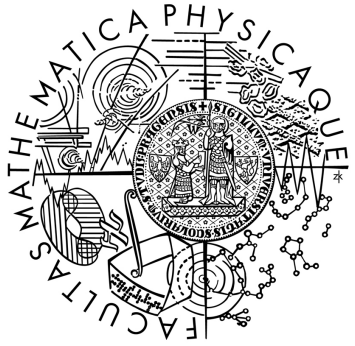
Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

BAKALÁŘSKÁ PRÁCE



David Kuboň

Framework pro extrakci informací z velkého množství jazykových dat

Ústav formální a aplikované lingvistiky

Vedoucí bakalářské práce: Mgr. Vincent Kríž
Studijní program: Informatika
Studijní obor: Obecná informatika

Praha 2014

Děkuji svému vedoucímu, Mgr. Vincentu Krížovi, za trpělivé vedení této práce a za to, že byl vždy k dispozici, když jsem si nevěděl rady.

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně a výhradně s použitím citovaných pramenů, literatury a dalších odborných zdrojů.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona v platném znění, zejména skutečnost, že Univerzita Karlova v Praze má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle §60 odst. 1 autorského zákona.

V ..... dne ..... David Kuboň

Název práce: Framework pro extrakci informací z velkého množství jazykových dat

Autor: David Kuboň

Katedra: Ústav formální a aplikované lingvistiky

Vedoucí bakalářské práce: Mgr. Vincent Kríž, Ústav formální a aplikované lingvistiky

Abstrakt: Tato práce popisuje program FAFEFI sloužící k extrakci n-gramů a skip-gramů z velkého množství jazykových dat. Řeší možnosti předání vstupních dat programu, návrh datových struktur pro reprezentaci n-gramů a skip-gramů v paměti, algoritmus jejich extrakce, paměťově úsporné varianty uložení extrahovaných dat a jejich finální zpracování do výstupních vektorů příznaků. Představuje i řadu rozšiřujících funkcí programu, jako jsou například řádkový filtr vstupních dat a modifikátor obsahu řádků, a široké spektrum konfigurovatelných parametrů — oddělovači v souborech počínaje a názvy výstupních souborů konče. Mimoto poskytuje variabilitu prováděných činností v podobě meziukládání trénovací sady dat a prezentuje nástroje pro paralelizaci výpočtu na clusteru.

Klíčová slova: n-gramy, skip-gramy, velké množství dat, strojové učení, vektory příznaků

Title: Framework for information extraction from the large language data sets

Author: David Kuboň

Department: Institute of Formal and Applied Linguistics

Supervisor: Mgr. Vincent Kríž, Institute of Formal and Applied Linguistics

Abstract: This thesis describes the FAFEFI program that focuses on n-gram and skip-gram extraction from large data sets. The thesis presents two different approaches to passing input data to the program. It also describes the design of data structures for n-gram and skip-gram representation within computer memory, the algorithm of n-gram and skip-gram extraction, memory-friendly options of saving extracted data and their final composition into output feature vectors. It also offers a variety of extra functions such as line filter and line modifier and a great deal of configurable parameters ranging from in-file separators to formatting the names of output files. Moreover, the program provides a differentiation in its activity by enabling saving data just after extraction from the train set and brings tools for cluster parallelization.

Keywords: n-grams, skip-grams, large data, machine learning, feature vectors

Obsah

Úvod

Každý den potkáváme mnohé problémy, které jsou manuálně jen velmi těžko řešitelné — například proto, že data jsou příliš rozsáhlá pro lidské zpracování nebo ono manuální řešení zvládá jen malá skupina lidí (expertů). To činí řešení těchto úloh drahým a časově náročným. Hodilo by se, kdyby počítač mohl ona rozsáhlá data projít a vyvodit z nich závěr, či se přiučit od expertů a v jejich činnosti je (bez ztráty kvality práce) nahradit. Právě takovéto úlohy řeší strojové učení.

Algoritmy strojového učení mohou být několika druhů. Vždy ale pracují s *příznaky*, což jsou určité charakteristiky popisovaného objektu. Soubory těchto příznaků, dále nazývané *vektory příznaků*, tvoří popis reálných objektů, se kterým poté algoritmus pracuje. Rozlišujeme úlohy, kde je pro vstupní data určen správný výstup (např. že konkrétní email je či není spam), které označíme jako *učení s učitelem*, a ty, kde správný výstup znám není, kterým říkáme *učení bez učitele*. Dalším kritériem dělení je schopnost přiučit se během výpočtu – tzv. *inkrementální algoritmus* – či nikoli – *dávkový*. Úlohy k řešení pak typicky bývají jedním z následující tří druhů.

Uvažme prodej ojetých automobilů. Cena takového vozu je jistě ovlivněna několika faktory — z těch se stanou příznaky reprezentující prodané auto ve světě strojového učení. Budou to například počet najetých kilometrů, značka automobilu a rok výroby. Z nich složíme trojrozměrný vektor příznaků. Dále máme k dispozici databázi doposud prodaných vozidel, která obsahuje všechny tyto údaje a navíc prodejní cenu. Cílem algoritmu strojového učení pro tuto *regresní* úlohu je naučit se odhadovat prodejní cenu ojetého vozu, který teprve má být prodán – tj. u kterého známe všechny údaje až na cenu. Takový algoritmus dokonce lze během výpočtu snadno zkvalitňovat rozšířením databáze o nově prodané automobily.

Dalším typem je úloha na klastrování — zařazování objektů do skupin s podobnými vlastnostmi, typicky během učení bez učitele. Příkladem aplikace takové úlohy je oddělování hlasů jednotlivých lidí nahraných v rušné místnosti. Předem nevíme, který z hlasů se vyznačuje jakými vlastnostmi, ale víme jistě, že kritéria pro jejich rozlišení existují.

Poslední častou kategorií jsou úlohy klasifikační, kde je nutné rozdělit vstupní data do několika tříd. Mějme texty v angličtině psané lidmi n různých národností, roztříděné do adresářů dle národnosti autora. Z nich chceme v dávkové úloze počítač naučit, jaké konstrukce se v textech jednotlivých národností vyskytují často, abychom posléze pro nové texty mohli klást dotazy typu *„Příslušník kterého národa je autorem tohoto textu?“*. K této úloze se budeme v dalším textu vracet, zabývá se jí též článek [1].

Jedná se o klasický případ aplikované úlohy z oboru zpracování přirozeného jazyka, viz [2], kde se význam strojového učení neustále zvyšuje. Dalšími příklady takových aplikací jsou spam filtry či strojový překlad. Stále rostoucí výpočetní kapacita počítačů navíc dává nemalou perspektivu rozvoji této oblasti. Pro širší informace o strojovém učení obecně doporučuji Alpaydina [3].

Právě ve výše popsané klasifikační úloze, ale i v řadě jiných, hrají podstatnou roli různé sekvence, ne nutně sousedních, slov jedné věty. V rámci uvedeného

příkladu lze, pro ilustraci, na základě autorova častého opakování trojice *podstatné jméno - přídavné jméno - přídavné jméno* v anglickém textu odlišit kupříkladu rodilého mluvčího francouzštiny od rodilého mluvčího němčiny. Mimo různých kombinací slov ve větě je užitečné vědět, i kterými slovy věta začíná a končí. To se řeší přidáním speciálních znaků na začátek (resp. konec) věty. Tedy sekvence „*znak-pro-začátek slovo*“ říká, že slovo je prvním slovem věty. Na koncích vět pak obdobně. Sekvenci délky n bez vynechaných položek nazveme *n-gram* a o *skip-gramu* hovoříme, jsou-li některé z vnitřních položek vynechány.

Při extrakci těchto údajů se obvykle pracuje s velkým množstvím dat a proto je důležité, aby byl výpočet rychlý. Velké množství dat také znamená značnou zátěž pro paměť a tedy i nutnost ji při zpracování šetřit. Smyslem programu FAFEFI — Fast Feature Extraction and Filtering — popsaného v této práci je doplnit nabídku volně dostupných programů na extrakci n-gramů a skip-gramů z velkého množství jazykových dat. Současné programy často umožňují extrakci pouze n-gramů a nabízejí jen mizivé množství nadstavbových funkcí, jako například filtraci. Existuje i výrazně komplexnější knihovna, která zvládá i následný proces strojového učení; ta je ale napsána v interpretovaném programovacím jazyku Python. Pokud uživatel nemá zájem použít její algoritmy pro strojové učení, určitě bude preferovat rychlejší výpočet. FAFEFI se snaží vyplnit tuto mezeru rychlým extrakčním programem, který nabízí obstojnou škálu uživatelských možností, je snadno konfigurovatelný a zvládá extrakci jak n-gramů, tak skip-gramů.

Jelikož jsou na sobě některé části extrakce nezávislé, je možné a vhodné ji paralelizovat, což kvůli nárokům na paměť nemusí být realizovatelné na běžných strojích. Ideální je k tomuto účelu použít cluster. Možnost distribuce výpočtu mezi počítače spojené do clusteru je užitečnou nadstavbou FAFEFI, který přináší i nástroje pro jednoduché generování příkazů právě ke spuštění na clusteru.

V první kapitole je čtenář důkladně seznámem s problematikou rychlé extrakce n-gramů a skip-gramů z velkého množství textových dat a s důvody volby jazyka C++ pro implementaci programu. Ve druhé kapitole jsou rozebrány principy návrhu programu. Třetí kapitola se zabývá technickými aspekty řešení, včetně popisu podstatných netriviálních datových struktur a hlavních algoritmů. Následuje kapitola formalizující podobu konfiguračního programu. V páté kapitole se dozvíme, jaké jsou možnosti konfigurace programu, či jak řešit případné chyby. Šestá kapitola obsahuje srovnání FAFEFI s dalšími podobnými programy.

1. Analýza

Na začátku, než se pustíme do popisu řešení, zavedeme základní terminologii. Dále v této kapitole shrneme, co chceme v programu řešit a seznámíme čtenáře se strojovým učením v rámci naší konkrétní úlohy. Poté popíšeme hrubou analýzu principů a funkčních prvků FAFEFI. Na konci kapitoly uvedeme důkladný rozbor volby programovacího jazyka, ve kterém je program implementován.

1.1 Základní termíny

Sekvenci n sousedních slov nazveme *n-gram* a n-gramu, kde je vynecháno právě k, $k<n$, ne nutně sousedních, nekrajních slov říkáme *skip-gram*. N-gramy a skip-gramy jsou definovány obecněji pro sekvence obecných položek, kterými mohou být například fonémy, slabiky, písmena, slova, lemmata či POS-tagy.

Příznak (angl. *feature*) nazýváme individuální měřitelnou vlastnost pozorovaného jevu, v případě n-gramů a skip-gramů jejich výskyt v souborech, *vektor příznaků* (angl. *feature vector*) pak n-rozměrný vektor těchto příznaků. *Třída* (angl. *class*) je prvek z množiny možných výsledků, kterému klasifikátor může/musí daný objekt přiřadit.

