů vyhodnotí autorství lépe tento postup než jiná porovnávaná metoda.

1.10.5 Vlastnosti vhodného syntaktického stromu

Zde popsaná metoda konstrukce syntaktického stromu stojí na principu POS slov, jak již bylo zmíněno výše v úvodu. V tomto syntaktickém stromu je jeden kořen a všechny uzly jsou uspořádány tak, aby dávaly celkový obsah věty. Jednotlivé vnitřní uzly pak mají POS značku podle přepisovacího pravidla na levé straně, ze které se derivovala strana pravá. Znamená to tedy, že proces tvorby vychází "zespoda-nahoru", podobně jako při syntaktické analýze během překladu zdrojového kódu ve vyšších programovacích jazycích. Vnitřní uzly tedy obsahují neterminální znaky z gramatiky a listové uzly jsou tvořeny terminály (přímo lexémy z věty).

Přepisovací pravidla jsou definována běžným způsobem, kde přepisovací pravidlo má stavbu $X \rightarrow Y$. Neterminál X lze kdekoliv v textu derivovat (odvodit) na symboly X (terminály i neterminály). V případě přirozeného jazyka tak budou pravidla vypadat například takto: $NP \rightarrow DT + JJ + JJ + NN$. Tedy noun phrase, neboli jmenná fráze, tedy vše co se pojí s podstatným jménem, jeho vlastnosti (JJ - adjektiva, přídavná jména), determinant (DT - determinant, v podstatě zájmena, číslovky nebo člen (the, a, der, die, das...) a podstatné jméno (NN). Příkladem takového vzoru: $NP \rightarrow DT + JJ + JJ + NN$ by byla např. tato část věty: "takový (DT) krásný (JJ) červený (JJ) míč (NN)"nebo "the (DT) big (JJ) white (JJ) house (NN).

Nepříjemností u metody přepisovacích pravidel je, že je potřeba definovat nejen pravidlo $NP \rightarrow DTDT + JJ + JJ + NN$, ale i pravidlo pro případ, že ve jmenné frázi nebudou dvě, ale pouze jedno adjektivum $NP \rightarrow DTDT + JJ + NN$. Tato dvě pravidla jsou pak ale rozdílná. Dále se hůře hledá spojitost mezi těmito přepisovacími pravidly, a tím se hůře hledá vzor v podobnosti. Proto není konstrukce syntaktického podstromu nijak jednoduchá.

Věta 1 (Indukovaný strom)
Definujme strom t jako indukovaný podstrom stromu s, pokud existují identické uzly pod t i s a ponechávají si tyto uzly vztah rodič-potomek ve stromu t.

Věta 2 (Vložená hrana)
Definujme vloženou hranu e stromu s jako dvojici uzlů se vztahem předek-následník. K-vložená hrana podstromu t ze stromu s je množina indukovaných podstromů z s, které se dají spojit nejvíce k hranami (bez vztahu rodič-potomek), kde k je dané libovolné celé číslo.

Protože by množina takových k-ee podstromů byla veliká (rostla by exponenciálně s velikostí vzorů), je potřeba vybrat pouze takové podstromy, které dávají

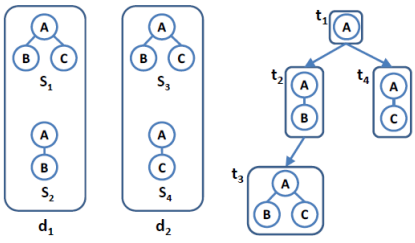
vzory s větším významem. Proto určíme hranici významnosti, neboli hranici minimální podpory θ. Tím se vyhneme již výše zmiňovanému problému přeučení (overfitting). Podpora množiny t budiž pak značena $sup(t)$ a vyjadřuje počet vět z učebního textu obsahujících vzor t. Vzor t je potom významný, pokud se vyskytuje vícekrát než je spodní hranice θ, tzn. pokud $sup(t) \geq \theta$. Takto předejdeme přeučení. Vyzkoušel jsem i časovou náročnost takového postupu. Výpočet byl tak časově náročný, že zcela zjevně přesahoval časovou náročnost jiných metod.

Věta 3 (Frekvence výskytu vzoru)
Definujme frekvenci výskytu $k-ee$ podstromu t v dokumentu d jako počet syntaktických stromů (analyzovaných vět) v d, které obsahovaly vzor t nad všemi větami v článku d. Značíme $freq(t, d)$.

Jednoduše řečeno počet vět v dokumentu obsahujících daný vzor podělíme počtem všech vět v učebním dokumentu. Dále můžeme pracovat se získanými významnými vzory jako s vektory. Pokud jsou všechny vzory uložené v množině $P = t_1, t_2 \ldots t_n$, pak celý dokument d lze vyjádřit jako vektor
$d = (freq(t_1, d), freq(t_2, d) \ldots freq(t_n, d))$.

1.10.6 Konstrukce vhodného syntaktického stromu

Jak již bylo zmíněno v textu výše, musíme udržet některé vlastnosti stromu. Hlavní problém je zkonstruovat strom tak, aby se vzor t opakoval častěji než θ. Autoři této metody navrhují tzv. "Pattern-growth approach". Není v něm potřeba nalezený vzor znovu vyhledávat v celém učebním textu, aby se zjistilo množství výskytů. Metoda prostě přidává k nalezenému vzoru další, často se vyskytující uzel. Základem je, že pokud se daný uzel nenachází často v celém textu, nebude se zde nacházet často, ani pokud ho připojíme k celému vzoru. Konstrukce vychází ze syntaktického stromu vzešlého z NLP analýzy.



Obrázek 4: Vlevo dva různé články d_1 a d_2 se syntaktickými stromy. Vpravo po spojení při $\theta = 0.5$ a $k = 0$.

1. Najdi 1-frekventovaný k-ee podstrom t v učebním množině D.
2. Promítni všechny postfixy objeveného t do nové množiny D_t. Postfix výskytu t v syntaktickém stromu s je les uzlů z s po výskytu t.

3. Najdi častý uzel v v D_t, který může být připojen nejvíce vpravo od t jako k-ee podstrom. Tím, že je připojený uzel často se vyskytující, nemusíme znovu zjišťovat, zda je nově vzniklá množina často se vyskytující v celé původní množině.

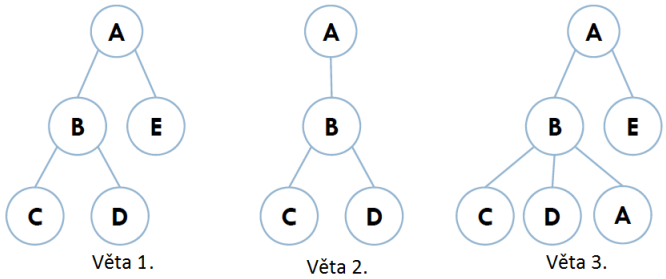
4. Opakuj krok 2. pro další vzor t.

Tím postupně ubývá vzorů t a vzory t jsou větší a větší.

Malý ilustrativní příklad je i na obrázku 4. Zde vidíme proces metody popsané v krocích výše. Vzniká 0-ee podstrom při $\theta = 0.5$. Každý ze čtyř vzorů je zapsán v pořadí, jak vznikal. Nejdříve tedy hledáme 1-frekventované vzory. Těmi jsou t_1, t_5, t_6. Vybereme t_1 jako začátek a postupně připojujeme frekventované uzly. Těmi jsou uzly B a C. Přes přidání uzlu B připojíme t_2. A tak se pokračuje rekursivně dále.

1.10.7 Vlastnosti k-ee stromu vzhledem k obsaženým informacím

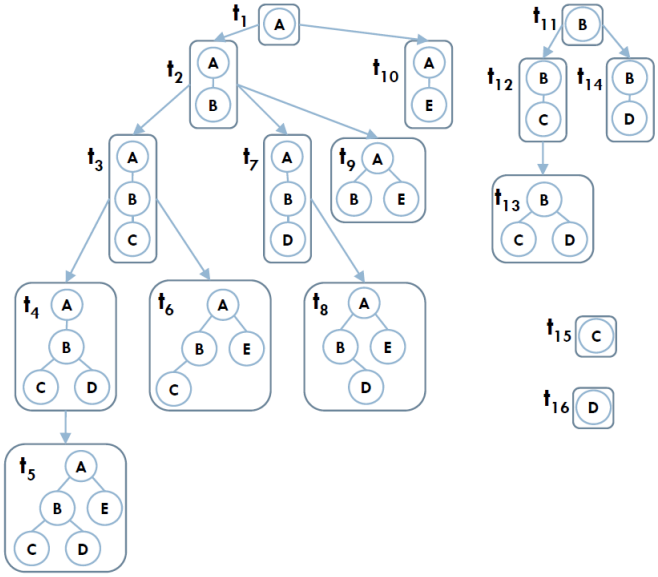
Protože vytvořený strom zahodí mnoho syntaktických vlastností, je potřeba formálně ověřit, že takovéto zahozené informace nezmění zásadně výsledek. Dále je třeba zjistit, zda některé další informace není možné v rámci vyhodnocování zahodit, a tak urychlit následné srovnávání. Autoři zavádějí "kvantizovaný informační zisk"(binned information gain, BIG) za pomoci entropie, díky kterému je na základě numerických vlastností možné ořezávat některé vzory metodou branch-and-bound.



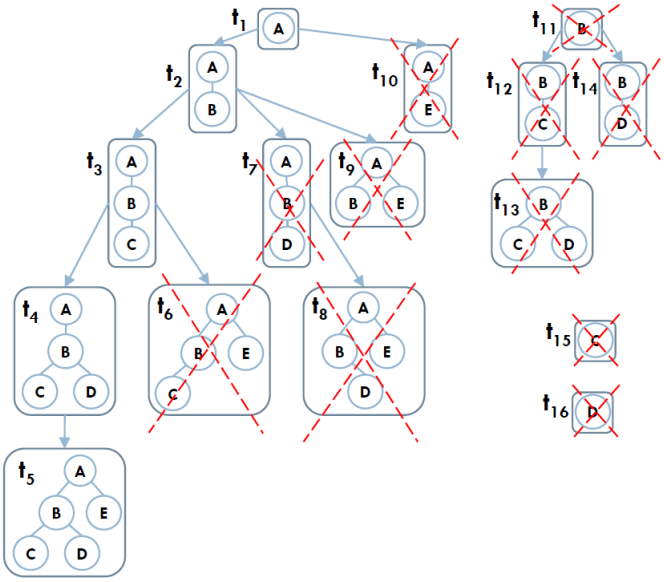
Obrázek 5: Zde je příklad tří vět na grafu s uzly. Na dalším obrázku 6 pak je vidět, jak takové tři stromy narostou bez ořezávání na velké množství supervzorů.

Věta 4 (Binned conditional entropy a Binned information gain)

Máme uživatelem definované číslo n. Dále rozdělíme interval $[0, 1]$ *frekvence výskytu vzoru t v dokumentu po n částech tak, aby platilo pro každé p:* $p_1 = [0, \frac{1}{n}), p_2 = [\frac{1}{n}, \frac{2}{n}) \dots p_{n-1} = [\frac{n-2}{n}, \frac{n-1}{n}), p_n = [\frac{n-1}{n}, 1]$. *Pro danou část p a pro m tříd* $C_1, C_2 \dots C_m$, *definujeme BCE vzoru t jako*
$H(C|X) = -\sum_{i=1}^{n} = P(X \in p_i) \sum_{k=1}^{m} = P(C_k|X \in p_i) \log p(C_k|X \in p_i)$ *a BIG vzoru*



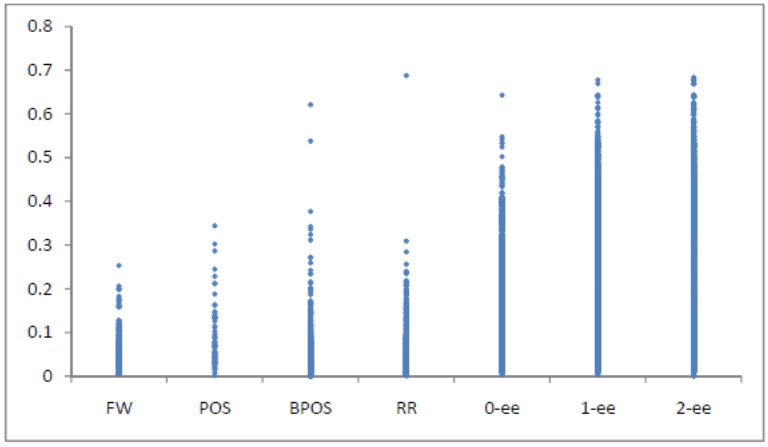
Obrázek 6: Na tomto obrázku je vidět, jak rychle nabývají nové vzory. Proto se používá upravený algoritmus CMTREEMINER od stejných autorů [1] k získání pouze frekventovaných vzorů postupným nepřidáváním málo frekventovaných uzlů, resp. jejich odebíráním ze zvažovaných kombinací.



Obrázek 7: Zde jsou vidět ořezané, nepotřebné podstromy vzorů. Je na nich přímo vidět, že se ve zdrojových datech neobjevují, nebo nesplňují požadovanou frekvenci výskytu. Toto je náročná část výpočtu zrychlená pouze o zmenšující se počet samostatných "tokenů"pro rozšíření.

t jako

$$BIG(C|X) = H(C) - H(C|X), kde H(C) = -\sum_{k=1}^{m} p(C_k) \log p(C_k).$$



Obrázek 8: Binned information gain pro různé vybrané množiny vlastností. V pořadí zleva Funkční slova, part of speech, bigramy part of speech (viz úvod), přepisovací pravidla, 0-ee, 1-ee a 2-ee strom. Zde je vidět malý nárůst mezi 0-ee a 1-ee a prakticky nulový 2-ee.

Vzor t bude mít větší BIG (binned information gain), pokud v každé třídě bude větší rozdíl mezi frekvencemi výskytu vzoru. Tím se prokáže význam vzoru t. Na obrázku 8 je vidět hodnota BIG pro jednotlivé metody. $BIG(t)$ pak značí informační zisk pro vzor t. Horní mez pro informační zisk vzoru t a jeho odvozených (zvětšených) vzorů jako $BIG_u b(t)$. Pokud pak máme k-ee podstrom t a vyznačenou část p, pak značíme (A, B, p) jako distribuci frekvence výskytu t, kde $A = (A_1, A_2 \dots A_n)$ a $B = (B_1, B_2 \dots B_n)$. Pro A_i odpovídá počet dokumentů v třídě C_i a pro B_i odpovídá tento počet v $p_i \in p$. (A', B', p) pak značí totéž pro super vzor t' s vlastnostmi dle následujících dvou lemma.

Lemma 5 (Vzory)
Pro všechna $k = 2, 3 \dots n$ *platí nerovnosti pro vzor* t *i* t'
$\sum_{i=k}^{n} A'_i \leq \sum_{i=k}^{n} A_i$, $\sum_{i=1}^{k-1} A'_i \geq \sum_{i=1}^{k-1} A_i$;
$\sum_{i=k}^{n} B'_i \leq \sum_{i=k}^{n} B_i$, $\sum_{i=1}^{k-1} B'_i \geq \sum_{i=1}^{k-1} B_i$

Důkaz
Protože t' je super vzor t, $\sum_{i=k}^{n} A'_i \leq \sum_{i=k}^{n} A_i$ *pro* $k > 2$.
Tudíž $\sum_{i=1}^{k-1} A_i = |C_1| - \sum_{i=k}^{n} A_i \leq |C_1| - \sum_{i=k}^{n} A'_i = \sum_{i=1}^{k-1} A'_i$, *kde* $|C_i|$ *je počet dokumentů v* C_i. Obdobně pak platí pro B_i. □

Následující lemma platí pro případ, kdy jsou pouze první dvě třídy rozdílné.

Lemma 6 (Distribuce frekvencí)

Pro danou distribuci frekvencí (A, B, p)*, nechť* (A', B', p) *je distribuce s* $A'_1 = A_1+x, A'_2 = A_2-x(0 \leq x \leq A_2)$ *a zbytek je beze změn. Pokud* $\frac{A_1}{A_1+B_1} \geq \frac{A_2}{A_2+B_2}$*, pak* (A', B', p) *dosahuje minimální podmíněné entropie, když* $x = A_2$*, jinak dosahuje svoji minimální podmíněnou entropii, když* $x = 0$*.*

Důkaz

Nechť $f(x)$ je podmíněná entropie (A', B', p) a N je celkový počet dokumentů.

$f(x) = \frac{A_1+B_1+x}{N}(-\frac{A_1+x}{A_1+B_1+x}\log\frac{A_1+x}{A_1+B_1+x} - \frac{B_1}{A_1+B_1+x}\log\frac{B_1}{A_1+B_1+x})$

$+ \frac{A_2+B_2-x}{N}(-\frac{A_2-x}{A_2+B_2-x}\log\frac{A_2-x}{A_2+B_2-x} - \frac{B_2}{A_2+B_2-x}\log\frac{B_2}{A_2+B_2-x}$

$+ \sum_{i=3}^{n} P(X \in p_i) \sum_{k=1}^{2} P(C_k|X \in p_i)\log p(C_k|X \in p_i).$

$f'(x) = \frac{1}{N}\log(\frac{A_1+B_1+x}{A_1+x}.\frac{A_2-x}{A_2+B_2-x}).$

Když $\frac{A_1}{A_1+B_1} \geq \frac{A_2}{A_2+B_2}, f'(x) \leq 0. Jinak f'(x) > 0.$ □

Následující theorem popisuje existenci horní meze u BIG a to, že je určena distribucí frekvence výskytu v prvních dvou třídách.

Věta 7 (Existence meze)

Máme daný vzor t*, jeho super vzor* i *on samotný mají mez danou podmíněnou entropií ve frekvenční distribuci* (A', B', p) *v jedné ze dvou forem*

(1) $A'_1 = A_1 + A_2$, $B'_2 = \sum_{i=2}^{n} B_i$, $B'_1 = B_1$, $B'_i = 0(i = 2, 3 \ldots, n)\, a\, A'_i = A_i(i = 3, 4, \ldots, n)$

(2) $B'_1 = B_1 + B_2$, $A'_2 = \sum_{i=2}^{n} A_i$, $A'_1 = A_1$, $A'_i = 0(i = 2, 3 \ldots, n)\, a\, B'_i = B_i(i = 3, 4, \ldots, n)$

Důkaz

Mějme rozložení frekvence výskytu $(\bar{(}A), \bar{B}, p)$ pro vzor $\bar{t}$ vzoru t s minimální podmíněnou entropii, která není ani v jedné ze dvou forem uvedené v theoremu o existenci meze. $P_i = \frac{\bar{A}_i}{\bar{A}_i+\bar{B}_i}\, a\, Q_i = \frac{\bar{B}_i}{\bar{A}_i+\bar{B}_i}\, pro\, (i = 1, 2, \ldots, n)$. Dle Lemma 2 platí buď $P_i \leq P_{i+1}$, $nebo\, P_{i+1} = 0\, pro\, (i = 1, 2 \ldots, n-1)$. Stejně pro $Q_i \leq Q_{i+1}$, $nebo\, Q_{i+1} = 0\, pro\, (i = 1, 2 \ldots, n-1)$. Potom pro všechny $i = 2, 3 \ldots n$ buď $P_i = 0\, nebo\, Q_i = 0$. Pokud by pro některá i platilo $P_i \neq 0\, a\, Q_i \neq 0$, pak $P_{i-1} < P_i\, a\, Q_{i-1} < Q_i$. Jenže musí platit, že $1 - P_{i-1} = Q_{i-1} < Q_i = 1 - P_i$ což je spor. Musí tudíž platit, že buď $P_2 = 0\, nebo\, Q_2 = 0$. Řekněme že platí $P_2 = 0$. Pak můžeme získat jinou distribuci $(\bar{(}A'), \bar{(}B'), p)$, kde $\bar{B}'_2 = \sum_{i=2}^{n} \bar{B}'_i$, ale $\bar{B}'_i = 0\, pro\, (i = 2, 3, \ldots, n)$, zbytek platí stejně jako u $(\bar{A}', \bar{B}', p)$. Protože je podmíněná entropie každého $CE(p_i) = 0(i = 2, 3, \ldots, n)$ (CE je podmíněná entropie), musí dle Lemma 1 mít stejnou, nebo minimální podmíněnou entropii jako $(\bar{A}, \bar{B}, p)$. Podle této lemma platí $\bar{A}'_1 \geq A_1 + A_2\, a\, \bar{B}'_1 \geq B_1$ ($\bar{A}'_2 = 0\, tedy\, P_2 = 0$). Když tedy $\bar{A}'_1 > A_1 + A_2\, nebo\, \bar{B}'_1 > B_1$, pak podmíněná entropie $(\bar{A}', \bar{B}', p)$ je větší než

podmíněná entropie (A', B', p), při předpokladu první formy dle theoremu a to že $P_2 = 0$ a při předpokladu že $(\bar{A}, \bar{B}, p)$ je minimální, což je ale spor (musí být stejná nebo menší, nikoliv větší). Stejný důkaz se dá provést s druhou možností theoremu pro $Q_2 = 0$. □

1.10.8 Upravená metoda sekvenčního pokrytí

Předchozí vlastnosti lze použít k úpravě metody sekvenčního pokrytí, jak ji implementovali jiní autoři.

1. Nejdříve získáme k-ee podstrom s největším diskriminačním skóre a přidáme ho do množiny vzorů.
2. Odstraníme stromy obsahující extrahovaný vzor. Spočítáme BIG skóre na zbytku databáze (tím by měly odpadnout méně významné vzory v dalším výběru).
3. Pokračujeme krokem 1, dokud je na čem pokračovat, nebo pokud nepřibudou nové vzory.

Zde je vidět iterativní postup, který je výpočetně náročnější než jednorázový průchod. Hledáme tedy možnosti, jak toho docílit jedním průchodem, kdy se najde neopakující se nejvíce diskriminační strom. Ostatní metody aplikovaly rozhodovací strom a metodu sekvenčního pokrytí jako základ pro SVM, nebo pouze rozhodovací strom.

1.10.9 Přímé získání k-ee stromů

Oproti minulé podsekci, kde bylo třeba použít extra přepočet skóre, zde tedy dojde ke spojení s prvním algoritmem z podkapitoly Konstrukce vhodného syntaktického stromu. Výsledkem bude efektivnější algoritmus.

Lemma 8 (IG stromové struktury)
Pro stromovou datovou strukturu D, $F = \{t | \exists s \in D \text{ takové že } t = argmax_{p \models s} IG(p)\}$.

Důkaz
Přímo dle definice v předchozí podsekci o upravené metodě sekvenčního pokrytí. □

Věta 9 (Branch-and-Bound ořezání)
Pokud $IG_{ub}(t) < min_{p \in A_t} IG(p)$*, potom není již žádný větší vzor t' z t v F.*

Důkaz
Neb $S_t \supseteq S_{t'}$, $IG_{ub}(t) < min_{p \in A_t} IG(p) \leq min_{p \in A_{t'}} IG(p)$. Tudíž t' nemůže být nejvíce diskriminační vzor pro libovolný strom S_t. □

$\models$	$t \models s$	Pokud k-ee podstrom t je obsažen ve stromu s.
S_t	$S_t = \{s \in D \| t \models s\}$	Množina stromů v množině dat D obsahující t.
A_t	$A_t = \{p : k-ee\, strom \| \exists s \in S_t, p = argmax_{p \models s} IG(p)\}$	Množina vzorů s nejvyšším skóre mezi ostatními stromy obsahující t.
B_t		Množina libovolných vzorů stromu z S_t.
F		Množina diskriminačních k-ee podstromů z D získané modifikovanou sekvenční metodou pokrytí.

Tabulka 2: Seznam symbolů pro zkrácení zápisu algoritmu pro získání k-ee stromů se skóre.

Věta 10 (Branch-and-Bound ořezání)
Pokud $IG_{ub}(t) < min_{p \in A_t} IG(p)$*, potom není již žádný větší vzor t' z t v F.*

Důkaz
Dle definice. □

V případě, že $IG_{ub}(t) = min_{p \in B_t} IG(p)$, nepokračuje hledání vzorů v D_t, protože všechny další vzory by měly stejné, nebo větší skóre.

Jakmile známe horní mez diskriminačního skóre super vzorů t', může se použít ořezávací metoda Branch-and-Bound. Bohužel jde o náročnou metodu. Řešení tohoto problému bylo uvedeno v kapitole výše. Toto řešení se zakládá na rozdělení či kvantizaci na konečné množství tříd po nejvíce dvou případech.

Během procesu dolování dat neznáme množinu A_t, nastavíme množinu B_t jako množinu nejlepších vzorů z S_t z Důsledku 1 a podmínkou Branch-and-Bound o vhodných podstromech. Tím získáme vhodné vzory pro každý strom.

Dále použijeme sekvenční metodu pro pokrytí. Po vygenerování vzoru t se stromy se vzorem t odmažou. Dále se použije hranice nejmenšího vhodného pokrytí δ. Stromy se tedy též odeberou, pokud je pokrývá alespoň δ vzorů. Taková hodnota δ se dá určit pomocí Lemma 3. Vyloučí se všechny stromy nedosahující daného skóre.