1.2 Zadání

Mějme netriviální množství textových dat v souborech, kde je na každém řádku právě jedna sekvence. Úkolem programu FAFEFI je z těchto vstupních dat získat všechny n-gramy a skip-gramy, o které má uživatel zájem, převést údaje o jejich výskytech na číselné příznaky a z těch pak poskládat vektory příznaků, které jsou nakonec uloženy do výstupního CSV[1] souboru. Při tom je nutné brát ohled na čas zpracování a velikost RAM[2].

Aby toto bylo možné, je třeba nalézt vhodnou reprezentaci n-gramů i skip-gramů a připravit dostatečně paměťově nenáročnou strukturu pro ukládání rozsáhlých extrahovaných dat. Je tedy nutné umožnit uživateli snadnou konfiguraci programu, která bude pohodlně a jednoznačně interpretovatelná a přitom poskytne širokou škálu možností. Podstatným rysem je, aby manipulaci zvládl i člověk bez hlubších znalostí informatiky, ale přitom aby nebyla příliš zjednodušená na úkor funkčnosti.

Chceme poskytnout škálu nadstavbových možností, kterými bude možné konkretizovat požadavky v rámci extrakce či zahrnout speciální požadavky. Například lze předpokládat, že uživatel bude chtít pracovat s různě označkovanými daty — u textu se může typicky jednat o slovní druhy, poddruhy, informace o mluvnických kategoriích atp. — a tedy je třeba umožnit extrakci i z nich.

Mimoto chceme, aby byl FAFEFI coby knihovna snadno připojitelný k rozsáhlejším programátorským dílům a aby bylo možné spustit extrakci paralelně na clusteru.

[1] Comma-separated values; souborový formát sestávající z řádků, ve kterých jsou jednotlivé položky odděleny znakem „ , “

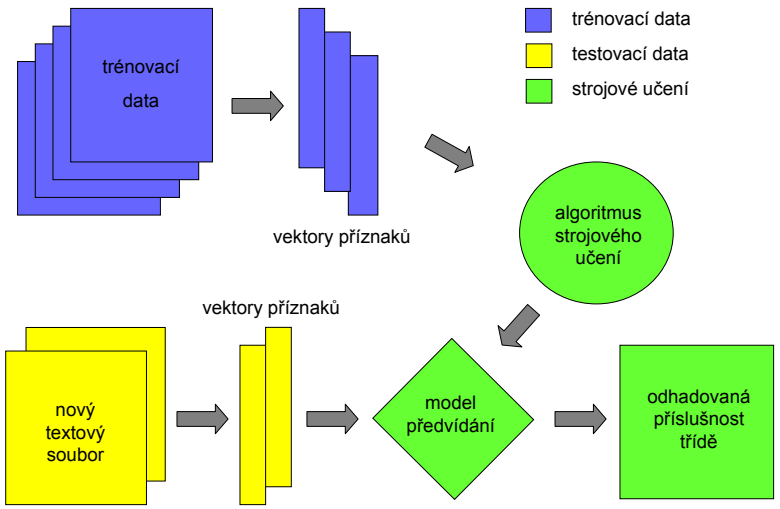
[2] Random Access Memory; paměť s přímým přístupem

1.3 O strojovém učení

Abychom lépe chápali, co od programu očekáváme, podívejme se nejprve na to, jak může vypadat zpracování výstupních dat FAFEFI během strojového učení. Pro ilustraci vezměme klasifikační úlohu o rodném jazyce autorů anglických esejí. Naše vstupní data, ze kterých se budeme učit, již musí mít svou značku — informaci o mateřském jazyku autora. Ta je před spuštěním výpočtu třídy známa a my předpokládáme, že je též správná. Jelikož předem známe cílové značky vstupních objektů, a to na základě vnějšího zásahu, jedná se o učení s učitelem, které je v oblasti zpracování přirozeného jazyka nejčastějším.

Víme tedy, že existuje funkce zobrazující z množiny objektů do množiny značek tak, že značka přiřazená objektu bude správná. Právě tuto funkci chceme strojovým učením co nejlépe aproximovat. Vědomosti o klasifikaci objektů získáváme ze vstupních dat — zde textových souborů. Tyto rozdělíme na trénovací a testovací, přičemž vhodný poměr pro toto dělení se může výrazně lišit úlohu od úlohy. Více trénovacích dat přináší větší kvalitu získaného modelu, více testovacích dat nám dá vyšší statistickou signifikanci výsledku. O datech každopádně platí, že *testovací* $\subset$ *všechna* a zároveň testovací a trénovací data mají prázdný průnik.

Po rozdělení dat z trénovací sady získáme podobu aproximace funkce přiřazující objektům jim skutečně odpovídající značku. Na této funkci pak pomocí testovacích dat na základě evaluační míry vyzkoušíme, jak moc přesný je náš model. Celý proces přehledně ilustruje obrázek 1.1.



Obrázek 1.1: Schéma výpočtu klasifikační úlohy strojového učení s učitelem

Jelikož přiřazování značek vstupním datům zpravidla vyžaduje netriviální znalosti, nemáme obvykle k dispozici dostatek expertů a s nedostatkem údajů si musíme poradit jinak. K tomu využijeme *křížovou validaci*. Při ní jsou vstupní data rozdělena na n částí - tzv. *cykly křížové validace* (angl. *cross-validation folds*). Výpočet probíhá tak, že v cyklu přes počet částí se v i-tém kroku použije i-tá sada jako testovací a všechny ostatní jako trénovací. Jasným kladem takového

kroku je díky n modelům vyšší signifikance výsledku, nevýhodou pak n-krát větší časová náročnost, což se při velkém množství dat značně projeví.

Samotné vyhodnocování kvality a úspěšnosti strojového učení v tomto textu vynecháme, protože pro program FAFEFI je tato informace irelevantní. Zájemce s ní důkladněji seznámí například Alpaydin v knize [3].

1.4 Úvodní analýza problému

Klíčovým aspeket problému extrakce příznaků z velkého množství jazykových dat je její časová i paměťová náročnost. Je třeba si uvědomit, že je nutné si pamatovat každý nalezený n-gram nebo skip-gram každé délky, a v případě skip-gramu každé varianty vynechání, navíc ještě s informací, ve kterých vstupních datech se vyskytl, případně i kolikrát — to přijde vhod později při filtraci extrahovaných dat. Jelikož celkový počet extrahovaných n-gramů a skip-gramů závisí na více faktorech — počtu žádaných druhů n-gramů a skip-gramů, rozsahu vstupních dat, šíři slovní zásoby ve vstupních datech (užití širší slovní zásoby zvyšuje počet různých n-gramů a skip-gramů) a dalších — z nichž některé (šíře slovní zásoby) jsou neaproximovatelné, nelze nalézt vhodný vzorec, který by tento počet byť i jen řádově odhadl. Můžeme sice dostat horní odhad přes délky konfigurovaných *-gramů a počet slov a vět ve vstupním souboru, ten ale bude značně nepřesný[3].

Optimální je tedy navrhnout program tak, aby zvládl data značného rozsahu, ale aby zároveň jeho časová a paměťová náročnost byla přímo úměrná objemu dat a tedy bylo možné výpočty menšího rozsahu provádět i na osobních počítačích. Je nutné připravit vhodné datové struktury, které bude FAFEFI využívat pro ukládání extrahovaných n-gramů a skip-gramů, a též i algoritmus zpracování těchto dat do finálních vektorů příznaků. Poslední jmenovaný proces je v rámci celého výpočtu časově nejnáročnější, protože vyžaduje opakované průchody všemi daty, která byla extrahována. Též přináší další extrémní zátěž paměti, protože pro každý soubor se vstupními daty si musíme připravit výstupní řádek s informací o všech extrahovaných n-gramech a skip-gramech a o jejich přítomnosti v tomto souboru. Tyto údaje sice není bezpodmínečně nutné držet v operační paměti, ale průběžné ukládání již zpracovaných řádků na disk by výrazně prodloužilo dobu běhu programu.

Je výhodné z dat odfiltrovat ty n-gramy a skip-gramy, jejichž množství nebo podoba budou mít na výsledek samý jen malý nebo dokonce žádný vliv. Tuto filtraci je možno provést již na řádcích se vstupními daty, kde odstraníme ty, které nás nezajímají. Taková operace nám zredukuje výsledný počet n-gramů a skip-gramů a tedy ušetří jak paměť, tak čas potřebný k jejich zpracování. Druhou fází, kde je možné aplikovat filtry, je zpracování extrahovaných příznaků do výstupních vektorů. Tam dává smysl odfiltrovat ty z n-gramů a skip-gramů, které se vyskytují jen zřídka — zbytečně bude ve výstupu velké množství nul, které zaberou značné množství místa a pro následné strojové učení budou mít jen mizivý význam. Tato redukce ušetří čas i paměť při závěrečném zpracování. Jsou-li obě filtrace provedeny dostatečně efektivně, přinesou výrazné urychlení doby výpočtu.

Podstatným rysem návrhu bude i způsob předání informace o uložení vstup-

[3]Pro lepší představu FAFEFI při extrakci 1-gramů (unigramů) ze sady dat čítající 3452735 slov získal pouhých 60263 různých unigramů, tedy ani ne padesátinu horního odhadu.

ních dat. Zpravidla se totiž bude jednat o nemalé množství souborů a je tedy třeba umožnit uživateli příjemnější variantu jejich zadání než vypisování cest do konfiguračního souboru. Ideální možností by byl průchod adresáři s určitým fixním uspořádáním, alternativou pak vypsání adres do CSV souboru, aby stejná data do různých výpočtů nebylo nutné kopírovat a zbytečně tak zvětšovat konfigurační soubory. Chceme-li ale podporovat více variant předání vstupu, je nutné je v paměti reprezentovat co možná nejpodobnějším způsobem, abychom nemuseli připravovat také více variant jejich zpracování.

Konečně je třeba ohlídat, aby návrh podporoval výpočet jen jednoho cyklu křížové validace nutný pro práci na clusteru, nebránil vynechání extrakce z testovacích dat a umožňoval uložení informací o rozpracovaném výpočtu po ukončení extrakce z trénovací sady.

1.5 Diskuze implementačního jazyka

Program má tři kritické body, jejichž optimalizace je alfou a omegou jeho kvalitního provedení. Jmenovitě jsou to práce s řetězci, rychlost výpočtu a paměťová náročnost.

Častá práce s řetězci směřuje k volbě některého z interpretovaných jazyků, které takové operace podporují - kupříkladu Perl či Python. Tyto poskytují široké množství funkcí navržených právě pro práci s řetězci. Interpretované jazyky ovšem bývají oproti těm kompilovaným pomalejší, což je při rozsáhlých výpočtech značné negativum, které zmíněné rozšířené možnosti jen těžko vyváží.