1.11 Náhled na filologické vlastnosti textu

Každý článek věnující se klasifikaci autorství, nebo přiřazování textu žánru, se věnuje alespoň jedním odstavcem obecným vlastnostem posuzovaného textu. Na tomto místě shrnu jejich poznatky napříč všemi texty uvedených ve zdroji. Samostatný článek na toto téma jsem bohužel nikde neobjevil, což vyplývá jednak

z "novosti"problematiky a také z toho důvodu, že většina autorů chce srovnávat své měření s kolegy.

1.11.1 Typické zdroje

Všichni autoři mnou studovaných textů používali při analýze některý z novinových celků. Obvykle jsem narážel na "TREC KBA Corpora", kolekci článků z "The New York Times"a jiné novinové kolekce. Autoři metody diskriminačního syntaktického stromu shodou okolností použili nezávisle oba tyto zdroje, respektive jejich podmnožinu.

Je poměrně zřejmé, proč se všichni autoři uchylují pouze k několika zdrojům textů. Pokud chceme srovnávat výkon (spíše úspěšnost) přiřazení, je vhodné jej provádět na stejné či podobné kolekci dat. Otázkou je, zda pak nejsou algoritmy vytvářeny na míru těmto textům. Tedy s premisou platnosti konkrétních vlastností. Chtěl bych se zde zamyslet nad některými těmito vlastnostmi. Chtěl bych tím také částečně obhájit motivy výběru textů pro praktickou část.

1.11.2 Nedostatky typických zdrojů

Autoři článku [1] se některým chybám vyhnuli. Například vybrali jednu konkrétní rubriku (kultura) a z této rubriky jedno téma (recenze filmů), a to vše z jednoho konkrétního časopisu (z čehož plyne podobný rozsah). Tím si logicky situaci zhoršili, protože články se žánrem neliší. Neliší se ani délkou. Téměř by se zdálo, že podobnější texty již nelze vybrat. Problém je v tom, že všechny tyto texty jsou psány profesionály. Každý profesionál má svůj styl psaní. Většinou je ve svém stylu konzistentní, nebo tento styl mění v průběhu dlouhých let postupně. Na druhou stranu právě výrazná podobnost všech textů vyvolává otázku, zda není algoritmus "vytvořen"jaksi na míru právě tomuto typu textů, resp. jaká by byla úspěšnost daného algoritmu při použití více heterogenního souboru textů. Přitom v praktických případech, kdy je zapotřebí určení autorství, můžeme právě očekávat spíše heterogenní texty, texty s odlišnou délkou a mnoha odlišnými styly.

Oba typy výběru textů mají svá úskalí a zároveň existují důvody pro výběr jednoho či druhého. Do budoucna bych však navrhoval, aby testování probíhalo na obou typech textů. Tedy jak těmi psané profesionály, tak těmi pocházejících od amatérů. Dále by testování mělo probíhat nejen na homogenním souboru textů, ale i na souborech textů, které jsou svým rozsahem a žánrem rozmanité.

Mějme příklad z článku [8]. Autoři zvolili od jednoho autora 100 článků (již popsané stejného žánru etc.). Dá se očekávat, že všechny tyto články budou mít společnou osnovu. Protože je to stejný článek, stejného autora jen s jiným konkrétním tématem. Autor tyto texty psal pravděpodobně ve velice podobných podmínkách. Dodržel tedy svou stylistickou konzistenci. Na druhou stranu neprofesionální bloger, např. cestovatel, by zřejmě nenapsal dva natolik stejné texty na stejné téma ani v rámci několika dnů. Kromě toho, že autor nemá pevný styl, je zároveň pod vlivem aktuálních emocí. Text psaný bezprostředně po zážitku

bude zcela odlišný od textu psaného retrospektivně. Tento předpoklad lze více či méně aplikovat na všechny laické autory.

Další problém nespočívá v algoritmech posouzení, ale v samotném NLP. Větný rozbor na gramaticky správně psaném textu, který bychom v novinách asi očekávali, bude zákonitě dávat lepší výsledky než text amatéra. Amatérsky psaný článek může obsahovat překlepy a gramatické chyby. V případě, že je článek psaný v jiném než rodném jazyce nebo je-li psán nespisovnou formou, pak bude zákonitě NLP dávat špatné výstupy, protože syntaktický rozbor se algoritmus učí na spisovném jazyce. A pokud je výstup větného rozboru mylný, nemůže být ani výsledek přiřazení přesný.

Dalším problémem je jakási střídmost textů. Novinové články v prvé řadě informují o jedné konkrétní věci. Pokud je takový článek faktografický, vystačí si s poměrně chudší slovní zásobou a větnou konstrukcí. To se může zdát jako nevýhoda, protože texty skladbou splývají. Problém zde pak není v samotném vyhodnocení stylistiky, ale v sestavení syntaktického stromu. Rozvité věty NLP vždy vyhodnotí hůře, než jednoduché informativní věty. Autoři textu [1] se této výtce poněkud vyhýbají tím, že zvolili v podstatě nejuměleˇctější text, jaký lze v novinách najít, přesto stále převážně informativní. Ideální text na větný rozbor pro NLP a zároveň projevení "větné individuality". Na jednoduchých informativních větách se ale stylistika autora nikdy neprojeví.

2 Řešení a návrh implementace

2.1 Implementace

2.1.1 Platforma

Většina článků věnujících se rozboru textů nezmiňuje svoji platformu buď vůbec, nebo jen mimochodem. Protože nejčastěji pracuji v platformě .NET v jazyce C#, vybral jsem si právě ji. Ve své předchozí diplomové práci jsem pracoval s výpočty na obrázky, k čemuž jsou vhodnější platformy a C# je jazyk poměrně pomalejší a rozsáhlý, tudíž méně vhodný. .NET je platforma, která umožňuje práci ve více jazycích v rámci jednoho projektu, takže bych případně v projektu mohl implementovat dle vhodnosti každý modul v jiném jazyce.

Po zkušenostech s implementací na tomto konkrétním projektu bych příště zvolil buďto jazyk Java, nebo ještě spíše jazyk Python. Java je od C# poměrně málo odlišná, ačkoliv .NET je robustnější a v mnoha ohledech se v něm dá programovat i přehledněji (třeba díky properties nebo lambda výrazům, nebo obecně omezené zpětné kompatibilitě). Hlavní důvod pro výběr Javy by zde ale byl kvůli velice dobré základně knihoven pro práci s přirozeným jazykem. Jednak Stanford NLP toolkit [29], který by se dokonce dal použít pro český jazyk (po přeučení daty z Karlovy Univerzity), a druhá open source knihovna OpenNLP [28], kterou jsem vybral i já. Dále v návrhu popisuji, jak jsem tuto knihovnu použil pro C#.

Důvodem pro použití jazyka Python je, že pro něj existují některé přímo vložené nástroje pro NLP (NLTK - [30]). Jedná se nejen o nástroje pro větný rozbor, ale i pro získání kořenů slov, synonym, antonym, tokenizace vět či tokenizace slov. Dále je v NLTK přímá podpora pro načítání online textů (Gutenberg Project - [31] aj. asi 22 zdrojů). Třeba mj. Reuters Corpus, což je asi 22 MB textu formátovaného do pseudo XML tvaru. Bohužel zrovna třeba Reuters Corpus má označeného autora jen asi u každého stého článku. Takový zdroj dat je pro tuto práci tedy nevhodný. Nicméně celkově by taková platforma byla pro příště mnohem lepším výběrem už jenom kvůli zázemí nástrojů a fór věnujících se této tématice.

Z počátku jsem předpokládal, že program bude zvládat jak anglické, tak české texty. Změnu jsem neprovedl proto, že by to nebylo technicky možné (místo OpenNLP bych zvolil Stanford NLP a data z UK, slovníky pro OpenOffice jsou i české). Problém spočívá v nutnosti sehnat dostatek českých textů. Při benchmarku výkonu jsem používal Bibli (ze 17. století). První testy jsem pak prováděl na divadelních hrách Oscara Wilda a Williama Shakespeara stažených ze stránek projektu Gutenberg [31]. Nicméně pro větší testy je potřeba mnohem více textů a shánět je dvojjazyčně by mohl být problém. Benefit by naopak spočíval v porovnání, který jazyk je jednodušší pro detekci. Ale to by vyžadovalo identické texty ve dvou jazycích s odfiltrovaným vlivem překladatele (dva překlady nejsou nikdy stejné).

2.1.2 Vlastní triviální metoda

Nejdříve jsem stál před nutností implementovat základní funkce, protože jsem měl v plánu srovnání i se zástupcem triviálního přístupu. Ve zdrojovém kódu ho označuji jako naivní metodu. S tím se pojilo prostudování a výběr některých základních atributů textu. Jejich výběr a způsob výběru jsem popsal v teoretické části. Potíž byla v nutnosti sehnat dobrý slovník, který by umožňoval kontrolu pravopisu, nabízel synonyma a přitom byl dostupný v plném rozsahu i bez poplatku. Jako vždy se nejlepší možností ukázal open source. Získal a použil jsem tak stejné slovníky, které se používají v open sourcovém kancelářském balíku Open Office [32]. A protože se s nimi nedá pracovat na přímo, použil jsem knihovnu pro práci s těmito slovníky NHunspell [33]. Slovníky i knihovna jsou pod licencí GPL/LGPL/MPL.

Slovníky obsahují thesaurus, tedy slovník synonym. Knihovnu používám jen pro test spisovného slova (spellcheck), získání kořene slova a získání synonym. Protože jsem nestudoval zdrojové kódy této knihovny, psal jsem většinu věcí tak, jako kdyby tato knihovna nebyla efektivní z hlediska rychlosti. Proto jsem pokaždé použil 3 dotazy na jedno slovo (na zmíněné vlastnosti), a to tak, že jsem tento dotaz vedl pro každé unikátní slovo, nikoliv pro každé slovo nalezené v textu. Tedy nejdříve jsem sestavil tabulku všech slov slovní zásoby. Protože málokteré slovo se v textu objeví pouze jednou, dá se očekávat minimálně poloviční úspora. Z reálných testů ale vyplývá, že poměr všech slov k velikosti slovní zásoby je cca 1:10 (průměrné slovo se tedy opakuje desetkrát), avšak velice roz-

dílně autor od autora a text od textu. Zásadní je také délka a konzistence textu. Tento poměr je koneckonců jedno z kriterii naivní paralelní metody.

Nyní se budu věnovat problému, jak udržovat slova v paměti a jak se slovy pracovat. Čtení souboru s textem probíhá pouze jednou a měření času při něm neprobíhá. Všechny metody tak pracují s textem, který už byl do paměti načten. Měření času začíná vždy, když začnou přípravné práce zpracování textu pro daný algoritmus. Pokud si držím slova v paměti, držím je vždy formou hash tabulky. Nicméně u každého slova je třeba držet si určité kvantum informací, například počet výskytů nebo jeho synonyma.... Proto je v hash tabulce hodnota hashe určená stringem (nativní hash hodnota je daná přímo platformou .NET), ale hodnota, kterou najde, není string, ale instance třídy strint (STRing + INTeger). Obsah třídy strint popisuji v programátorské příručce, ale dá se říci, že drží string slovo a int počet výskytů a některé další hodnoty. Pokud tedy v načteném textu objevím další lexém, vypočtu hash, zkontroluji výskyt a poté buď zvýším počet výskytů v objektu strint, nebo jej do tabulky přidám s novým objektem inicializovaným na počet výskytů 1. Tím se mi značně zrychlil průběh vytváření databáze slov. Původní průchod Biblí trval asi 2 hodiny. Po změně struktury ze seznamu a pár dalších úprav klesl čas průchodu asi na 5 sekund. Stejný princip platí pro databázi $n - gram$ů, kořenů slov....

Nakonec jsem zjišťoval, která část metody je nejpomalejší v toolkitu ANTS Performance Profiler. Jednoznačné zpomalení se vyskytlo u kontrolování synonym, kde algoritmus trávil 30 % výpočetního času.

Algoritmus jsem konstruoval s ohledem na přehlednost kvůli počtu vláken. Postup je následující: jedno vlákno je vyčleněno na GUI. Při volbě načtení libovolného textu vzniká nové vlákno, které načte daný soubor do paměti. K jednomu souboru náleží vždy vlastní vlákno. Toto vlákno po načtení spustí další 2 vlákna, z nichž každé počítá jednu vlastnost. Některá z těchto vláken předpočítávají potřebné atributy, proto po dopočtení pouští další vlákno. Těch je dohromady dalších 6. Tudíž při výpočtu jednoho textu běží až 8 vláken současně. Mezi sebou se synchronizují bariérou. Jakmile se bariéra naplní, text je spočítán. GUI podporuje zobrazení vlastností textu, proto ještě každé vlákno nezávisle po svém ukončení dává vláknu GUI informaci, že může vypsat vlastnosti textu, ač ještě některá nejsou hotová. Nicméně poslední vlákno zajistí, že se hotová statistika vypíše (opět přes vlákno GUI).