Jakýmsi mezikrokem mezi interpretovanými a kompilovanými jazyky je Java, která, ač je interpretovaná, díky *just-in-time* překladu dosahuje výkonu srovnatelného s jazyky kompilovanými. Navíc je nezávislá na architektuře a software v ní psaný je spolehlivý — například neumožňuje práci s ukazateli, která může být častým zdrojem chyb, a vyžaduje ošetření výjimek. Na druhou stranu je ale nešetrná k paměti. Její objekty nabalují pro naše účely zbytečné hlavičky, což je při počtu přinejmenším mnoha milionů objektů problematické. Poslední nevýhodou je garbage collector, jehož výkonnost lze bohužel jen velmi mizivě ovlivnit. Jelikož po zpracování cyklu křížové validace bude v paměti viset enormní množství zcela zbytečných dat, je dealokace paměti podstatnou složkou. Z těchto důvodů se ani Java nejeví jako optimální volba.

Přejděme tedy ke kompilovaným jazykům, kde nás zajímá především C++. Tento objektově orientovaný imperativní jazyk umožňuje jak vysoceúrovňové, tak nízkoúrovňové programování, poskytuje programátorovi široké možnosti a je hojně využíván v mnoha oblastech — od operačních systémů, ovladačů a vysoce výkonných serverů přes aplikační software až po videohry. Naše podmínky pak splňuje dokonale — svou rychlostí je známý, jeho datový typ `std::string` poskytuje vše, co pro práci s řetězci potřebujeme, a objekty neobaluje nadbytečnými údaji. Navíc má poměrně extenzivní standardní knihovnu. Více o jazyce C++ lze nastudovat v Eckelově volně dostupném *Myslíme v jazyce C++*[4].

Jako jednoznačného vítěze zvolíme tedy jazyk C++, jelikož je rychlý, paměťově relativně úsporný, zvládá veškerou potřebnou funkčnost a dává vysokou šanci na spustitelnost programu napříč širokým spektrem výpočetních strojů od stolních počítačů po servery.

Právě kvůli univerzálnosti použijeme ve FAFEFI jen standardní knihovnu

jazyka C++ a knihovnu jazyka C POSIX, která je běžnou součástí většiny distribucí UNIXu. Zároveň nepoužijeme nic, co bylo definováno standardem C++11 nebo novějším. Tento standard sice přináší ve FAFEFI využívané regulární výrazy, které je bez něj nutné čerpat z knihovny POSIX, ale v době zpracování stále není plnohodnotnou a funkční součástí většiny překladačů a tedy není příliš vhodný pro tuto aplikaci. Závislost na jmenované knihovně by stejně ani s C++11 nebyla odstraněna, neboť je potřebná i pro průchod adresářovou strukturou nutný při jedné z variant vstupu.

2. Implementace

V této kapitole nalezneme rámcové informace o implementaci FAFEFI. Rozebereme jak základní rysy programu, tak konkrétní principy podstatné pro různé nadstavby. Detailní informace o klíčových datových strukturách a hlavních algoritmech jsou v následující kapitole.

2.1 Základní informace

Program je členěn do dvou logických částí — v první z nich je zpracováván konfigurační soubor předaný uživatelem v parametrech a ve druhé se na základě této konfigurace extrahují n-gramy a skip-gramy, ze kterých se posléze připraví vektory příznaků na výstup. Vše by sice bylo možné reprezentovat pouze jednou třídou a ne dvěma, jak tomu v programu je, ale dvoutřídní reprezentace umožňuje změnit formát konfiguračního souboru dle přání a potřeb uživatele, aniž by přitom ohrozil funkčnost extrakce. Taktéž, při použití FAFEFI jako knihovny, lze konfigurací načíst vstupní data a pak je, s různými nastaveními žádaných *-gramů, předat více instancím extrakční třídy. To se hodí například při řešení úlohy, kdy ze stejných dat chceme extrahovat unigramy i bigramy, ale každé z nich zvlášť.

Podstatným implementačním rysem je snaha nepracovat s konkrétními položkami, ale se sekvenčními kontejnery `std::vector` knihovny STL [5] plněnými těmito položkami. Extrahované *-gramy dostávají podle své formy unikátní číselný kód, kterým jsou pak v kontejnerech indexovány údaje jim příslušící. Konkrétní typ *-gramu je zase jen indexem jeho definice v `std::vectoru` definovaných *-gramů a např. regulární výraz pro filtr je opět jen anonymní položkou vektoru filtrovacích regulárních výrazů. Takto je jednoduché provést vždy správný výpočet v souladu s uživatelskou konfigurací. Navíc sekvenční průchod kontejnerem i přidávání nového prvku na jeho konec jsou (amortizovaně) rychlé. Ať se uživatel rozhodne definovat n *-gramů a m regulárních výrazů nebo naopak, lišit se budou pouze velikosti jim příslušných kontejnerů.

Je důsledně dbáno o upřednostňování využívání kontejnerů standardní knihovny před navrhováním vlastních struktur. Proto je u všech analýz časových složitostí přístupů do datových struktur nutné počítat s tím, že dynamicky alokované struktury jsou uloženy na haldě — dále to již nebude zmiňováno. Výhodou tohoto přístupu je garance funkčnosti kontejnerů oproti jen nepatrně rychlejším vlastním strukturám. Navíc máme možnost využít řadu algoritmů standardní knihovny navržených právě pro zmíněné kontejnery.

2.2 Konfigurační třída

Primárním účelem třídy `Configuration` je zpracovat konfigurační soubor a v dalším výpočtu svými datovými položkami reprezentovat z něj získanou konfiguraci. V konstruktoru jsou nastaveny implicitní konfigurační hodnoty a cesta k souboru je předána jako argument spouštěcí metodě `start(std::string)`, která řídí celou konfiguraci.

Jednotlivé řádky konfiguračního souboru jsou porovnávány se sadou regulárních

výrazů popisující všechny varianty korektně zadaných příkazů. Vyhovuje-li řádek některému z nich, je zavolána metoda určená ke zpracování toho konkrétního příkazu. Není-li nalezena shoda s žádným z regulárních výrazů, je formát řádku označen za neplatný a uživatel je o tomto faktu informován.

Metody zpracovávající jednotlivé příkazy pak zpravidla sestávají z rozdělení řádku na jednotlivé položky, tzv. *tokeny*, a na jejich základě provedení příslušné konfigurace. To bývá buď nastavení konkrétní položky — např. u přepínačů — či přidání nového údaje do kontejneru, který reprezentuje všechna nastavení dané kategorie.

Datové položky, které jsou určeny k použití při budoucí extrakci, jsou čtyř typů. Předně se jedná o jednotlivé znaky reprezentující různé oddělovače, minimální hodnoty, atp. Dále jsou to řetězce popisující cesty k výstupním souborům a jejich názvy. Třetím druhem jsou sekvenční kontejnery různých položek k opakovanému užití — ať už regulárních výrazů k filtraci nebo třeba modifikátorů řádků. Do posledního typu pak shrneme všechny složitější datové struktury od definic n-gramů a skip-gramů po strukturu s cestami ke vstupním datům. Třída obsahuje i další datové položky, ty jsou ale použity jen v průběhu konfigurace.

2.3 Extrakční třída

Extrakční třída `Extraction` extrahuje dle zadané konfigurace z připravených datových souborů požadované n-gramy a skip-gramy, provede filtraci a vytvoří výstupní vektory příznaků. Její spouštěcí metoda `start(Configuration, int)` má v argumentech žádanou konfiguraci, jež obsahuje veškerá nastavení oddělovačů, cesty ke vstupním datům a výstupním adresářům atd.; a číselné označení cyklu křížové validace ke zpracování v rámci výpočtu na clusteru, nebo hodnotu -1, má-li být provedena extrakce napříč všemi cykly.

Řídící metoda postupně prochází strukturu s cestami k datům a pro každý nový soubor zavolá metodu extrahující z něj *-gramy a zařazující je do datových struktur pro uložení extrahovaných dat. Konkrétně se jedná o slovník převádějící *-gramy dle jejich typu a formy (tj. textového obsahu) na unikátní číselné identifikátory, slovník (realizovaný pomocí `std::vector`) pro převod těchto identifikátorů zpět na formu a typ *-gramu a strukturu pro ukládání výskytů všech těchto *-gramů v souborech. V rámci extrakce se ověří, že každý řádek projde filtry, a následně jsou na něm provedeny změny dané příkazem pro řádkové úpravy a až poté je předán k vytvoření *-gramů, které jsou následně uloženy.

Po zpracování všech souborů trénovací či testovací části aktuálního cyklu křížové validace je na získané *-gramy aplikován filtr pro minimální počet výskytů. Ten existuje ve dvou variantách — absolutní, kde nás zajímá celkový počet výskytů *-gramu v aktuálním cyklu křížové validace; a dokumentové, kde zkoumáme počet souborů v cyklu křížové validace, ve kterých se *-gram objevil alespoň jednou. Nakonec jsou vytvořeny vektory příznaků. V těch je na každém řádku na začátku jméno souboru následované třídou, které přísluší. Zbytek řádku pak tvoří číselné záznamy o výskytech jednotlivých *-gramů v tomto souboru, jak lze nahlédnout v tabulce 2.1. Všechny položky jsou oddělené aktuálně nakonfigurovaným oddělovačem.

Číselné záznamy mohou být tří typů — binární, poměrný a absolutní. Typ je *-gramu přiřazen uživatelem v jeho definici. Binární typ hodnotami 0 nebo 1

říká, zda se *-gram v souboru vyskytl, či nikoli. Poměrný je pro konkrétní *-gram poměr počtu jeho výskytů vůči počtu výskytů všech *-gramů vzniklých na základě stejné definice. Jelikož bude vždy mít hodnotu mezi 0 a 1, je zaokrouhlen na 4 desetinná místa a vynásoben 1000, tedy vystupuje jako celé číslo mezi 0 a 1000. Absolutní typ je pak prostým počtem výskytů daného n-gramu či skip-gramu v aktuálním vstupním souboru.

file	class	g1	g2	g3	g4	g5	g6	...
soubor1	A	72	0	5	3	21	8	...
soubor2	A	17	2	1	0	3	0	...
soubor3	B	3	9	14	27	1	30	...
soubor4	C	23	13	9	0	7	1	...

Tabulka 2.1: Příklad výstupních vektorů příznaků

Byla-li extrahována trénovací data, jsou smazány pouze záznamy o výskytech, ale formy a typy *-gramů jsou zachovány. Tyto jsou smazány pouze po zpracování testovacích dat. V případě, že uživatel pracuje s tzv. indexovými soubory, jejichž smysl a použití jsou popsány dále, se tato část procesu nepatrně liší.

Po dokončení průchodu skrz všechny cykly je výpočet ukončen.