V okamžiku, kdy si uživatel vybere možnost zjistit autora neznámého textu, se spustí vyhodnocení daného textu. Po dokončení se spustí algoritmus srovnání. Nejdříve se ale musí zkontrolovat bariéra všech autorů. Protože srovnávám autory, a nikoliv jednotlivé texty, má každý autor ještě jakési paralelizované sloučení vlastností textů. Protože se toto sloučení podobá tomu, které probíhá na každém textu, je i tento postup obohacen vlákny, nicméně jsou již jen 4 a výpočet je značně rychlejší, protože celý text je předzpracovaný do podoby hash tabulek. Když jsou tedy všechny bariéry autorů hotovy, spustí se samotná metoda pro srovnání, jejíž výstup je opět předán vláknu GUI, aby si daný výsledek zobrazilo. Čas nutný pro výpočet metody se počítá tak, že se vezme nejdelší čas výpo-

čtu určitého článku autora a pak se přičte délka výpočtu spojení všech textů autora. Za výsledný čas metody pak používám nejdelší čas výpočtu ze všech autorů, ke kterému přičítám čas výpočtu přiřazovaného textu a čas metody pro srovnání.

Původně jsem měl metodu zpracovanou tak, že každé shodě v nějaké vlastnosti mezi autorem a posuzovaným textem (s tolerancí) přiřadím nějaké množství bodů. Za významnější parametry pak dostával autor více bodů, za méně významné a často chybně vyhodnocené vlastnosti byl odměněn autor méně body. Nakonec jsem přestoupil na metodiku zařazení těchto základních vlastností jako vektor pro vstup SVM.

2.1.3 Kompresní metoda

S přidáním kompresní metody se pojí také změna návrhu struktury programu. Původně jsem počítal s tím, že budu mít de-facto pár tříd podle srovnávacích metod, každá se svým algoritmem a nástroji. Se vzájemnou komunikací jsem počítal tak, že se jedna metoda se spustí po ukončení jiné metody. Na problém jsem narazil již při přidání druhé metody. Naštěstí má každý text svoji třídu pro článek a každý autor svoji třídu pro autory. Řešil jsem tak onu potřebu pro pečlivé strukturování kódu kvůli přehlednosti u složité paralelizace minulé metody. Pouštění jedné metody po druhé mělo spočívat v malé metodě formuláře. To nebyl zásadní problém v komunikaci logika-grafika. Zásadní problém nastal v okamžiku, kdy se některé průběžné výsledky měly dostat k jiné metodě, tak jak je tomu u u třetí metody. Nicméně i synchronizace mnoha vláken musí skončit v puštění nové metody, a to až po úplném konci činnosti, aby jedna metoda neubírala výpočetní prostředky následné metodě. Proto jsem si musel přidat novou třídu, která řídila spouštění jednotlivých metod tak, aby spolu neinterferovali.

Samotná kompresní metoda měla svůj vývoj. V teorii jsem popsal, že výběr algoritmu prošel určitým testem. Abych nemusel implementovat všechny známé nebo nejpoužívanější algoritmy, použil jsem knihovnu, která tentokrát poskytuje přímo sílu výpočtu. Protože bylo třeba vybrat knihovnu, která má ozkoušené výsledky a zároveň je rychlá, vybral jsem knihovnu programu 7-Zip ([35]).

V tuto chvíli jsem měl jasně načtenou teorii. Použití knihovny díky sporadické dokumentaci bylo poněkud problematické, avšak nakonec jsem se dostal k výsledku, který jsem potřeboval. Chtěl jsem se zejména vyhnout tomu, aby se jakékoliv mezivýsledky nebo výsledky zapisovaly na pevný disk. To by bylo zásadní zpomalení algoritmu. Řešení se naskýtala dvě. První bylo, že bych při zápisu a čtení informace z disku prostě vypnul programové stopky. Toto řešení je čisté z pohledu měření, ale existuje jedno lepší řešení, a to je zkomprimovat obsah do paměti RAM. Poupravil jsem tedy vstupní stream tak, aby šel z paměti, a ne z pevného disku. Knihovna 7zipu si s tím poradila dobře a výsledek jsem stejným způsobem dostal jen do paměti.

Výpočet se pouští pro každého autora dvakrát. Jednou bez přiřazovaného textu a jednou s ním. Výsledkem jsou dvě pole v paměti. Změřit jejich velikost a odečíst je trvá zanedbatelný čas. Kompresní metoda má tak sama o sobě jednoduchou implementaci. Těžší částí byl výběr knihovny a algoritmu.

2.1.4 SVM s kořeny slov

SVM je poměrně náročný algoritmus pro strojové učení. Protože jsem chtěl použít nějaké často používané implementace, vybral jsem již hotový program SVM^{LIGHT} [3]. Nevybral jsem tuto implementaci náhodou. Jednak je psaná v jazyce C, a je tudíž velice rychlá. Navíc ji používali i autoři textu o diskriminačním syntaktickém stromu. Tato implementace má navíc za sebou mnoho odborných článků jejího tvůrce T. Joachimse a z některých z nich jsem čerpal taktéž v teoretické části (např. [3]). Samotná implementace je běžně dostupná na webu autora (http://svmlight.joachims.org/). Vybral jsem pro SVM lineární jádro - stejné, jaké bylo použito v metodě diskriminačního syntaktického stromu. Jak tento kernel vypadá, je popsáno v teoretické části.

U této metody jsem potřeboval využít nástroj z triviální metody, a to vytváření seznamu kořenů slov. Abych ušetřil výpočetní čas, přejímám tuto tabulku z triviální metody. Tabulku přejímám z toho důvodu, že vstupem SVM je vektor vlastností. Zde bude vektorem vlastností frekvence výskytů daného kořene u daného autora. Po dobu zápisu souboru jsem zastavil programové stopky. SVM^{LIGHT} má výborně a velice pečlivě napsanou dokumentaci. Proto nebyl problém aplikaci využít. Struktura souboru je taková, že každý příklad začíná buď "+1", pokud je příklad pozitivní, nebo "-1", pokud je příklad negativní. Za tímto potom přichází vektor ve formátu "číslo_vlastnosti:hodnota", oddělovačem prvků je mezera. Číslo vlastnosti jsem určil tak, že jsem všechny kořeny všech článků převedl do seznamu, tím jsem jim dal pevné pořadí, které začíná od 0, a přičítám k němu 1. Tím pádem první a nejčastější kořen má číslo 1. Protože každý kořen má u sebe informaci o výskytu, není problém přiřadit hodnotu, která je v intervalu $(0-1]$. Pokud se daný kořen u autora nebo v porovnávaném textu nenachází, stačí danou vlastnost vynechat a SVM^{LIGHT} ji bere automaticky jako 0.

SVM zde využívám binárně, což znamená, že pro každý článek autora vytvořím "+1"příklad z jeho textů a "-1"jako texty všech ostatních (každý článek jeden "-1"vektor). SVM umí i multiclass, ale protože autoři diskriminačního syntaktického stromu zvolili stejnou taktiku, postupoval jsem tak zde i já. Učení tak probíhá pro každého autora zvlášť na jiném modelu. Vektory jsou samostatný soubor. Oproti zdrojovým textům je ale tento soubor velice malý, tudíž zápis neprobíhá dlouho. To je rozdíl oproti kompresní metodě, kde by se zapisoval celý komprimovaný text. Vektor pro SVM je několikanásobně menší než původní texty. Hodnota vlastnosti je typu double v en-US notaci zápisu (s tečkou místo čárky).

Pro učení slouží "svm_learn.exe"v adresáři "SVM_files". Pochopitelně si kontroluji, zda daný soubor existuje a nebyl odmazán. Soubory zapisuji k němu jako "jméno_autora+pořadí_autora.dat". Polohu v adresářové struktuře určuji podle spuštěného programu. Vstupem je soubor typu .dat v popsané struktuře a argumenty pro jádro a parametry pro něj. Výstupem programu je modelový soubor "jméno_autora+pořadí_autora.model". Druhý výstup je na konzoli, ze které čtu stav výpočtu. Po dokončení tak hned pokračuje automaticky výpo-

čet.

Pro klasifikaci slouží "svm_classify.exe". Vstupem je soubor typu ".model"a srovnávaný text v typu ".dat". V parametrech je pak jméno výstupního souboru "output+číslo_autora.txt". Výstupem je výpis na konzoli a hlavně výstupní soubor z vstupního parametru. Ten obsahuje pravděpodobnosti, na jednom řádku vždy jednu. Hledám tedy mezi všemi výstupy tu největší. Výstup je typu double a právě tak se jej snažím parsovat podle en-US notace zápisu.

Když jsem řešil, jak budu vlastně zjišťovat stav výpočtu, neměl jsem tušení, kolik možností .NET nabízí. Naštěstí se s tímto počítá a .NET k tomu má třídu Process. Ta umí číst výstup na konzoli a zároveň vyvolá asynchronní volání v okamžiku, kdy končí spuštěný program. Také bylo třeba pracovat se soubory a adresáři. I k tomuto má .NET mnoho nástrojů. Ověřování existence souboru nebo zápis plain textu do neexistujícího souboru je pokaždé úkon na jeden řádek. Taktéž není problém k přístupu seznamu souboru ve složce a jejich smazání. Takový "úklid"je nezbytný, jinak by se adresář "SVM_files"brzy zaplnil soubory s vektory. Ač se jedná o malé soubory, není třeba je udržovat déle než v průběhu programu.

Vstupem do SVM může být libovolná vlastnost, která lze nějak kvantifikovat. Zrovna kořeny slov jsem vybral z toho důvodu, že jejich použití je obecně uznáváno jako nejvíce vypovídající oproti použití celých slov, n-gramů nebo synonym. T. Joachims však používá ve svých článcích jako vektor celá slova [3]. Autoři [1] použili pro srovnání frekvence bi-gramů a tri-gramů.

2.1.5 Metoda diskriminačního syntaktického stromu

Samotná metoda je poměrně detailně popsána v teoretické části.

Metodě předchází větný rozbor vstupních článků. Potřeboval jsem tedy již hotový nástroj pro větnou analýzu. Pro tento účel jsem si zvolil open source knihovnu OpenNLP [28]. Knihovna byla ovšem vytvořena pro jazyk JAVA. Vyžaduje například JAVA verzi StringBuilder pro rychlou práci s textem. Abych mohl používat JAVA knihovnu v .NET, musel jsem použít převodní knihovny IKVM [36]. Toto řešení může celkový výsledek zpomalit. Po přidání namespace IKVM již bylo možné plně používat tříd známých v JAVA.

OpenNLP poskytuje mnoho funkcí. Kromě slovníků synonym, antonym aj. implementuje nástroj pro dělení textu na věty. Věty však nedělí triviálně podle znaků interpunkce, ale bere v potaz i jejich kontext. Nedělí tak například zkratky s tečkou na nové věty nebo datum s tečkovou notací na jednotlivé věty a podobně.

Hlavní důvod využití OpenNLP je však snadné strojové učení a použití funkcí pro větný syntaktický rozbor. Výsledky, které OpenNLP vrací, nejsou vždy ideální, ale některé z těch chyb bylo možné očistit později v kódu. Poté, co OpenNLP projde učební data, jsou všechny články rozděleny na jednotlivé věty (tokenizace vět). Na jednotlivou větu se pak volá funkce větného rozboru. Větný rozbor se vrací v podobě stringu jakožto uzávorkovaný tokenový text. Uveďme příklad.

Původní věta (doslovně i s formátovacími chybami): "Today was almost as nerve racking as yesterday- my marks would be handed back."

Strojově provedený větný rozbor: "(TOP (S (NP (NN Today)) (VP (VBD was)(ADVP (RB almost)) (SBAR (IN as) (S (NP (NP (NN nerve) (NN racking))(PP (IN as) (NP (NNS yesterday-)))) (VP (VBP my) (SBAR (S (NP (NNS marks)) (VP (MD would) (VP (VB be)(VP (VBN handed))))))))) (. back.)))".

Jen poznámka, je zde vidět i chyba ve větném rozboru. Chybí příslovečné určení místa.

Následuje očištění na tokeny (protože samotná slova algoritmus nezajímají): "(TOP (S (NP (NN)) (VP (VBD) (ADVP (RB)) (SBAR (IN) (S (NP (NP (NN) (NN)) (PP (IN) (NP (NNS))))(VP (VBP) (SBAR (S (NP (NNS)) (VP (MD)(VP (VB) (VP (VBN))))))))) (.)))"

Na konci zůstal neznámý token ".", což je chybějící určení místa. Algoritmus není žádným způsobem toto dopočítat, proto bude token "."vyřazen, a tím vznikne chyba v přesnosti.

Následuje vytvoření syntaktického stromu v běžné datové struktuře typu strom s předkem a textem svého uzlu. Seznam těchto stromů existuje pro každý článek. Tyto stromy jsou pak vstupem pro algoritmus vytvoření k-ee stromu.