2.4 Indexové soubory

Indexové soubory slouží k reprezentaci stavu výpočtu po ukončení extrakce trénovacích dat. Použijí se buď pro odložení extrakce z testovací sady — např. proto, že v danou chvíli tato sada není k dispozici — nebo pokud uživatel explicitně stojí o informace uložené v nich. Pro každou z extrahovaných položek se jedná právě o následující údaje v tomto pořadí:

- unikátní číselné označení *-gramu,
- forma (textový obsah) *-gramu,
- typ *-gramu (vyjádřený indexem pořadí jeho definování),
- absolutní frekvence *-gramu a
- dokumentová frekvence.

Z takovéhoto souboru lze posléze snadno jak rekonstruovat pro extrakci testovacích dat podstatné složky (tj. všechny až na frekvence výskytů), tak s použitím vhodných filtračních nástrojů, např. shell skriptem, vydat informace například o nejčastěji se vyskytujících *-gramech obecně, nebo určitého typu. Jelikož tyto soubory jsou určeny jen pro FAFEFI, není uživateli umožněno jakkoli ovlivnit jejich podobu. Tu ilustruje tabulka 2.2.

Přeje-li si uživatel načtení trénovacích dat z indexového souboru, je toto provedeno namísto extrakce *-gramů z trénovací sady a další výpočet probíhá zcela běžně. Má-li zájem o uložení indexového souboru v rámci výpočtu, je uložen po dokončení extrakce z trénovací sady. Toto nijak neovlivňuje případnou bezprostředně následující extrakci testovacích dat.

374	I went	0	873	91
375	a and	2	1134	76
376	his new bike	1	1	1
377	blue ball	0	3	1

Tabulka 2.2: Ukázka části indexového souboru

2.5 Systém pojmenování výstupu

Výstup je většinou tvořen více soubory, jejichž názvy je třeba šikovně, srozumitelně a jednoznačně odlišit. Toto je ponecháno na uživateli, který bude jistě mít vlastní preference, nicméně jsou mu poskytnuty vhodné nástroje. V rámci specifikace cílového adresáře uživatel zadá i obecný název souborů obsahující speciální řetězce `%FOLD%` a `%TorT%`, z nichž první bude pro každý soubor nahrazen číslem jeho cyklu křížové validace a druhý údajem, zda se jedná o trénovací, či testovací data. Obdobný systém je použit i u názvů indexových souborů.

2.6 Příprava pro cluster

Výpočty jednotlivých cyklů křížové validace jsou na sobě nezávislé a tedy umožňují paralelní spuštění. Je ale nutné, aby bylo možné spustit extrakci pouze právě z jednoho cyklu křížové validace. Jelikož v rámci zpracování všech dat pracujeme v cyklech, stačí, aby ten pro křížovou validaci byl proveden právě jednou se správným validačním cyklem. To je zařízeno jednoduše — extrakční třídě je předán mimo jiné i číselný údaj vyjadřující, který z validačních cyklů nás v tomto výpočtu zajímá, nebo hodnota -1, má-li proběhnout celý výpočet. Na první pohled tedy zbytečně připravujeme adresy vstupních souborů pro celý výpočet, ale používáme jen jejich část. Tato fáze však trvá jen několik vteřin a tedy je to ztráta, ve srovnání se ziskem díky paralelnímu zpracování více cyklů, naprosto zanedbatelná.

Výpočet na clusteru zjednodušuje shell skript, který na základně údajů od uživatele připraví volání spouštějící extrakci z jednotlivých cyklů křížové validace. Uživatel skriptu předá cestu ke konfiguračnímu souboru a číslo n reprezentující celkový počet cyklů a skript pak na základě analýzy konfiguračního souboru vygeneruje n skriptů, z nichž každý bude spouštět FAFEFI s oním konfiguračním souborem pouze pro „svůj“ validační cyklus.

Skript v konfiguračním souboru zkoumá příkazem *grep* přítomnost direktiv pro práci s indexovými soubory a výstupem a dle těchto generuje skripty buď obecné, nebo se slovy train či test v názvu. Spuštění těchto skriptů paralelně na clusteru je dále jen na uživateli.

3. Datové struktury a algoritmy

Zde je podrobně rozebráno několik nejpodstatnějších datových struktur a algoritmů celého programu. Jsou nejprve zkoumány v kontextu jiných možných variant zpracování a dále důkladně popsány co se finální verze týče.

3.1 Strukturalizace vstupních dat

3.1.1 Problém vstupu

Uživatel programu může být z různých důvodů motivován k rozličnému umístění souborů, z nichž mají být extrahovány n-gramy a skip-gramy. Typicky se bude jednat o tyto varianty:

- Jeden adresář obsahující všechny soubory, z nichž mají být data extrahována. Jejich příslušnost cyklům křížové validace může být uložena v názvu souboru.
- Adresářová struktura, kde každý cyklus, trénovací/testovací data a jednotlivé třídy jsou, v libovolném pořadí, různé úrovně (pod)adresářů.
- Kombinace obou předchozích, kde navíc mohou být vstupní soubory uloženy na více místech.

Proto bylo esenciální nalézt vhodnou vnitřní reprezentaci uložení, která je nezávislá na skutečném umístění dat na disku a zároveň umožňuje uživateli dostatečně pohodlně a přehledně umístění dat na disku v konfiguračním programu popsat.

3.1.2 Konfigurace

Definujeme následující dva možné přístupy k uložení dat, které jsou oba relativně snadno převoditelné na identickou datovou strukturu. Pro zbytek výpočtu tedy nebude záležet na tom, jak byla data FAFEFI předána, neboť datová struktura bude nezávislá na způsobu předání.

- Vycházíme z kořenového adresáře a informace o uspořádání jeho podadresářů. Konkrétně nás zajímají tři nejvyšší úrovně, v nichž předpokládáme adresářové odlišení tříd, cyklů křížové validace a trénovacích/testovacích dat. Tyto položky mohou být v libovolném pořadí, ale v každé adresářové úrovni musí být právě adresáře jednoho typu. Není zajištěna žádná kontrola, že struktura odpovídá konfiguraci — vše je v rukou uživatele. Adresáře čtvrté a nižší úrovně od zadaného kořene již nehrají sebemenší roli a jsou zpracovávány jako by jejich obsah byl v jediném adresáři třetí úrovně.

  Tento přístup je vhodný v případě, že uživatel má data pohromadě v adresářích, kde s nimi může snadno manipulovat a příslušnou strukturu vytvořit. Kopírováním podadresářů lze též poměrně snadno vytvořit ze stejných datových souborů vícero sad testovacích a trénovacích dat pro křížovou validaci.

- Druhou variantou je zpracování CSV souboru, kde jsou adresy jednotlivých souborů se vstupními daty. Na každém řádku najdeme cestu k jednomu souboru, přičemž, v pořadí dle parametrů příkazu v konfiguračním souboru, na stejném řádku následuje:
  - název třídy,
  - označení cyklu křížové validace a
  - odlišení testovacích a trénovacích dat.

Přínosem této varianty je možnost mít vstupní data uložena na více místech. Další výhodou je absence nutnosti duplikovat data pro jejich začlenění do různých cyklů křížové validace, neboť při tomto přístupu stačí odkazovat na identické soubory na více řádcích čteného CSV, pouze s odlišnými identifikátory cyklu. Vzorový soubor lze nahlédnout v tabulce 5.1.

3.1.3 Uložení v programu

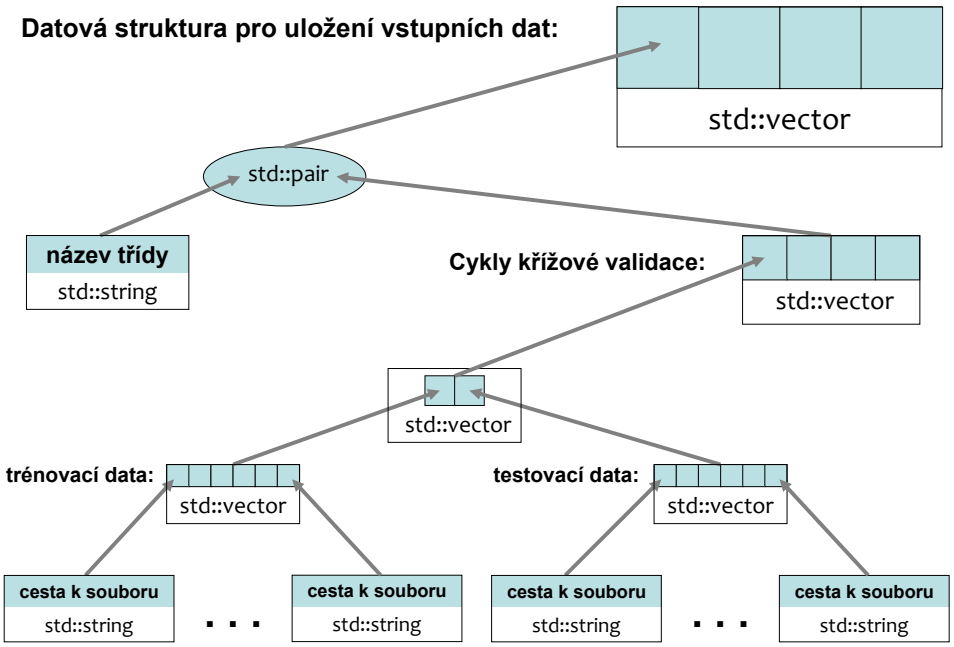
Pro zjednodušení dalšího průběhu výpočtu je nutné, aby datová struktura obsahující cesty k jednotlivým souborům se vstupními daty měla jednotnou podobu při obou variantách zadání těchto cest popsaných výše. Prozkoumáme-li, co mají oba přístupy společného, zjistíme, žejsou to informace o třídě, cyklu křížové validace a testovacím/trénovacím příznaku dat. Pokud tedy datová struktura bude pracovat jen s těmito údaji a nebude záležet na pořadí vkládání dat, bude ji možné použít pro oba konfigurační přístupy.

Ježto předem známe, kolik práce bude s touto strukturou nutné vykonat, můžeme tomu přizpůsobit její návrh. Bude třeba vložit všechna data, tedy strukturu postavit, a pak ke každému souboru právě dvakrát přistoupit — jednou při extrakci n-gramů a skip-gramů a podruhé při konstrukci výstupního vektoru příznaků. Nezáleží tedy příliš na menších ztrátách v rychlosti, pakliže budou patřičně vyváženy.

Vhodným řešením by mohl být strom, jehož kořen by byl k-ární , kde k je počet cyklů křížové validace, vrcholy ve vzdálenosti 1 od kořene l-ární, kde l je počet tříd, vrcholy ve vzdálenosti 2 od kořene binární — reprezentující testovací a trénovací data — a listy stromu pak konkrétní cesty k souborům. Navíc by bylo možné ušetřit místo odkazováním na stejný soubor ukazateli z více cyklů křížové validace. Vzhledem k malé míře využití této datové struktury a možné drobné toleranci vůči rychlostním ztrátám na tomto místě je tvorba takto specializovaného systému, který by pro svou složitost a množství ukazatelů mohl být náchylný k nestabilitě, nevýhodná. Jsme totiž schopni vytvořit velmi podobnou strukturu jen za využití stabilních kontejnerů ze standardní knihovny.

Postupujme zdola. Cesty k souborům jistě chceme reprezentovat řetězci znaků. Soubory příslušné jedné třídě, cyklu křížové validace i trénovací či testovací skupině pak snadno sdružíme do dynamicky alokovaného sekvenčního kontejneru `std::vector`. Dva tyto `std::vectory`, odpovídající cestám k souborům ze sady testovacích, resp. trénovacích dat stejné třídy a cyklu křížové validace, vložíme do dalšího, dvousložkového `std::vectoru`. Jelikož cykly křížové validace bývají typicky číslovány, budou indexy umístění ve `std::vectoru` jednoznačně převoditelné na příslušné cykly, a naopak. K tomuto vnějšímu `std::vectoru` pak přidáme

`std::string` s názvem třídy a vznikne pár reprezentující adresy všech souborů se vstupními daty této třídy. Jednotlivé třídy pak již jen poskládáme do dalšího `std::vectoru`, abychom si zjednodušili následné průchody. Celou datovou strukturu viz na obrázku 3.1.



Obrázek 3.1: Schéma datové strukturty pro uložení cest ke vstupním souborům

3.2 Reprezentace zadaných n-gramů a skip-gramů

3.2.1 Přístupy

Nabízí se dvě základní možnosti:

- rozlišovat n-gramy a skip-gramy hned při konfiguraci a informace ukládat různě, nebo
- ukládat informace o n-gramech a skip-gramech nezávisle na jejich typu.

Zcela zřejmě se oba typy gramů při zpracování liší právě o hodnoty vynechané v rámci skip-gramu. Při první variantě si stačí v poli pamatovat délky jednotlivých n-gramů, ve druhém poli jejich typ a pro skip-gramy vymyslet jinou vhodnou reprezentaci, například přidání dalšího pole s informací o tom, které tokeny je třeba vynechat. Druhou variantou je navržení obecnější struktury schopné pracovat s oběma druhy *-gramů stejně efektivně.

Snadnou úvahou lze předpokládat, že první varianta bude při nepříliš se lišícím počtu gramů obou typů o něco rychlejší, zatímco druhá varianta vede k systematičtějšímu řešení v podobě jednoho bloku kódu pro obecný *-gram. Musíme ale

ještě vzít v úvahu, že gramizovaný řádek je reprezentován polem řetězců a my musíme při získávání nového *-gramu délky n sekvenčně projít prvních n políček tohoto řádku, abychom získali formu *-gramu. Ve výsledku se tedy první možností nic neušetří, ba spíše ztratí potřebou dvojího, hůře optimalizovatelného, kódu. Proto zvolíme variantu s jednou reprezentací pro obecný *-gram nezávisle na druhu.

3.2.2 Reprezentace

Každý n-gram i skip-gram je nezávisle na svém typu reprezentován dvěma údaji. Informací o tom, zdali je binární, poměrný nebo absolutní a polem booleanů velikostí odpovídající požadované délce n-gramu či skip-gramu. Toto pole má pro n-gram samé hodnoty `true`, pro s-gram pak `false` právě na pozicích, které mají být vynechány.

Na první pohled se může zdát, že je taková reprezentace zbytečně složitá, ale jak bylo rozebráno výše, celkovému výsledku to prospívá. Především to směřuje k triviálnímu kódu extrakce, kdy pouhým for-cyklem odpovídajícím rozsahu pole gramu projdeme gram a zároveň indexem tohoto cyklu odkazujeme i do pole tokenů aktuálního řádku. Skrze zřetězení následně snadno zkonstruujeme žádaný n-gram či skip-gram.

3.3 Struktura uložení *-gramů

3.3.1 Analýza

Předně shrňme, co nás u *-gramů zajímá. Potřebujeme znát jeho formu, tj. původní znění slov, ze kterých vznikl, v odpovídajícím pořadí, přičemž je podstatné, i který z typů uživatelem v konfiguračním souboru zadaných *-gramů jej vytvořil: 2-gram "very smart" a 3-skip-2-gram "very smart" vzniklý ze sekvence slov "very very smart" jsou pro účely výpočtu různé příznaky. Dále nás ještě zajímá, ve kterých souborech se vstupními daty se vyskytl a kolikrát. Formu a druh využijeme, budeme-li chtít vědět, zda se již takový n-gram či skip-gram v rámci dané části výpočtu objevil, a tedy zda pouze inkrementujeme počet jeho výskytů, nebo musíme přidat zcela nový údaj. Informace o výskytu v souborech jsou zase esenciální pro vytvoření finálního vektoru příznaků.

Nabízelo by se použít strom, kde by v uzlech byla dvojice formy *-gramu a index jeho typu v rámci `std::vectoru` evidujícím v konfiguračním souboru zadané n-gramy a skip-gramy (dále jen "index typu"). To by sice umožňovalo logaritmickou složitost při hledání existujících a přidávání nových *-gramů, ale pakliže bychom chtěli v takových uzlech uchovávat i informace o výskytech v souborech, potřebovali bychom k jejich uchování ještě přinejmenším seznam či pole.

Pokud ustoupíme od myšlenky stromu a budeme soubory údajů (forma, typ, výskyty) skládat do `std::vectoru`, pořád budou jednotlivé položky tvořeny onou zvláštní dvojicí z uzlu a navíc se dostaneme v nejhorším případě až k lineární složitosti, čemuž bychom se, vzhledem k velkému počtu n-gramů a skip-gramů, který velmi pravděpodobně dostaneme, měli vyhnout. Bude tedy třeba oddělit evidenci různých n-gramů a skip-gramů od databáze jejich výskytů.

Třetí variantou by bylo vytvořit `struct Gram`, která by *-gramy reprezentovala. Snadno by mohla obsahovat `std::string` s formou *-gramu, jeho index typu a slovník převádějící adresy souborů na počet výskytů toho daného n-gramu či skip-gramu v nich, případně i celkový počet výskytů tohoto *-gramu. To se na pohled jeví jako jednoduché a elegantní řešení. Potíž nastává s uložením těchto datových tříd. Potřebujeme v nich totiž během extrakce *-gramů vyhledávat, zda právě extrahovaný *-gram již máme někde uložen, či ne. Seskládat tedy tyto `structy` do vektoru nepřichází v úvahu pro až lineární složitost následného vyhledávání. Vhodné by bylo vytvořit slovník převádějící hodnotu a typ *-gramu na index jeho umístění v onom vektoru. Pak ale položky forma a index typu v rámci datové třídy ztrácejí význam a po jejich odstranění vyvstává otázka, zda bychom výskyty v souborech nedokázali reprezentovat šikovněji než zvláštní třídou.

3.3.2 Realizace

Oddělení slovníku evidujícího nalezené n-gramy a skip-gramy od struktury, v níž jsou uloženy informace o jejich výskytech, nám umožňuje nepoužívat k indexaci páry s formami a typy *-gramů, ale obyčejná čísla. Využijme tedy standardní knihovnu a použijme asociovaný kontejner `map<klic, hodnota>`. Jako klíč použijme již zmíněný pár forma-typ a hodnotou v mapě nechť je unikátní kód toho daného n-gramu či skip-gramu. Ten získáme snadno - bude jím číslo označující o kolikátý nový unikátní *-gram se jedná. Mapa je v rámci standardní knihovny implementována jako binární strom, takže překlad z hodnoty a typu na unikátní kód bude mít logaritmickou časovou složitost.

Jelikož *-gramy máme nyní označeny čísly od 0 do *n-1*, kde n je počet *-gramů, můžeme nadále používat dynamicky alokovaný `std::vector` s konstantním přístupem k prvkům podle indexů jako strukturu pro ukládání dalších informací o *-gramech. Předně se bude jednat o zpětný slovník, který nám převede *-gram dle indexu zpět na jeho formu a typ. To by sice bylo možné realizovat pointery na klíče výše zavedené mapy, ale z důvodu stability, a protože v rámci celkové paměťové náročnosti výpočtu už to hrálo jen malou roli, hodnoty do tohoho vektoru zduplikujeme.

Nyní již k datové struktuře evidující výskyty. Zde se nabízí použít pro každý z *-gramů zvlášť opět mapu, kde klíčem bude adresa zdrojového souboru – ta je jistě unikátní – a hodnotou počet výskytů. Tuto mapu pak umístíme do dalšího `std::vectoru` na příslušný index.

3.4 Algoritmus zpracování *-gramů

Tento proces je velmi přímočarý. Aktuálně otevřený soubor se vstupními daty čteme sekvenčně po řádcích. Stranou máme zjištěno, jak dlouhý je nejdelší požadovaný *-gram – označme toto číslo k. Pracujeme-li s označkovanými daty, prvně z nich vybereme ty části, o které máme zájem, čímž práci s nimi de facto převedeme na práci s daty neoznačkovanými. Pro každý řádek pak postupujeme následovně:

1. Vytvoř pracovní pole k řetězců, kde každý z nich bude obsahovat jen znak začátku věty a vnější oddělovač, tj. oddělovač slov v rámci textu.

2. Na konec aktuálního řádku přidej k znaků konce věty oddělených vnějším oddělovačem.

3. V cyklu, běžícím dokud je v řádku nějaké slovo, postupujme následovně:

   - Z pracovního pole odebereme první `std::string`.
   - Z řádku odebereme vše před prvním oddělovačem a coby nový řetězec to připojíme na konec pracovního pole.
   - Pomocí pracovního pole vytvoříme jednotlivé n-gramy a skip-gramy a ty patřičně zařadíme.

Získání jednotlivých n-gramů a skip-gramů z pracovního pole také není nijak složité. V cyklu přes všechny *-gramy vyžádané uživatelem v rámci konfigurace vezmeme vždy dle délky d aktuálně žádaného n-gramu či skip-gramu příslušně dlouhý cyklus přes prvních d položek pracovního pole. Podmíněným příkazem testovaným proti `true` nebo `false` v definici *-gramu buď přidáme nebo nepřidáme řetězec do výsledného *-gramu. Na konci vnitřního cyklu máme hotový jeden konkrétní n-gram či skip-gram, přičemž index vnějšího cyklu nám udává jakého typu tento n-gram či skip-gram je. Po všech iteracích vnějšího cyklu je celé pracovní pole zpracováno. Jelikož takto se každé slovo může stát prvním slovem v *-gramu právě tolikrát, kolik extrahujeme různých typů *-gramů, je jisté, že nevzniknou žádné duplicity.