Poté spočítám výskyty k-ee vzorů ve článcích, jejich relativní frekvence výskytu (ve tvaru vektoru) je vstupem pro SVM. Tak jako i některé předchozí algoritmy i tento implementuje rozhraní IKVM.

3 Výsledky měření

3.1 Měření

Měření jsem prováděl na množině textů stažené ze serveru, kam jednotliví blogeři píší své cestovatelské zážitky. Jediným kritériem při výběru bylo, zda má autor (nebo partneři jako jeden přispěvatel) více než 100 textů. Blogy amatérů jsem zvolil záměrně. Dle mého názoru je snazší identifikovat stylistiku profesionálního spisovatele, resp. novináře. Chci tedy testovat ten nejhorší případ - nekonzistentní autor v rámci několika textů. Již při stahování blogových příspěvků jsem si pak byl vědom, že některé příspěvky jsou odděleně psány dvěma autory. V tomto případě, pokud to bylo zřejmé, jsem text o jednoho autora zkrátil. Konkrétně jsem zkracoval o autora - muže, protože ženy psaly své příspěvky delší. Delší texty se lépe hodnotí. Nicméně jsem si plně vědom toho, že některé články plynule píše více autorů (maximálně dva) zároveň. Snažím se zde tedy zjistit, kterému blogu článek patří, nikoliv které osobě. Články vznikající kontribucí více autorů jsou běžné, a přesto je požadováno tyto texty přiřadit (ač okruhu autorů).

Při měření jsem se snažil napodobit množstevní podmínky podobné původnímu článku. Neměl jsem však k dispozici rozsáhlou databázi blogů. Byl jsem tak nucen použít nejmenší skupinu, která byla dostupná i autorům [1]. Tj. 400 článků pro čtyři autory. Každý autor má 100 článků. Testování na množině blogů jsem provedl tak, že jsem pseudonáhodně vybral 90 článků a přiřadil je autorovi. Po přiřazení základu jsem pak zbývající blogové texty zkoušel aplikací přiřadit. Výsledky jsem si zapisoval pro každý text zvlášť. Zaznamenával jsem však, i

komu byl článek přiřazen omylem. Zajímalo mě totiž, zda se při atribuci zmýlí některé algoritmy stejně. Mohl bych tuto chybu pak snáze připsat inkonzistenci stylu autora. V tabulce jsou tyto společné neshody barevně vyznačeny.

3.1.1 3 autoři

NM:	B	B	C	B	C	B	C	B	B	B
KM:	A	A	A	A	A	A	A	A	A	A
SM:	A	A	A	B	A	A	A	A	A	A
DM:	C	B	A	A	A	A	A	B	A	B

Tabulka 3: V tabulce jsou shody (zeleně), neshody (červeně) a společné neshody (žlutě) pro autora A. NM - naivní metoda, KM - kompresní metoda, SM - kořeny slov + SVM, DM - diskriminační syntaktický strom.

NM:	B	B	B	B	B	B	B	B	C	B
KM:	B	B	B	B	B	B	B	B	B	B
SM:	B	B	B	B	B	B	B	B	B	B
DM:	B	B	B	B	B	B	C	B	C	B

Tabulka 4: V tabulce jsou shody (zeleně), neshody (červeně) a společné neshody (žlutě) pro autora B. NM - naivní metoda, KM - kompresní metoda, SM - kořeny slov + SVM, DM - diskriminační syntaktický strom.

NM:	C	B	B	B	C	C	B	C	B	C
KM:	C	C	C	C	C	C	C	C	C	C
SM:	C	B	B	B	A	C	B	C	B	C
DM:	C	B	C	B	A	C	A	C	B	C

Tabulka 5: V tabulce jsou shody (zeleně), neshody (červeně) a společné neshody (žlutě) pro autora C. NM - naivní metoda, KM - kompresní metoda, SM - kořeny slov + SVM, DM - diskriminační syntaktický strom.

Metoda	správných přiřazení	%	správných přiřazení	%
NM:	14	46,7	24	79,2
KM:	30	100,0	30	100,0
SM:	23	76,6	30	100,0
DM:	19	62,7	27	89,1

Tabulka 6: V prvním sloupci je název metody. Ve druhém sloupci je počet úspěšných přiřazení, ve třetím odpovídající % podíl. Ve čtvrtém sloupci je úspěšnost přiřazení po odečtení společných chyb a v posledním sloupci je odpovídající % podíl.

3.1.2 4 autoři

NM:	B	B	A	B	A	B	A	B	B	B
KM:	A	A	A	A	A	A	A	A	A	A
SM:	A	A	A	B	A	A	A	B	A	A
DM:	B	B	A	A	A	A	A	B	A	A

Tabulka 7: V tabulce jsou shody (zeleně), neshody (červeně) a společné neshody (žlutě) pro autora A. NM - naivní metoda, KM - kompresní metoda, SM - kořeny slov + SVM, DM - diskriminační syntaktický strom.

NM:	B	B	B	B	B	B	B	D	B	B
KM:	B	B	B	B	B	B	B	B	B	B
SM:	B	B	B	B	B	B	B	B	B	B
DM:	B	B	B	B	B	B	B	D	B	B

Tabulka 8: V tabulce jsou shody (zeleně), neshody (červeně) a společné neshody (žlutě) pro autora B. NM - naivní metoda, KM - kompresní metoda, SM - kořeny slov + SVM, DM - diskriminační syntaktický strom.

NM:	D	B	B	B	D	D	B	B	B	C
KM:	C	C	C	C	C	C	C	C	C	C
SM:	B	B	B	B	A	D	B	B	B	C
DM:	C	C	C	B	A	D	B	B	B	C

Tabulka 9: V tabulce jsou shody (zeleně), neshody (červeně) a společné neshody (žlutě) pro autora C. NM - naivní metoda, KM - kompresní metoda, SM - kořeny slov + SVM, DM - diskriminační syntaktický strom.

NM:	D	D	D	D	D	D	D	D	D	D
KM:	D	D	D	D	D	D	D	D	D	D
SM:	D	B	B	B	B	B	B	D	B	B
DM:	D	B	D	B	B	D	D	C	D	D

Tabulka 10: V tabulce jsou shody (zeleně), neshody (červeně) a společné neshody (žlutě) pro autora D. NM - naivní metoda, KM - kompresní metoda, SM - kořeny slov + SVM, DM - diskriminační syntaktický strom.

Autorům textu [1] vyšla přesnost přiřazování pro 4 autory v rozmezí 87.75 - 91.50 %. Výsledky "se shodným omylem"algoritmů jsem uvedl čistě pro náhled, jak různé algoritmy dělají společné chyby, ač fungují na zcela jiném principu.

Metoda	správných přiřazení	%	správných přiřazení	%
NM:	26	65,0	36	90,0
KM:	40	100,0	40	100,0
SM:	24	60,0	35	87,5
DM:	26	65,0	39	97,5

Tabulka 11: V prvním sloupci je název metody. Ve druhém sloupci je počet úspěšných přiřazení, ve třetím odpovídající % podíl. Ve čtvrtém sloupci je úspěšnost přiřazení po odečtení společných chyb a v posledním sloupci je odpovídající % podíl.

3.2 Diskuze

Z naměřených údajů i z průběžných testů, které jsem dělal v průběhu psaní, mohu potvrdit, že postup z článku [1] má vysokou účinnost "na standardní"sadě textů. Tato metoda však trpí předcházejícím procesem, kterým je zde větný rozbor. Razantně horší výsledky, než které by měly vycházet, připisuji faktorům popisovaným v teorii v kapitole popisující nedostatky typických zdrojů srovnávání textů. Rozumím tomu, že algoritmy je třeba srovnávat na podobných datech. Avšak nepřijde mi vhodné, aby všichni porovnávali své postupy na datech, která jsou víceméně ideálním případem. Horší výsledky jsou zde tedy dle mého názoru zapříčiněny zejména charakterem vybraných textů.

Pokud se podíváme na ostatní metody, zcela nejlépe dopadla kompresní metoda. Tato metoda má 100 % přesnost. Zároveň je nejrychlejší. Není závislá na jakýchkoliv jiných nástrojích jako engine SVM, slovníky, nebo nástroje pro větnou analýzu. Domnívám se, že jde o vedlejší produkt úžasného vývoje na poli kompresních algoritmů. Koneckonců vývoj těchto algoritmů běží násobně déle, než snaha o strojovou atribuci textů. Jednoduchá, a přesto geniální myšlenka meziooborového použití zde nese své ovoce.

Metody stojící na lexikálních atributech se na těchto konkrétních datech ukázaly být stejně účinné, nebo o něco účinnější než metoda syntaktická. Při měření v rámci jiných článků byly tyto metody vždy pozadu pouze o jednotky, nebo desítku procent. Znovu ovšem musím poukázat na to, jaké byly jejich testovací články. Domnívám se na základě měření, že lexikální vlastnosti textu se liší méně, než se liší syntaktické vlastnosti textu, pokud na článku pracuje přímo či nepřímo více lidí. Přesto, pokud by byl nástroj NLP schopen přesněji zpracovávat vstupní texty, obecně přesnější "syntaktický"algoritmus by tento rozdíl smazal.

Okrajovou věcí, kterou bych přesto chtěl rozebrat, je časová složitost. Jednoznačně nejrychlejší je metoda komprimační, a to i při velkých objemech textu. Metody stojící na lexikálních základech jsou velice rychlé, pokud počet sledovaných vlastností nepřekračuje cca 1000 prvků. Aplikace, tak jak jsem ji napsal, počítá všechny výskyty bigramů, trigramů, synonym apod. V tuto chvíli však nejsou vstupem pro SVM, protože se výpočet značně prodloužil na straně SVM (více než 10 000 vlastností) a přesnost se nezlepšila. Metoda diskriminačních syntaktických stromů je sama o sobě též velice rychlá. Co je ovšem

na první pohled při testování zjevné, je pomalost NLP. A protože je větný rozbor nezbytnou součástí syntakticky zaměřených algoritmů, padá tento čas právě "na vrub"syntaktického rozboru. Další výzkum, jak zefektivnit výpočet těchto algoritmů by se tak dle mého názoru měl ubírat směrem ke zrychlení strojového rozpoznávání struktury textu.

Dohromady všechny tyto mnou popsané výsledky mne vedou k zamyšlení nad využitelností těchto metod v praxi. Ponechám stranou kompresní metodu, protože není bohužel v mých silách ověřit v plném rozsahu její účinnost. Testy jsem prováděl mezi čtyřmi autory. Některé články pracovaly až s deseti autory. Článek pro metodu diskriminačního syntaktického stromu [1] uvádí svoji účinnost při sedmi autorech 87 %, při měření bigramů 78 %. Tato čísla platí pro novinové články. Jak by taková atribuce dopadla, pokud bychom například chtěli přiřazovat autora seminární práce na menší katedře, kde by srovnání proběhlo s cca 1000 autory a každý autor by měl $\tilde{3}$ seminární práce? Domnívám se, že posuzovaná metoda, pokud by vůbec někdy doběhla vzhledem k objemu textu, by dobré výsledky neposkytla. Předpokládám, že toto bude hlavní důvod, proč se používá srovnání bloků textů místo těchto pokročilých metod. V dalším výzkumu by proto bylo vhodné řádově zvýšit počet autorů a obzvláště se zaměřit na výsledky komprimační metody a metod založených na principu syntaktického větného rozboru.

4 Uživatelská příručka

Aplikaci jsem vyvíjel pokud možno s co nejjednodušším ovládáním. Proto celý program poskytuje v podstatě jen tři dialogová okna a systémový dialog k výběru souborů.

4.1 Soubory

K tomu, aby program fungoval bezchybně, je potřeba dodržet jeho konzistenci. Proto si program kontroluje přítomnost všech nezbytných souborů.