I zařazení nalezeného *-gramu do příslušných datových struktur je poměrně snadné. Nejprve je otestováno, zda je *-gram smysluplný – tzn. zda obsahuje něco více než jen znaky začátku nebo konce řádku či jejich kombinaci. Pak je vytvořen pár forma-typ, který je použit pro hledání v mapě-slovníku převádějícím formu a typ *-gramu na jeho unikátní identifikátor. De facto se jedná o průchod stromem. Není-li ve slovníku nový pár nalezen, je do něj přidán, počet různých *-gramů je inkrementován a i ostatní datové struktury pracující s *-gramy (viz výše) jsou příslušně aktualizovány. Následně je do mapy na příslušném indexu vložena informace o tom, že v aktuálním souboru se tento *-gram vyskytl zatím právě jednou. Byl-li pár ve slovníku nalezen, mapa na příslušném indexu je dotázána zda již obsahuje informaci o přítomnosti toho konkrétního *-gramu v aktuálním souboru – jedná se o vyhledání klíče, kterým je adresa tohoto souboru v podobě `std::stringu`. Není-li adresa nalezena, je do mapy přidána a jí příslušná hodnota je 1. Pokud nalezena je, jí příslušná hodnota je o 1 inkrementována.

Při zpracování testovacích dat, kdy již nesmějí vznikat nové *-gramy, je část s přidáváním neznámeho *-gramu vynechána.

3.5 Zpracování vektorů příznaků

3.5.1 Filtrace

Po ukončení extrakce je možné aplikovat filtr eliminující málo frekventované n-gramy a skip-gramy. Ten je veskrze triviální, jelikož se jedná o prostý průchod strukturou evidující výskyty *-gramů v souborech. Podstatné je ale zachování konzistence dalšího výpočtu, zvlášť když pracujeme s trénovacími daty. Odfiltrované *-gramy totiž nelze zcela zahodit, protože vyjmutí prvků ze `std::vectoru`

by znamenalo znehodnocení použití indexů jako jednoznačných identifikátorů. Proto jsou odstraněny pouze záznamy o výskytech a *-gram se chová tak, jako by v žádném souboru nalezen nebyl.

3.5.2 Algoritmus vektorizace příznaků

Jelikož se pro každý soubor vstupních dat nelze vyhnout průchodu přes všechny n-gramy a skip-gramy extrahované ze všech souborů aktuálně řešeného cyklu křížové validace, je tento algoritmus, ač veskrze jednoduchý, časově velmi náročný a bohužel výrazněji nezlepšitelný. Postupem výpočtu z datových struktur odebíráme informace o souborech, které jsme již zpracovali, takže ubývá dat k procházení. Bohužel se kvůli indexování unikátními označeními *-gramů celková šířka datové struktury nemůže zmenšit. Samotný algoritmus pak vypadá následovně:

- Pro každý soubor z aktuálního cyklu křížové validace.
  - Vypiš jeho jméno a příslušnost některé třídě.
  - Pro každý *-gram z extrakce tohoto cyklu křížové validace.
    * V kontejneru projdi data pod jeho indexem.
    * Je-li tam tento soubor zmíněn, dle konfigurovaného typu vypiš příslušnou hodnotu.

4. Formát konfiguračního souboru

V této kapitole, jejímž jediným cílem je přesně popsat podobu konfiguračního souboru, je použita následující notace:

␣	povinná mezera
NUM	číslo
?	jeden, libovolný znak
@X	variabilní položka, podle indexu X dále specifikována
kurzíva	nepovinná část

Příkazy pro konfigurační soubor:

- `Length␣NUM␣@1␣`*`Skip␣@2`*
  - @1 ⇒ právě jedno z klíčových slov `Absolute`, `Binary` nebo `Ratio`
  - @2 ⇒ čísla oddělená čárkami; bez mezer
- `DataRoot␣"@1"␣@2-@3-@4`
  - @1 ⇒ cesta k adresáři
  - @2,@3,@4 ⇒ parametry `Fold`, `Class` a `TorT`, každý právě jednou, v libovolném pořadí
- `File␣"@1"␣@2-@3-@4`
  - @1 ⇒ cesta k souboru
  - @2,@3,@4 ⇒ parametry `Fold`, `Class` a `TorT`, každý právě jednou, v libovolném pořadí
- `Inner␣Separator␣"?"␣Index␣NUM`
- `Outer␣Separator␣"?"`
- `LineFilter␣"@1"`
  - @1 ⇒ regulární výraz
- `LineModifier␣"@1"␣"@2"␣`*`Recursive`*
  - @1 ⇒ co nahrazujeme
  - @2 ⇒ čím nahrazujeme
- `SetMin␣NUM␣@1`
  - @1 ⇒ právě jedno z klíčových slov `DocCount`, nebo `Absolute`
- `In␣IndexFile␣"@1"`
  - @1 ⇒ cesta k souboru, která musí obsahovat vzor %FOLD%

- `Out␣IndexFile␣"@1"`
  - @1 ⇒ cesta k souboru, která musí obsahovat vzor %FOLD%
- `Out␣FeatureVector␣"@1"`
  - @1 ⇒ cesta k souboru, která musí obsahovat vzory %FOLD% a %TorT%
- `Out␣Separator␣"?"`
- `#@1`
  - @1 ⇒ zde může být cokoli

5. Uživatelská dokumentace

Tato kapitola seznamuje uživatele s principy ovládání programu. Vysvětluje, jak program spustit, popisuje, jak fungují a k čemu lze použít jednotlivé příkazy v konfiguračním souboru, rozebírá techniku využití indexových souborů a v závěru objasňuje význam chybových hlášek.

5.1 Spuštění

FAFEFI lze použít třemi způsoby:

- jako spustitelný program,
- knihovnu pro extrakci n-gramů a skip-gramů, nebo
- jej spustit na clusteru.

5.1.1 Spustitelný program

FAFEFI při spuštění vyžaduje právě jeden parametr, který udává adresu konfiguračního souboru — ta musí být v uvozovkách. Bez něj nemůže výpočet proběhnout. Uživatel je v průběhu konfigurace upozorňován na řádky, jejichž význam nebyl pro chyby v syntaxi rozpoznám. Na konci konfiguračního procesu je tázán, zda i přes tyto chyby chce zahájit extrakci. Spuštění může vypadat například takto:

```
./fafefi "./config.txt"
```

5.1.2 Knihovna

Možnost využít FAFEFI v rámci jiného programu je plně podporována. Pro takové užití je třeba do tohoto programu přidat hlavičkové soubory `extraction.h` a `configuration.h`. Konfigurace se spouští voláním metody `start(string)` třídy `Congfiguration`, která jako parametr bere adresu konfiguračního souboru — ten je pro korektní běh bezpodmínečně nutný. Samotná extrakce, která není spustitelná bez předchozí konfigurace, se volá metodou `start(configuration, int)` třídy `Extraction`. Ta jako parametr přijímá objekt třídy `Configuration` a číslo -1. Obě spouštěcí metody vrací hodnotu 0, pokud v jejich průběhu dojde k chybě, která si nevynucuje pád celého programu.

Tato varianta se nedoporučuje uživateli bez alespoň základní znalosti programování.

5.1.3 Cluster

Spuštění na clusteru je částečně na uživateli, FAFEFI pouze poskytuje podporu takové varianty. Pomocí připraveného nástroje[1] lze vygenerovat spouštěcí skripty

[1] Skript cluster.sh z přiloženého CD

pro jednotlivé cykly křížové validace a ty potom spustit na clusteru. Při odděleném výpočtu jednotlivých cyklů není uživatel tázán na pokračování výpočtu po dokončení konfigurace. Generovací skript používá následující přepínače:

- `--folds`
  Za přepínačem musí následnovat přirozené číslo vyjadřující počet cyklů křížové validace, pro který mají být generovány spouštěcí skripty.

- `--file`
  Tento přepínač následuje cesta ke konfiguračnímu souboru, která je předána generátoru spouštěcích skriptů. Na základě příkazů v tomto konfiguračním souboru se budou lišit názvy vygenerovaných souborů dle toho, zda budou sloužit trénovacím datům, testovacím datům nebo celému výpočtu.

- `-o, --outputDir`
  Umožňuje uživatel specifikovat adresář, kam budou spouštěcí skripty uloženy.

- `-h, --help`
  Vypíše help popisující přepínače a jejich použití.

Žádný z přepínačů není povinný, ale bez `--folds` a `--file` nedá skript žádný relevantní výstup. Korektní spuštění z terminálu mohou vypadat například takto:

- `./cluster.sh --folds 7 --file ./config.txt`

  Budou vygenerovány spouštěcí skripty pro 7 cyklů křížové validace, z nichž každý bude jako konfiguraci brát soubor config.txt z aktuálního adresáře.

- `./cluster.sh --file ./config.txt --folds 12 -o ./cluster/`

  V tomto případě, který ilustruje, že na pořadí přepínačů nezáleží, budou vygenerovány spouštěcí skripty pro 12 cyklů křížové validace. Každý pak bude jako konfiguraci brát soubor `config.txt`. Všechny vytvořené skripty se uloží do adresáře `./cluster/`.

5.2 Konfigurace

Konfigurační příkazy jsou založeny na jednoduchých klíčových slovech a pevné syntaxi; tato ale musí být bezpodmínečně dodržena, jinak je řádek označen za chybný a není uvažován, případně může zabránit úspěšnému dokončení výpočtu. Uživatel je na takovou chybu upozorněn a tázán, zda chce i přes chybu ve čtení onoho řádku pokračovat ve výpočtu. Na každém řádku smí být nejvýše jeden příkaz, ale mezi jednotlivými příkazy mohou být vynechané řádky. U některých příkazů může záležet na jejich pořadí v rámci konfiguračního souboru, u většiny však nikoli. Konfigurační příkaz je hierarchický - část, která není povinná, nikdy nepředchází povinné.

5.2.1 N-gramy a skip-gramy

`Length␣NUM␣@1␣`*`Skip␣@2`*

- @1 ⇒ právě jedno z klíčových slov `Absolute`, `Binary` nebo `Ratio`
- @2 ⇒ čísla oddělená čárkami; bez mezer

Při konfiguraci je skip-gram uvažován jako speciální typ n-gramu. N-gram je snadno definován pouze na základě své délky, u skip-gramu je definice rozšířena ještě o indexy slov k vynechání. Indexování probíhá od 1 dál, tedy první token v rámci řetězce má index 1. Pokud uživatel indexuje mimo rozsah skip-gramu, je tento index ignorován.

Nedílnou součástí definice n-gramu i skip-gramu je jeho typ v závěrečném výstupu. Rozlišujeme tři: binární, poměrný a absolutní. V případě binárního (`Binary`) je výstupem 0 nebo 1 sdělující, zda se ten konkrétní n-gram či skip-gram v souboru vyskytl, nebo ne. Poměrný (`Ratio`) výstup je poměrem počtu těchto *-gramů vůči všem *-gramům stejného typu. Jelikož by byl vždy hodnotou mezi 0 a 1, je zaokrouhlen na 4 desetinná místa a přenásoben 1000, tedy vystupuje jako celé číslo. Absolutní (`Absolute`) typ je prostým počtem výskytů daného n-gramu či skip-gramu v aktuálním vstupním souboru. Příklady:

- `Length 2 Binary`
  Zajímá nás, které dvojice slov (délka 2) se v souborech vyskytují (binární vyhodnocení).
- `Length 4 Absolute`
  Ve výstupu budou všechny různé existující čtveřice, hodnotou jim přiřazenou bude počet výskytů této čtveřice v daném souboru.
- `Length 5 Ratio Skip 2,4`
  Hledáme všechny pětice, ze kterých vynecháváme tokeny na sudých pozicích. Reprezentujeme je na celé číslo zaokrouhleným tisícinásobkem poměru jejich počtu ku počtu všech skip-gramů tohoto typu.

5.2.2 Vstupní data

`DataRoot␣"@1"␣@2-@3-@4`

- @1 ⇒ cesta k adresáři
- @2,@3,@4 ⇒ parametry `Fold`, `Class` a `TorT`, každý právě jednou, v libovolném pořadí

`File␣"@1"␣@2-@3-@4`

- @1 ⇒ cesta k souboru
- @2,@3,@4 ⇒ parametry `Fold`, `Class` a `TorT`, každý právě jednou, v libovolném pořadí

Uživateli je umožněno volit ze dvou možností nastavení informace o zdrojích vstupních dat. Příkazy `DataRoot` a `File`, jež jsou oba následovány cestou, v případě `DataRoot` adresáře a v případě `File` CSV souboru, jak názvy napovídají, se navzájem vylučují. Jejich vzájemnou nekompatibilitu zajišťuje program — po korektním zpracování libovolného z nich žádné další zpracování neumožní. Taktéž není možné použít stejný příkaz dvakrát — po prvním úspěšném zpracování vstupních dat je jakákoli další manipulace s jejich nastavením zablokována.

Program nijak nekontroluje korektnost dodaných cest, ani to, zda struktura adresářů odpovídá popisu. Je-li v adresářích nějaký soubor, který místo vstupních dat určených ke zpracování obsahuje něco odlišného, může způsobit znehodnocení výpočtu nebo dokonce zabránit jeho dokončení[2]. Špatný popis struktury s velkou pravděpodobností způsobí špatný výsledek výpočtu. V závislosti na konkrétní chybě buď výpočet spadne, nebo je na pohled korektně dokončen, ovšem se špatnými daty. Protože omezování uživatele v názvech tříd a podadresářů je spíše překážkou než bezpečnostním prvkem, je právě na něm, aby k těmto chybám nedocházelo. Obsah všech podadresářů ve vzdálenosti od kořene větší než 3 je interpretován, jako by byl přímo v adresáři ve vzdálenosti 3 od kořene. Je naprosto nutné, aby adresáře odpovídající trénovacím resp. testovacím sadám byly nazvány „train“ resp. „test“ a adresáře cyklů křížové validace byly označeny číslem. Jinak nelze garantovat plnou funkčnost programu.

Ve vstupu z CSV souboru taktéž není kontrolována příslušnost třídám, cyklům křížové validace ani testovacím či trénovacím datům. Názvy adresářů a souborů ale mohou být libovolné, neboť příslušnost třídám, cyklům a trénovací nebo testovací sadě jsou programu předány dále na řádku. Oddělovače nejsou pro tento soubor konfigurovatelné — očekává se oddělování novými řádky, přičemž jednotlivé položky na jednom řádku musí být separovány čárkou. CSV soubor může vypadat například takto:

```
./data/1074.txt,ClassA,07,train
./data/1074.txt,ClassB,07,train
./data/1521.txt,ClassA,07,test
./data/3002.txt,ClassA,07,test
./data/2377.txt,ClassB,05,train
```

Tabulka 5.1: CSV soubor se zadáním cest ke vstupním datům

- `DataRoot "./data/fafefi/" Class-Fold-TorT`
  Vstupní soubory jsou v podadresářích adresáře fafefi. Přímo v něm jsou adresáře odpovídající jednotlivým třídám, v nich adresáře odpovídající cyklům křížové validace a v nich adresáře reprezentující testovací a trénovací data.

- `File "./MyData/datalist.csv" Fold-TorT-Class`
  CSV soubor s cestami k souborům se vstupními daty nalezneme v adresáři MyData.

[2]Záleží na konkrétním obsahu a konfiguraci. Stejná sekvence znaků může pro jednu konfiguraci vést k několika nesmyslným *-gramům a pro jinou k věčnému cyklu.

5.2.3 Oddělovače

`Inner␣Separator␣"?"␣Index␣NUM`

`Outer␣Separator␣"?"`

V rámci vstupních dat může jako oddělovač tokenů sloužit nejen mezera, ale též jakýkoli jiný, právě jeden, znak (např. i tabulátor). FAFEFI lze na toto upozornit příkazem `Outer Separator` následovaným oním znakem. Pak je toto nastavení použito ve všech souborech, ze kterých jsou vstupní data čtena. Pakliže není v konfiguračním souboru tento příkaz zadán, je použita implicitní hodnota tohoto oddělovače, kterou je mezera.

Může být potřebné pracovat s anotovanými daty. K tomu slouží příkaz `Inner Separator`, který určuje oddělovač položek v rámci anotovaného tokenu. Je také implicitně nastaven na mezeru. Jeho nutnou součástí je i určení indexu požadovaného segmentu odděleného tímto oddělovačem — bez ní je příkaz považován za neplatný. Je-li hodnota indexu, počítaného od 1, větší, než počet segmentů v tokenu, program nebude korektně pracovat. Bohužel není v době konfigurace možné zjistit, zda se uživatel nespletl a není účelné hádat, co asi měl na mysli, pokud se spletl.

Je-li v konfiguračním souboru více korektně zadaných příkazů pro nastavení oddělovačů, výsledná konfigurace bude odpovídat poslednímu z nich.

- `Outer Separator " "`
  Oddělovačem tokenů v souborech se vstupními daty je mezera.
- `Inner Separator ";" Index 3`
  V rámci jednoho tokenu jsou položky odděleny středníkem; textová hodnota tokenu je obsažena ve třetí položce dané tímto oddělovačem.

5.2.4 Filtr řádku

`LineFilter␣"@1"`

- @1 ⇒ regulární výraz

Pro filtraci řádků, ze kterých mají být extrahovány n-gramy a skip-gramy, slouží příkaz `LineFilter`. Ten umožňuje uživateli definovat regulární výraz v rozšířené (Extended) syntaxi, proti kterému se přečtený řádek musí namečovat, aby z něj byly extrahovány *-gramy. Nestane-li se tak, není tento řádek do výpočtu zahrnut. Tento příkaz je možné v konfiguračním souboru opakovat, potom se každý řádek bude muset namečovat na každý z takto definovaných regulárních výrazů, jinak nebude zpracován.

Pokud by měl uživatel zájem o opačný přístup, tj. určit regulárním výrazem, které řádky mají být při namečování vyloučeny, tak stačí mezi příkazové slovo `LineFilter` a zadaný výraz vložit slovo `Not`. Potom je význam příkazu invertován a odpovídající řádky jsou z výpočtu vyloučeny. I tuto variantu lze v rámci konfigurace opakovat. Vyloučen je každý řádek, který se namečuje alespoň na jeden z těchto regulárních výrazů a i zde se pracuje s rozšířenou syntaxí. Je samozřejmě též možné obě varianty příkazu v rámci jedné konfigurace kombinovat.

Podobu rozšířené sady regulárních výrazů v rámci knihovny POSIX lze nastudovat na `http://en.wikipedia.org/wiki/Regular_expression`.

- `LineFilter ".*[a-zA-Z]+( )+[a-zA-Z]+( )+[a-zA-Z]+.*"`
  Uživatel má zájem pouze o řádky, kde jsou nejméně 3 slova.

- `LineFilter Not ".*[0-9].*"`
  Pokud řádek obsahuje číslovku, nebude zpracován.

5.2.5 Modifikátor řádku

`LineModifier␣"@1"␣"@2"␣`*`Recursive`*

- @1 ⇒ co nahrazujeme

- @2 ⇒ čím nahrazujeme

Mnohdy může být užitečné před extrakcí *-gramů upravit řádek, na kterém je prováděna. Kupříkladu nás jako příznak příliš nezajímá, kolikrát a kde se v souborech vyskytlo číslo 5 či 9, ale spíše fakt, že se někde vyskytlo nějaké číslo. Modifikátor řádku nám dává možnost toto číslo nahradit nějakým uživatelem zvoleným obecným označením čísla, díky čemuž 2-gramy „5 jablek“ a „6 jablek“ budou uvažovány jako stejné, což pro mnoho úloh také jsou.

Druhé, snad ještě důležitější, využití je při práci s anotovanými daty. V těch může často být řádek uvozen či ukončen nějakým označením, které souvisí s anotací a nikoli s obsahem textu. Takové označení samozřejmě do extrakce zahrnou nechceme a právě modifikátorem řádku jej lze odstranit.

Protože většinou nestojíme o nahrazení pouze prvního výskytu, nýbrž všech, je možné příkaz rozšířit slovem `Recursive`, které zajistí rekurzivní zpracování modifikace řádku. Ten je tak modifikován, dokud je možné v něm nalézt alespoň jeden řetězec k nahrazení.

Pro konkrétní představu o tom, co přesně tento příkaz umí, lze konzultovat libovolný přehled standardní knihovny C++, například [5], neboť metoda používá právě knihovní funkce `find()` a `replace()`. Na řádek jsou nejprve aplikovány nerekurzivní úpravy v pořadí, jak byly zadány v konfiguračním souboru a až potom ty rekurzivní, ve stejném pořadí.

- `LineModifier "VALUE: " ""`
  Uživatel chce smazat označení řádku „VALUE“.

- `LineModifier " thee " " you "`
  Nahrazení knižních výrazů současnými. Oddělení mezerou garantuje, že první slovo nebude nahrazeno, je-li podřetězcem jiného slova.

- `LineModifier "00" "0"`
  Zmenšování příliš velkých čísel.

5.2.6 Filtr výskytů

`SetMin␣NUM␣@1`

- @1 ⇒ právě jedno z klíčových slov `DocCount`, nebo `Absolute`

Pro redukci počtu *-gramů ve výstupním vektoru příznaků slouží příkaz `SetMin`. Jelikož se řada n-gramů i skip-gramů vyskytuje v souborech jen řídce a tedy má pro další výpočet jen malý význam, je ku prospěchu věci tyto *-gramy vypustit. Učiníme tak určením minimálního počtu jejich výskytů, a to buď v součtu v rámci jednoho cyklu křížové validace (tj. varianta `Absolute`), nebo stanovením nejmenšího povoleného počtu souborů se vstupními daty v rámci jednoho cyklu křížové validace, kde se onen *-gram musí vyskytnout alespoň jednou (parametrem `DocCount`).

Je-li stanovená podmínka porušena byť o 1, je tento n-gram či skip-gram z celého výpočtu vyloučen. Je-li v rámci konfigurace příkaz `SetMin` zopakován, program bere v potaz jen poslední korektní zadání a předcházející ignoruje. Implicitní hodnotou filtru počtu výskytů je 0 (tedy se bez vědomí uživatele nekoná žádná filtrace tohoto druhu) a implicitním nastavením parametru je varianta `Absolute`. Na tu dojde v případě, že uživatel použije příkaz `SetMin` bez typového parametru.