Seznam všech souborů v adresáři se spouštěcím souborem:

- "authorship classification.exe"
- "Hunspellx64.dll"
- "Hunspellx86.dll"
- "ICSharpCode.SharpZipLib.dll"
- "IKVM.OpenJDK.Core.dll"
- "IKVM.OpenJDK.Util.dll"
- "IKVM.Runtime.dll"

- "NHunspell.dll"
- "opennlp.dll"

Seznam všech souborů v adresáři dict:

- "en_US.aff"
- "en_US.dic"
- "th_en_US_v2.dat"
- "th_en_US_v2.idx"

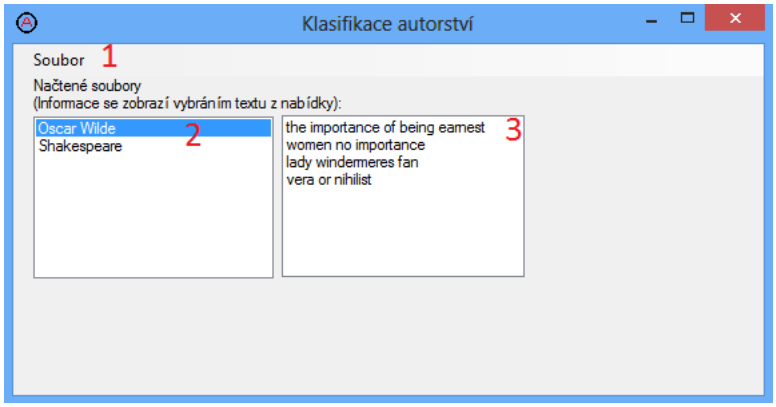
Seznam všech souborů v adresáři OpenNLP_models:

- "en-parser-chunking.bin"
- "en-sent.bin"

Seznam všech souborů v adresáři SVM_files:

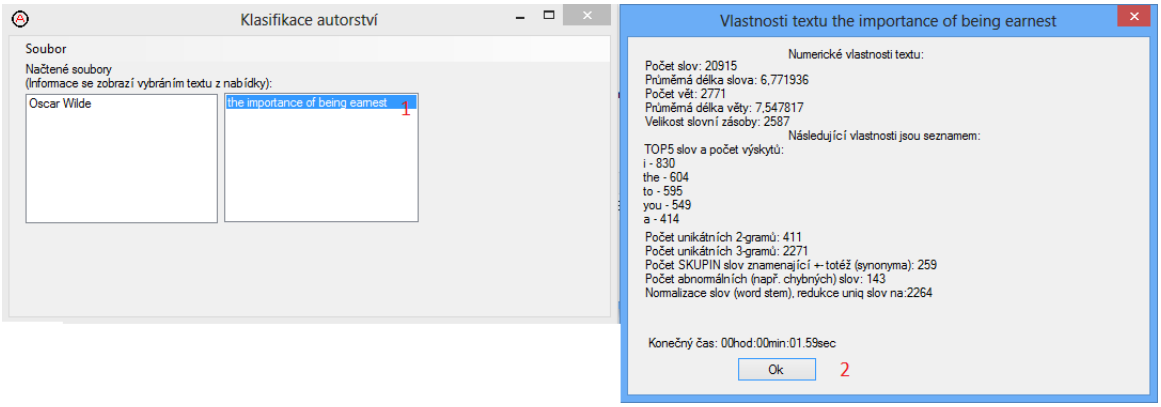
- "svm_classify.exe"
- "svm_learn.exe"

4.2 Práce s aplikací



Obrázek 9: Na obrázku je vidět příklad programu po přidání dvou autorů. Seznam autorů je v levém listu (2). Seznam textů daného autora vpravo (3). Menu se nachází pod položkou "Soubor"(1).

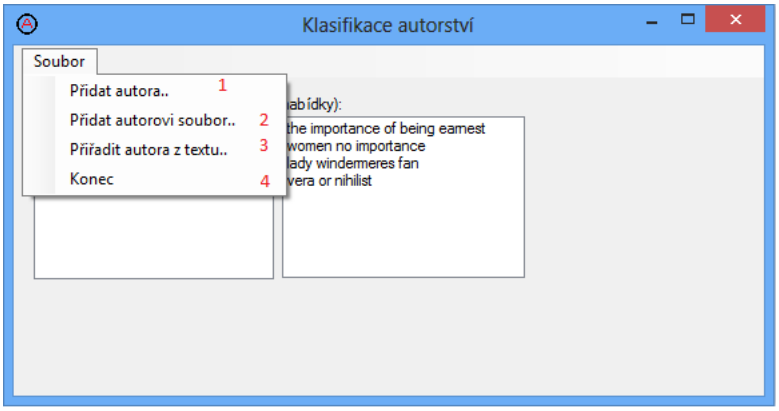
Na obrázku 9 je vidět příklad užití aplikace po přidání dvou autorů. Menu se vyvolává položkou v menu "Soubor"(1). Kliknutím na tuto položku se zobrazí nabídka, jak je vidět na obrázku 11. Kliknutím na jméno autora v seznamu autorů(2) se v pravém seznamu(3) zobrazí všechny texty, které byly tomuto autoru přiřazeny. Vybráním textu se zobrazí jeho vlastnosti, jak je vidět na obrázku



Obrázek 10: Zobrazené vlastnosti vybraného textu ze seznamu.

10. Pod seznamem autorů a jejich textů se může zobrazovat informativní text a nebo chybné hlášení v případě, že se uživatel pokusí provádět více analýz zároveň, nebo smazal potřebný soubor k výpočtu.

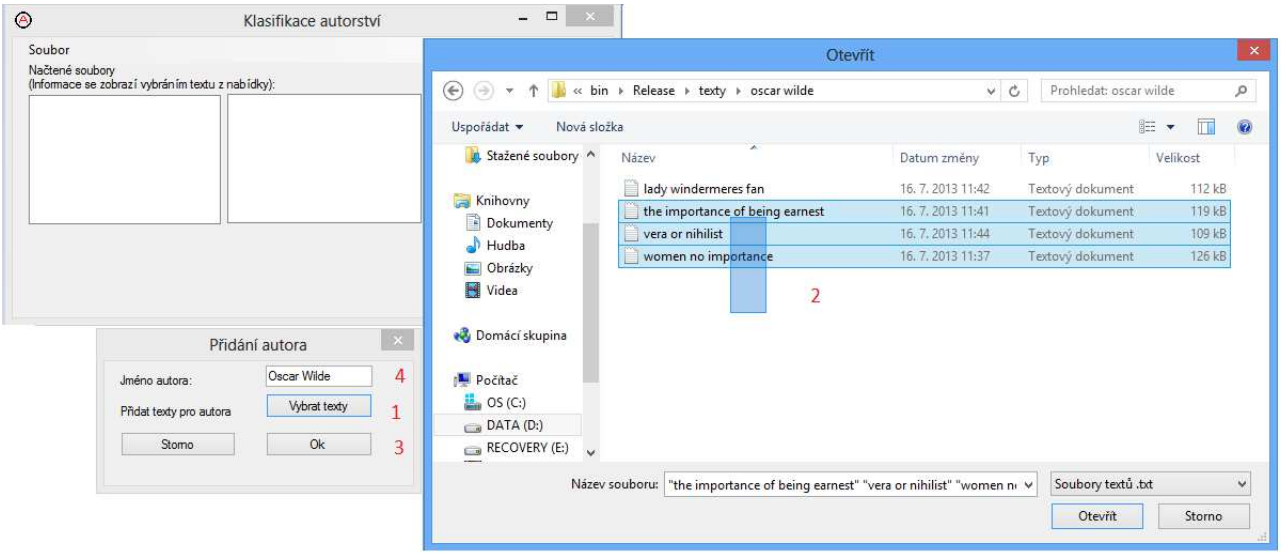
Na obrázku 10 jsou vidět detaily o vybraném textovém souboru z pole textů (1). Zobrazuje všechny vlastnosti (2), které využívá naivní metoda. Pokud je vybraný text příliš dlouhý a vlastnosti se teprve počítají, zobrazuje se hlášení o průběžných výsledcích.



Obrázek 11: Rozkliknuté menu "Soubor".

Na obrázku 11 je vidět rozkliknuté menu "Soubor". Pod položkou "Přidat autora.."(1) se skrývá dialogové okno pro přidání autora, jak je vidět na obrázku 12. Pod položkou "Přidat autorovi soubor.."(2) se skrývá systémové dialogové okno pro doplnění textů k vybranému autorovi. Není-li žádný autor vybrán, přidá se text k poslednímu přidanému autoru, existuje-li takový. Pod položkou "Přiřadit autora z textu"(3) se skrývá systémový dialog pro výběr souboru typu ".txt". Tomuto textu bude přiřazen autor v informačním okně. Položkou "Konec"se vypne program pro detekci autorství.

Na obrázku 12 je vidět dialogové okno pro přidání autora. Dialogové okno pro výběr souborů s texty autora se zobrazí tlačítkem "Přidat soubory.."(1).



Obrázek 12: Dialogové okno pro přidání autora. Dialogové okno pro výběr souborů s texty autora se zobrazí tlačítkem "Přidat soubory.."(1). Dialog pro výběr souborů na obrázku vpravo(2). Potvrzení, nebo zrušení výběru tlačítky(3). Pojmenování autora (4).

Toto systémové dialogové okno podporuje hromadný výběr pro pohodlnější přidávání textů autorovi. Nicméně pokud uživatel vybere texty a poté dialog zruší tlačítkem Storno(3), nebo křížkem, pak se soubory ani autor nepřidají. Autor se nepřidá ani v případě, že autor nemá žádné vybrané texty. Jediná možnost, jak přidat autora, je potvrzujícím tlačítkem "OK"(3). Autora je možno pojmenovat. Na jménu nezáleží, aplikace funguje, i pokud se všichni autoři jmenují stejně. V takovém případě jen není patrné, kterému konkrétnímu autoru byly články přiřazeny.

5 Programátorská příručka

V programátorské příručce uvedu seznam všech tříd s popisem významu třídy. Dále uvedu významné metody a atributy tříd.

5.1 AddAuthor

AddAuthor je třída typu Form, neboli formulář.

Nemá významné atributy. Metody pouze obsluhují tlačítka. Všechny metody mají funkčnost v logických třídách. Jedinou kontrolu, kterou metody provádí, je to, aby nebyl vstup z OpenFileDialog prázdný. OpenFileDialog je systémový dialog pro otevření souboru.

5.2 Třída Article

Třída Article reprezentuje každý jeden přidaný textový soubor. Třída nemá složité funkce, v podstatě slouží jen jako kontejner informací.

Třída Article obsahuje proměnnou typu string text, která obsahuje čistý text načtený ze vstupního souboru. Dále obsahuje informaci o tom, zda má přiřazeného autora, a pokud ano, který z nich to je. Třída Article rovněž vlastní atributy s informacemi pro výpis vlastností naivní metody.

Třídu Article jsem nevytvářel za účelem toho, aby měla nějaké vlastní výpočty. Všechny metody s algoritmy jsem dával vždy do svých vlastních tříd kromě naivní metody, která částečně přesahuje do třídy Author. Třída Article slouží pouze jako kontejner pro zpřehlednění kódu.

5.3 Třída Author

Třída Author je jedna z nejdůležitějších tříd v celém programu. Tato třída má jako jediná metody pro výpočet. Některé její metody jsou potřebné pro předání vlastností textu jiným metodám. Tato třída si s použitím třídy naive_p sesumarizuje vlastnosti všech článků tohoto autora. Třída Author se také stará o to, aby byl předán správný nutný čas výpočtu naivní metody. Tato třída také obsahuje dvě bariéry sloužící pro synchronizaci výpočtů mezi různými články.

Důležité atributy:

```
1  List<Article> articles
2  string authorName
3  Dictionary<string, strint> uq_words
4  Dictionary<string, strint> duo_gram
5  Dictionary<string, strint> tri_gram
6  List<List<string>> used_synonyms
7  List<strint> abnormal_words
8  Dictionary<string, strint> uq_stems
9  int num_w
10 int num_sentences
11 float avg_sentence_lenght
12 float avg_word_lenght
13 Stopwatch stopWatch
```

Zdrojový kód 1: Důležité atributy třídy Author

První dva atributy slouží pro uchování seznamu článků a jména autora. Následují hashovací tabulky pro unikátní slova, bigramy, trigramy a kořeny slov, seznam použitých synonym a nespisovných slov. Dále obsahuje hodnoty počtu slov, vět a průměrné délky vět a slov. Poslední položkou jsou stopky pro výpis nutného času výpočtu.

Důležité metody:

Všechny tyto metody dělají to, že vezmou vlastnosti textu ze všech instancí Article patřící autorovi dle seznamu articles a sloučí je do atributů zmíněných

```
1 public void computeNaiveStats()
2 private void uqw()
3 private void tri_gramF()
4 private void duo_gramF()
5 private void thesaurus()
6 private void spellcheck()
7 private void getStems()
```

Zdrojový kód 2: Důležité metody třídy Author

výše. Důvod, proč si i autor drží tyto informace, je, že se tak zjednodušuje výpočet. Je také možnost přidání výpisu vlastností autora. Tento výpis implementován byl, ale velice rušil při ovládání programu.

5.4 Třída Compression_p

Třída compression_p slouží jako kompletní výpočetní prostředek pro metodu přiřazování autora pomocí komprese.

Důležité atributy:

```
1 public float compressCoeficient
```

Zdrojový kód 3: Důležité atributy třídy Compression_p

Jediný atribut slouží pro uchování výsledku daného autora, který si instanci zavolal. Obsahuje rozdíl ve velikosti mezi zkomprimovanými texty a texty s neurčeným textem.

Důležité metody:

```
1 public void Zipit(string toCompare, Author au)
```

Zdrojový kód 4: Důležité metody třídy Compression_p

Jediná metoda má na starosti výpočet koeficientu. Metoda načte knihovnu pro kompresi a založí fiktivní stream z paměti RAM. Tímto streamem plní pak buffer algoritmu pro kompresi. Postupně přidá do ZipOutputStream všechny texty jako novou ZipEntry. Poté, co jsou všechny texty přidány, zruším vytvořený kompresní slovník a proces zopakuji s přiřazovaným textem. Poté oba ZipOutputStream porovnám na velikosti.

5.5 Třída Dstma_p

Třída řeší výpočet diskriminačního syntaktického stromu. Protože používá při svém výpočtu SVM, implementuje i rozhraní ISVM.

Důležité atributy:

```
1  string toCompare;
2  List<KeyValuePair<string, int>> toCompareList;
3  Dictionary<string, int> toCompareDict = new Dictionary<string, int
       >();
4  List<Author> authorsList;
5  public Dictionary<string, strDb> tokens = new Dictionary<string,
       strDb>();
6  List<syntTree> actualTrees = new List<syntTree>();
7  Dictionary<string, int> features = new Dictionary<string, int>();
8  List<KeyValuePair<string, int>> features_list;
9  methodCenter center;
10 Stopwatch dstmaW;
11 NLP nlp;
12 CultureInfo cultureInfo = new CultureInfo("en-US");
```

Zdrojový kód 5: Důležité atributy třídy Dstma_p

První properties udržují pouze autory, jejich texty a aktuálně posuzovaný text. Pochopitelně držím i syntaktické stromy v property actualTrees a seznam tokenů. Nejdůležitějšími properties jsou jednoznačně ty, které drží výskyty jednotlivých vzorů respektive později vlastností pro vektor SVM. Abych urychlil celý výpočet, provádím učení NLP pouze jednou, proto se NLP inicializuje v konstruktoru. Při tvoření vektoru pro SVM je pak potřeba používat americký zápis, proto "en-US"formát.

Důležité metody:

```
1  private List<KeyValuePair<string, int>> ddmta(List<syntTree>
       syntTrees)
2  private List<KeyValuePair<string, int>> PrefixDiscriminativeSpan(
       double minSup, List<syntTree> trees)
3  private void doSPaN(KeyValuePair<string, strDb> pattern, double
       minSup, List<KeyValuePair<string, int>> output, List<
       KeyValuePair<string, strDb>> tokensDt, int baseCount)
4  private int discriminativeP(string pattern, List<KeyValuePair<string
       , int>> output)
5  private void feedFeaturesSyntTreesForSVM()
6  private void applySyntaticTreesToSVM()
7  private void preprocess_SyntTrees(Author positive, List<Author>
       negatives, int differ)
8  private string getFormatedValue(double value, int featureId)
9  public void classify(string model, string fileName, int diff)
```

Zdrojový kód 6: Důležité metody třídy Dstma_p

První metoda začíná samotný výpočet, následující metody pak slouží přímo pro algoritmus a jeho jednotlivé iterace výpočtu. Nakonec tato třída obsahuje

metody pro přípravu a výpočet SVM a jeho vektorů.

5.6 Form MainW

Třída MainW je typu Form. Jde tedy o třídu formuláře. Na rozdíl od jiných formulářů má tento i některé metody, které plní aktivně nějakou funkci a neslouží pro pouhou obsluhu tlačítka. Pochopitelně nejde o zásah do logiky. Nejdůležitější funkcí je, že metody kontrolují elementární správnost vstupu z dialogů pro otevření souborů. Například aby se zabránilo pokusu o přidání při stornu výběru apod. Další důležité metody jsou pro příjem volání z jiného vlákna o nutnosti něco vypsat do GUI. Při inicializaci také probíhá kontrola existence vybraných souborů, knihoven. Pokud taková knihovna chybí, program zobrazí chybovou hlášku a zabrání zhroucení aplikace.

Důležité atributy:

```
1 methodCenter center
2 bool failure
```

Zdrojový kód 7: Důležité atributy třídy MainW

Mezi atributy se také nachází delegáti na funkce pro cross-thread volání. To je totiž přímo nepřípustné, a provádí se tak skrze delegáta, který zavolá vlákno GUI s delegátorskou metodou. MethodCenter je třída logiky, jakýsi rozcestník programu. Boolean failure je implicitně false, pokud nabude hodnotu true, nelze přidávat autory. Potom se nedá s aplikací vůbec pracovat a aplikace přejde do nedefinovaného stavu. Atribut je private a nabývá hodnoty true pouze v případě, když při načtení a inicializaci neproběhne úspěšně test přítomnosti knihoven.

Důležité metody:

```
1 void readFileFunc(object data)
2 void setResultLabels(string method, string solution,
3          string textName, TimeSpan ts)
```

Zdrojový kód 8: Důležité metody třídy MainW

Třída obsahuje mnoho metod obsluhujících menu a tlačítka. Vždy volají funkce v logice nebo k tomu provádí základní kontrolu správnosti. Funkce readFileFunc je pouštěna vždy ve vlastním vlákně a slouží k načtení textového souboru. Funkce setResultLabels zavolá podřízené okno s výsledkem a předá mu parametry výsledku jedné z metod. Zároveň kontroluje, zda to byla poslední metoda, která se má počítat. Pokud ano, zresetuje blokační proměnnou, takže lze přidat autora nebo přiřadit další text. Třída dále obsahuje metody pro různé druhy chybového výpisu.

5.7 Třída methodCenter

Třída methodCenter slouží, jak již samotný název indikuje, jako centrum metod výpočtu klasifikace autorství. Třída methodCenter slouží jako rozcestník metod a je použita pokaždé, když se zavolá přiřazení autora. Jde tedy o jakýsi singleton. Pokud by se přidávala další metoda, musela by se projevit i v této metodě.

Důležité atributy:

```
1 List<Author> authorList
2 List<Article> article_list
3 Article lastArticle
```

Zdrojový kód 9: Důležité atributy třídy methodCenter

Tato třída obsahuje některé další pomocné proměnné sloužící například pro synchronizaci paralelní metody a proměnných k indikaci průběhu výpočtu pro GUI. Seznamy obsahují autory a všechny články. Proměnná lastArticle nabývá hodnoty určovaného textu.

Důležité metody:

```
1 void identifyClosestAuthor(string text, string name)
2 void naiveReady(Article art)
3 void compressReady(Article art)
4 void compressDone()
5 void svmDone(int autor, TimeSpan ts)
6 void nlpDone(int autor, TimeSpan ts)
7 void dstmaDone(int autor, TimeSpan ts)
8 void someFileIsMissing(string myFile1 = "",
9           string myFile2 = "", string directory = "")
```

Zdrojový kód 10: Důležité metody třídy methodCenter

Metoda identifyClosestAuthor spouští naivní paralelní metodu. Dále vytvoří instanci Article s neznámým textem. NaiveReady je metoda volaná po konci výpočtu naivní metody. Volá ji poslední vlákno, které dorazí do bariéry. Poté se volá funkce srovnání a ta opět zavolá metodu compressReady. Tato metoda po svém skončení volá metodu compressDone. Ta opět předá parametry pro GUI s výsledkem a spustí metodu SVM (s kořeny). Když je tato metoda hotová, zavolá metodu svmDone. Tato opět nedělá nic jiného, než že spustí metodu pro výpočet diskriminačního syntaktického stromu. Po dokončení celé metody se zavolá dstmaDone, jíž výpočet končí.

Metoda nlpDone je zvláštnost. Výpočet NLP totiž trvá dlouho a, aby uživatel neměl pocit, že program nepracuje, po každém dokončení větného rozboru na vstupním textu se zavolá tato pomocná metoda, která předá aktuální informace pro výpis v GUI. Druhou výjimečnou metodou je someFileIsMissing.

Pokud v průběhu činnosti aplikace zmizel některý potřebný soubor pro výpočet, zavolá se tato metoda s parametry jména souborů a adresář a zastaví se činnost programu.

5.8 Třída naive_p

Třída naive_p obsahuje všechny metody a atributy pro paralelní výpočet některých lexikálních vlastností. Třída implementuje interface ISVM.

Důležité atributy:

```
1  string text;
2  Dictionary<string, strint> uq_words
3  Dictionary<string, strint> duo_gram
4  Dictionary<string, strint> tri_gram
5  List<List<string>> used_synonyms
6  List<strint> abnormal_words
7  Dictionary<string, strint> uq_stems
8  private int num_w
9  int num_unique_w
10 int num_sentences
11 float avg_sentence_lenght
12 float avg_word_lenght
13 object locker
14 Stopwatch stopWatch
```

Zdrojový kód 11: Důležité atributy třídy naive_p

Všechny tyto atributy jsou důležité pro výpočet. Domnívám se, že jejich názvy jsou dostatečně vypovídající samy o sobě. Přesto string text obsahuje text článku, nad kterým běží výpočet. Object locker je pak pomocný objekt pro synchronizaci. Použil jsem zde totiž vlastní bariéru, pro kterou jsem potřeboval atomický synchronizační objekt. K tomu v .NET slouží libovolný netriviální datový typ. Poslední atribut je pak nástroj pro měření uplynulého času.

Důležité metody:

Metoda read_ended je spuštěna po přečtení textového souboru. Spouští dvě nová vlákna výpočtu. Následující metody jsou opět dostatečně vypovídající samy o sobě. Prepare_vectors vytvoří text pro připojení do výpočetního souboru SVM, getValue formátuje text do tvaru pro SVM s ID vlasnosti. mergeFeatures a buildFeauteresList zajišťují unikátní číslo vlastnosti pro každý zohledněný jev textu. Metoda classify pak implementuje samotné ISVM, pro použití SVM.

V této třídě je mnoho zakomentovaného kódu. Kód je zcela funkční, avšak po testech jsem naznal, že některé (zakomentované) vlastnosti nebudou při výpočtu zohledněny, protože měly většinou malý přínos k přesnosti výsledku.

```
1  void read_ended()
2  void word_avg_count()
3  void ct_words()
4  void sentence_average()
5  void tri_gramF()
6  void duo_gramF()
7  void thesaurus()
8  void spellcheck()
9  void getStems()
10 prepare_vectors(Author positive, List<Author> negatives, int differ,
       methodCenter center)
11 private void appendVector(StringBuilder vectors, Article art, List<
       Article> sameSideList)
12 private string getValue(float number, int id)
13 private void mergeFeatures(Article article)
14 public void buildFeaturesList(List<Author> authors, Article
       toCmpArticle)
15 public void classify(string model, string fileName, int diff)
```

Zdrojový kód 12: Důležité metody třídy naive_p

5.9 Form result

Třída result je typu Form, tedy formulářové okno. Neobsahuje žádné informace ani vstupy, pouze zobrazí to, co je mu předáno k zobrazení, což jsou výsledky měření všech metod a NLP. Neobsahuje žádné atributy kromě delegáta pro předání výsledků. Obsahuje dvě metody pro zápis do labelů. Jedna je insecure, která ale fakticky nikdy nepoběží. Druhá je bezpečná a spustí se, pokud by ke GUI mělo přistoupit vlákno, které ke GUI nepatří. Tato metoda předá vláknu GUI parametry a spustí jej.

5.10 Form stats

Třída stats je typu Form, tedy formulářové okno. Neobsahuje žádné informace ani vstupy, pouze zobrazí to, co je mu předáno k zobrazení, což jsou lexikální vlastnosti vybraného článku. Neobsahuje žádné atributy, kromě delegáta pro předání výsledků. Obsahuje jedinou metodu pro zápis, ta se spustí, pokud by ke GUI mělo přistoupit vlákno, které GUI nepatří. Tato metoda předá vláknu GUI parametry a spustí jej.

5.11 Třída strDb

Třída strDb je podpůrná třída pro výpočet často se vyskytujících vzorů v textu. Třída nemá metody a v jiných jazycích by byla implementována jako struktura. C# ale nerozlišuje mezi třídou a strukturou.

Třída obsahuje atributy počtu výskytů daného vzoru, jak daný vzor vypadá a které podčásti textu pokrývá.

5.12 Třída strint

Třída strint je podpůrná třída pro výpočet lexikálních vlastností. Strint je spojením slov string a integer. Obsahuje tedy string slova a k němu počet výskytů. Protože je ale třeba k tomuto slovu znát i existenci synonym, je tato informace držena právě zde. Proto jsem nemohl použít hashtabulku, kde hashovací hodnota je string a hodnota, na kterou odkazuje, je integer s frekvencí výskytu. Atributy jsou tedy popsané, metody jsou pouze přístupové get, set nebo add.

5.13 Třída SVM_tools

Třída SVM_tools je statická třída, která je dostupná z celé aplikace. Umožňuje přístup k SVM light a zároveň zapouzdřuje nástroje SVM. Třída s algoritmem tak nemusí řídit nízkoúrovňový přístup k souborům. Taktéž třída SVM_tools zcela řídí samotný výpočet a vyhodnocení vstupů SVM. Jako jeden ze vstupů slouží třída algoritmu implementující interface ISVM.

Důležité atributy:

```
1 Directory
```

Zdrojový kód 13: Důležité atributy třídy SVM_tools

Directory je globální property, vracející přes reflexi aktuální prostředí .exe souboru.

Důležité metody:

```
1 public static void learn(string data, string fileName, int diff,
    methodCenter center, ISVM algorithmImpementation)
2 public static int elaborate(int diff, Stopwatch sw, out TimeSpan ts)
3 public static void ExecuteLearner(string svmLearnPath, string
    trainingFile, string modelFile)
4 public static void ExecuteClassifier(string svmClassifyPath, string
    testFile, string modelFile, string outputFile)
5 public static void errorExe(methodCenter center)
6 public static void clean()
```

Zdrojový kód 14: Důležité metody třídy SVM_tools

Learn je metoda pro vytvoření modelu SVM. Volá metodu ExecuteLearner, která přímo spravuje výstup konzole SVM. Totéž dělá classify a ExecuteClassify, ale pro klasifikaci. Metoda elaborate vyhodnotí výstup z textových souborů a zavolá výsledkovou funkci v příslušném objektu implementující ISVM. Metodou clean končí metoda SVM, uklízí po SVM vzniklé soubory. errorExe je metoda pro signalizaci, že chybí jeden ze dvou souborů svm_classify.exe nebo svm_learn.exe.

5.14 Třída SVM_p

Třída SVM_p je třída obsahující vše potřebné pro výpočet vektorů frekvencí kořenů a jejich využití pro vstup SVM.

Důležité atributy:

```
1 List<Author> authors
2 Dictionary<string, strint> features
3 List<KeyValuePair<string, strint>> features_list
4 int featureID = 1
5 Dictionary<string, int> nouseWords
6 Stopwatch sw
7 Article toCompare
```

Zdrojový kód 15: Důležité atributy třídy SVM_p

Seznam autorů je nezbytný proto, že metoda potřebuje stavět výstup pro celou známou kolekci textů a autorů už na počátku metody. Hashtabulka features pak obsahuje všechna slova, pro rychlé hledání je stanovena takto. Je nezbytná pro pojmenování každého slova svým číslem. features_list je pak tato tabulka převedená do seznamu a očíslovaná pro rychlou iteraci, která přes hashtabulku je mírně pomalejší. FeatureID obsahuje aktuální nejvyšší ID, začíná na čísle 1. Property nouseWords (no use words) je naplněná při inicializaci třídy. Je do ní manuálně přidáno 150 nejfrekventovanějších anglických slov[9]. Následují stopky a srovnávaný článek. Poslední property je nástroj .NET pro rychlé sestavování stringů. Tato třída je značně rychlejší než klasické operace se stringy jako "+", "+="apod. [15].

Důležité metody:

```
1 SVM_p(List<Author> aut, Article Compare, methodCenter c)
2 void SVM_run()
3 void preprocess(Author positive, List<Author> negatives,
4       int differ)
```

Zdrojový kód 16: Důležité metody třídy SVM_p

Výjimečně jsem mezi metody přidal také konstruktor. Důvodem je, že zde konstruktor nejen naplní proměnné nutné pro výpočet, přičemž hodnoty jsou odjinud, ale v tomto případě konstruktor naplní seznam nepovolených slov. Získá od triviální metody všechny kořeny s výskytem a přitom nepřipustí, aby mezi těmito kořeny byl kořen nepovoleného slova.

SVM_run je metoda rozebíhající výpočet, získá například aktuální adresářovou cestu k .exe souboru. Metoda preprocess sestaví soubory ve tvaru, jak je umí využít implementace SVM^{LIGHT}, a zapíše je na místo do složky SVM_files

podle předem daného pojmenování. Aby se žádný soubor nikdy nemohl jmenovat stejně, jmenuje se soubor jednak podle autora a navíc k němu připojuji jeho pořadí v seznamu.

5.15 Třída syntNode

SyntNode je třída pro uzel syntaktického stromu. Svou strukturou se ničím příliš neliší od jiných nevyvážených n-stromů.

Důležité atributy:

```
1 string type
2 List<syntNode> descendants
3 syntNode father
```

Zdrojový kód 17: Důležité atributy třídy syntNode

Jediný rozdíl zde viditelný je string type. V něm je uchována POS značka prezentující význam slova. Samo slovo zde ale není.

Důležité metody:

```
1 void buildSubtree(string partSentence)
```

Zdrojový kód 18: Důležité metody třídy syntNode

Metoda odfiltruje slova a ze syntaktických značek sestavuje své následníky.

5.16 Třída syntTree

SyntTree je třída pro syntaktický strom. Třída má za úkol sestavit správný popisný label a držet strukturu stromu.

Důležité atributy:

```
1 syntNode root
2 string label
3 Dictionary<string, int> tokens
```

Zdrojový kód 19: Důležité atributy třídy syntTree

Pochopitelně jako každý správný strom i tento musí mít svůj jediný kořen - root. Atribut label je typu string, tato třída si sama po sestavení stromu umí sestavit label pomocí průchodu do hloubky. V hashtabulce tokens udržuji všechny POS značky vyskytující se v tomto stromu. Jsou nezbytně nutné pro konstrukci frekventovaných vzorů.

```
1 void BuildTree(string sentence)
2 void buildLabel()
3 StringBuilder buildSub(StringBuilder sofar, syntNode me)
```

Zdrojový kód 20: Důležité metody třídy syntTree

Důležité metody:

Metoda první volá pouze metodu uzlu kořene, aby si sestavil potomky. buildLabel iniciuje sestavení popisného štítku. Přitom používá metodu buildSub, která je de facto průchodem do hloubky.

5.17 Interface ISVM

Interface ISVM implementují třídy, které používají SVM.

Interface neobsahuje atributy.

Důležité metody:

```
1 void classify(string model, string fileName, int diff);
```

Zdrojový kód 21: Důležité metody interface ISVM

Metoda implementuje vytváření klasifikačních souborů.

Závěr

Ukázal jsem, že algoritmus prezentovaný v [1] má dobrou účinnost, ale spíše na textech autorů s konzistentní syntaktickou strukturou textu. Primárně jsem však v této práci jako protipříklad provedl testy na textech i nespisovně psaných, tedy na takových článcích, které jsou dle mého názoru běžnější. Rozebral jsem některé důležité aspekty, proč je analýza na těchto textech složitější. Zároveň jsem provedl srovnávací měření s jinými metodami různých paradigmat. Výsledky těchto testů jsem porovnal a kriticky je zhodnotil. Jako nejúčinnější se nakonec prokázala komprimační metoda.

Conclusions

I have shown that algorithm presented in [1] has good accuracy, but primary with texts containing consistent syntactic structure. As counter-example I have done tests on text with inconsistent writing style and even with incorrect grammar. These kind of articles are in my opinion overall more common than the formal articles. In thesis I have discussed some important aspects of analysis on these "more difficult" articles. I have also compared measurements with methods of other paradigms. I have compared the results of the tests and showed some important differences. As the most successful method came out the comprimation method.

A Obsah přiloženého CD/DVD

Zde uvádím datovou strukturu na DVD přiloženému k této diplomové práci.

bin/
: Obsahuje instalátor programu s implementovanými metodami.

doc/
: Text této diplomové práce a všechny soubory LaTexu s obrázky použitými v textu.

src/
: Kompletní zdrojové kódy programu s implementovanými metodami.

readme.txt
: Instrukce pro instalaci a spuštění programu.

Navíc CD/DVD obsahuje:

data/
: Obsahuje testované blogové články.

install/
: Obsahuje instalátor MS .NET verze 4.0.

Literatura

[1] Kim, Sangkyum. *Authorship Classification: A Discriminative Syntactic Tree Mining Approach.* ACM, New York, 2010.

[2] Diederich, Joachim. *Authorship Attribution with Support Vector Machines.* Kluwer Academic Publishers, Brisbane, 2003.

[3] Joachims, Thorsten. *Making Large-Scale SVM Learning Practical.* MIT Press, Cambridge (USA), 1998.

[4] Stamatatos, Efstathios. *A Survey of Modern Authorship Attribution Methods.* University of the Aegean, Karlovassi.

[5] V.N. Vapnik. *Statistical Learning Theory.* Wiley Periodicals, Inc., New York, 1998.

[6] J. Rocchio *Relevance feedback in information retrival* Prentice Hall, 1971

[7] George K. Zipf *The psycho-biology of language.* Oxford, 1935

[8] Zheng, R., Li, J., Chen, H. a Huang, Z. *A framework for authorship identification of online messages: Writing-style features and classification techniques.* Wiley Periodicals, Inc., New York, 2006.

[9] Oxford Dictionary. *Oxford Dictionary.* http://oxforddictionaries.com/words/the-oec-facts-about-the-language

[10] Bing Li, Lin Zhang, Zhuangzhuang Shang, Qian Dong *Implementation of LZMA* Electronic Letters, 50/21 p1522-1524, 2014

[11] Yun Chi, Yirong Yang, Richard R. Muntz. *Indexing and Mining Free Trees* UCLA Computer Science Department Technical Report, Los Angeles, 2003.

[12] Yun Chi, Yirong Yang, Richard R. Muntz. *Mining Frequent Rooted Trees and Free Trees Using Canonical Forms* UCLA Computer Science Department Technical Report, Los Angeles, 2003.

[13] Hong Cheng, Xifeng Yan et. al. *Direct Discriminative Pattern Mining for Effective Classification* ICDE, 2008.

[14] Jian Pei *PrefixSpan: Mining Sequential Patterns Efficiently by Prefix-Projected Pattern Growth* 2001.

[15] MSDN. *Class StringBuilder efficiency.* http://msdn.microsoft.com/en-us/library/system.text.stringbuilder.aspx

[16] Antonio Miranda-Garcıa and Javier Calle-Martín *The Authorship of the Disputed Federalist Papers with an Annotated Corpus* English studies 93/3, 2012.

[17] Warren Chernaik *Shakespeare as Co-Author: The Case of 1 Henry VI* Medieval and Renaissance Drama in England, vol 27, p192-220, 2014.

[18] A. M. Turing *Computing Machinery and Inteligence* Mind 49, 1950

[19] N. Chomsky, D. Lightfoot *Syntactic Structures* De Gruyter Mouton, Berlin, 2002.

[20] Harry M. Chang *Constructing n-gram rules for natural language models through exploring the limitation of the Zipf–Mandelbrot law* Computing, vol 91, p241-p264, 2011.

[21] Tomasz J. Kozubowski, Krzysztof Podgórski, Igor Rychlik *Multivariate generalized Laplace distribution and related random fields* Journal of Multivariate Analysis, 113, p59-p72, 2013.

[22] Robert Layton, Paul Watters, Richard Dazeley *Authorship Attribution for Twitter in 140 characters or less* Second Cybercrime and Trustworthy Computing Workshop, 2010.

[23] Eli Rohn, Gil Erez *A framework for agro-terrorism intentions detection using overt data sources* Technological Forecasting & Social Change, 80, 2013.

[24] Teahan W.J, Harper D.J. *Using compresion based language models for text categorisations.* Kluwer Academic Publishers, 2003.

[25] John Burrows *'Delta': a Measure of Stylistic Difference and a Guide to Likely Authorship* Literary and Linguistic Computing, 17/3, 2002.

[26] Graeme Hirst, Ol'ga Feiguina *Bigrams of Syntactic Labels for Authorship Discrimination of Short Texts* Literary and Linguistic Computing, 22/4, 2007

[27] Ross Clement, David Sharp *Ngram and Bayesian Classification of Documents for Topic and Authorship* Literary and Linguistic Computing, 18/4, 2003

[28] OpenNLP. *Apache OpenNLP library.* https://opennlp.apache.org/

[29] The Stanford NLP Group. *Stanford NLP toolkit.* http://nlp.stanford.edu/

[30] Natural Language Toolkit. *Python - Natural Language Toolkit.* http://www.nltk.org/

[31] Guntenberg Project. *The Guntenberg Project.* http://www.gutenberg.org/

[32] Apache OpenOffice. *Apache OpenOffice.* http://www.openoffice.org/

[33] NHunspell. *NHunspell - Free Spell Checker, Hyphenation and Thesaurus.* http://nhunspell.sourceforge.net/

[34] Redgate ANTS profiler. *ANTS Performance Profiler.* http://www.red-gate.com/products/dotnet-development/ants-performance-profiler/

[35] 7Zip *7Zip* http://www.7-zip.org/

[36] IKVM *IKVM* http://www.ikvm.net/