- `SetMin 17 Absolute`
  Každný n-gram či skip-gram, který se celkem nevyskytl alespoň sedmnáctkrát nebude zařazen do výstupního vektoru příznaků.

- `SetMin 17`
  Má naprosto stejný význam jako příkaz v příkladu výše.

- `SetMin 5 DocCount`
  Pokud se n-gram či skip-gram nevyskytl alespoň v 5 souborech, bude na výstupu ignorován.

- `SetMin DocCount`
  Nelze!

5.2.7 Indexový soubor

`In␣IndexFile␣"@1"`

- @1 ⇒ cesta k souboru, která musí obsahovat vzor %FOLD%

`Out␣IndexFile␣"@1"`

- @1 ⇒ cesta k souboru, která musí obsahovat vzor %FOLD%

Tento pár příkazů, jeden vstupní a jeden výstupní, umožňují odložit extrahovaná trénovací data stranou a později je opět načíst za účelem extrakce dat testovacích. Je třeba vzorem specifikovat cykly křížové validace, pro které se místo čerstvé extrakce mají načíst již extrahovaná data z indexových souborů. Bude-li v konfiguračním souboru více takových definic, bude použita právě ta poslední korektně zadaná. Tento typ výpočtu nelze kombinovat s novou extrakcí z trénovací sady.

Příkaz ve verzi `In` musí být až za příkazy `File` a `DataRoot`, pomocí nichž chceme načítat testovací data! Je nutné, aby uživatel zajistil, že cykly budou číslovány právě od 1 do k, a to bez vynechání kteréhokoli čísla mezi 1 a k, jinak nelze zaručit funkčnost tohoto příkazu.

Ve výstupní verzi budou uloženy indexové soubory pro všechny cykly křížové validace aktuálního výpočtu. Stane-li se, že uživatel o některé z nich nemá zájem, je na něm je odstranit.

- `In IndexFile "./fafefi/idxfile_%FOLD%.csv"`
  Načtení indexových souborů z adresáře dle vzoru.
- `Out IndexFile "./fafefi/idxfile_%FOLD%.csv"`
  Uložení indexových souborů do adresáře dle vzoru.

5.2.8 Výstup

`Out␣FeatureVector␣"@1"`

- @1 ⇒ cesta k souboru, která musí obsahovat vzory %FOLD% a %TorT%

`Out␣Separator␣"?"`

Specifikace příkazem `Out` nijak neovlivňuje výpočet, ale pouze stanovuje konkrétní podobu výstupních vektorů příznaků. `Separator` určuje oddělovač sloupců reprezentujících *-gramy ve výstupním souboru. Používá znak čárka coby implicitní hodnotu. `FeatureVector` udává adresář, kam budou data na výstupu uložena, i specifika jejich názvů. Je nutné, aby tento adresář již existoval a byl pro program přístupný k zápisu. Zajištění těchto podmínek je ponecháno uživateli. V názvu souboru mohou být vzory %FOLD% a %TorT%, přičemž první bude pro každý soubor nahrazen číslem jeho cyklu a druhý informací, zda se jedná o trénovací či testovací data. Zbytek názvu výstupních souborů je zcela na vůli uživatele, stejně jako zajištění dostupnosti a možnosti zapisovat do těchto souborů. Pokud se uživatel rozhodne nekonfigurovat umístění a název výstupních souborů, budou uloženy do adresáře ./ pod názved *k_ train.csv* a *k_ test.csv*, kde *k* je číslo cyklu křížové validace a train/test odpovídají trénovacím, resp. testovacím datům.

- `Out FeatureVector "./fafefi/out_%FOLD%_%TorT%.csv"`
  Do tohoto adresáře budou uloženy všechny soubory s výstupními daty a budou se jmenovat dle vzoru.
- `Out Separator ";"`
  Oddělovačem sloupců s jednotlivými *-gramy bude středník.

5.2.9 Komentář

`#@1`

- @1 ⇒ zde může být cokoli

Uživateli je umožněno do konfiguračního souboru vkládat i komentáře. Učiní tak příkazovým znakem # na začátku řádku, jehož celý obsah pak bude zakomentován. Komentáře v rámci řádku, a to ani od konkrétního místa do konce

řádku, nejsou možné. Předpokládá se totiž, že užívání komentářů bude sloužit buď k dočasnému neutralizování některého z příkazů, či ke krátkému komentáři příkazu, který je možné napsat i na řádek výše nebo níže.

- `#Outer Separator ","`
  Nadále není zájem o to, aby čárka byla vnějším oddělovačem.

- `# definice všech n-gramů`
  Pokus o zpřehlednění konfiguračního souboru sobě i ostatním čtenářům.

5.2.10 Zajímavé možnosti konfigurace

- Při nahrazení čísel obecným znakem pro číslo bohužel nelze zadat nic na způsob „Najdi všechna čísla a nahraď je zvláštním znakem“, jelikož lze hledat a nahrazovat jen konkrétní řetězce a nikoli zobecnění, jako třeba „čísla“ či „velká písmena“. Je tedy nutné mít 10 příkazů nahrazujících konkrétní číslovku obecným číslovkový znakem zvoleným uživatelem. Dostaneme řetězce těchto zvláštních znaků, na které použijeme rekurzivní variantu redukce jejich počtu (tj. nahrazení dvou stejných znaků za jeden). Pro znak # na místo libovolného čísla by taková konfigurace mohla vypadat třeba takto:
  - `LineModifier "0" "#"`
  - `LineModifier "1" "#"`
  - ... a tak dále až do ...
  - `LineModifier "9" "#"`
  - `LineModifier "##" "#" Recursive`

- Pro zpracování označkovaných dat, kde, kupříkladu, je relevantní řádek vždy uvozen slovem „VALUE:“ a v rámci jednotlivých anotovaných slov nás zajímá vždy druhá položka oddělená středníkem, by v konfiguračním souboru mohlo být napsáno toto:
  - `LineFilter "^VALUE: "` ⇒ vybrání pouze relevantních řádků
  - `LineModifier "VALUE: " ""` ⇒ odstranění pro extrakci irelevantních značek
  - `Inner Separator ";" Index 2` ⇒ určení oddělovače a indexu položky

5.3 Indexové soubory

FAFEFI umožňuje zpracovat trénovací data a uložit je na disk k pozdějšímu využití v podobě tzv. *indexových souborů*. Příklad indexového souboru lze nalézt v tabulce 2.2. Jsou užitečné v případě, že uživatel nemá testovací data k dispozici v době extrakce trénovacích dat, nebo pokud chce extrahované položky trénovací sady uložit k případnému pozdějšímu použití. Jak používat příkazy konfiguračního souboru sloužící k práci s indexovými soubory je popsáno výše, nyní se podívejme na praktické využití těchto souborů a na to, jak se k němu oněmi příkazy dobrat.

Prvním použitím může být prostý zájem o to, které *-gramy se ve vstupních datech vyskytly a kolikrát. Při této variantě není v plánu s indexovými soubory dále pracovat a budou sloužit jen ke čtení. Pak stačí přidat jeden `Out IndexFile` příkaz do konfiguračního souboru, kde je i příkaz `Out FeatureVector`, který nám zajistí standardní výstup extrahovaných příznaků.

Na tuto variantu využití indexových souborů přímo navazuje další, kdy nadále chceme výstup příznaků, ale plánujeme v budoucích extrakcích testovacích dat použít získaný indexový soubor. Konfigurace v tomto případě zůstává stejná a liší se jen použití.

V případě, že nemáme testovací data k dispozici, či z nějakého důvodu chceme výstup jen trénovacích dat, vynecháme příkaz `Out FeatureVector` a výstupem bude pouze indexový soubor.

Nyní k využití indexových souborů na místo extrakce příznaků z trénovacích dat. To řešíme příkazem `In IndexFile`. Při takovéto práci s indexovými soubory je bezpodmínečně nutné, aby byl FAFEFI předán zcela identický konfigurační soubor jako při tvorbě oněch indexových souborů! Jinak totiž není možné zajistit správnost výsledku.

Indexové soubory nenesou plnou informaci o extrakci, ze které vznikly. To znamená, že je nelze v rámci dalšího výpočtu rozšiřovat novými trénovacími daty. Jelikož extrakční fáze zabere oproti sestavení vektorů příznaků jen zlomek času, nelze toto považovat za překážku.

5.4 Chybové hlášky

Program během výpočtu může narazit na nějakou překážku, která proces více či méně narušuje. V takovém případě na terminál (konkrétně `std::cerr`) napíše, o jaký problém se jedná. Následuje výpis hlášek i s vysvětlením a návrhem, jak je řešit.

5.4.1 Řídící třída

Pokud jsou třídy Configuration a Extraction použity jako knihovny, tato sekce je irelevantní.

- *Missing an argument* ⇒ Programu chybí jeho jediný povinný argument, tj. adresa konfiguračního souboru.
- *Due to an error during configuration the program will now close* ⇒ Během konfigurace došlo k chybě, ze které se nelze zotavit a bez jejíhož vyřešení není možné, aby program doběhl.
- *Invalid character, please try again* ⇒ Uživatel byl tázán, aby zadal buď y/Y coby kladnou, nebo n/N coby zápornou odpověď. Bylo ale zadáno něco jiného a tedy je třeba zkusit zadat odpověď znovu.
- *Due to an error during extraction the program will now close* ⇒ V extrakci nastala nezotavitelná chyba. Program bude ukončen, všechny doposud zpracované cykly křížové validace budou uloženy v cílovém adresáři, ale nic z doposud netknutých nebo jen rozpracovaných nebude dokončeno ani dáno na výstup.

5.4.2 Konfigurační třída

- *Could not compile regex* ⇒ Některý z regulárních výrazů používaných k parsování řádků konfiguračního souboru nebylo možné zpracovat a tedy je nutné program ukončit, jelikož by nebylo možné vytvořit funkční a správnou konfiguraci.

- *Unable to open the configuration file* ⇒ Cesta ke konfiguračnímu souboru byla zadána ve špatném formátu. Typicky se může jednat o špatná lomítka, fakt, že se nejedná o soubor, či opomenutí uvedení argumentu do uvozovek. Program vypisuje konkrétní typ vyvolané výjimky.

- *Line n: wrong format --> index is too low* ⇒ N-tý řádek má špatný formát, jelikož index pro vnitřní oddělovač je menší nebo roven nule. Tento řádek nebude v rámci konfigurace použit.

- *Already using different form of data access* ⇒ Metoda přístupu k souborům se vstupními daty a jejich poloha byly již definovány jiným způsobem. Jedná se jen o varování, že byl příkaz ignorován, a je na zvážení uživatele, zda chce po dokončení konfigurace pokračovat ve výpočtu s prvním validním nastavením zdroje dat.

- *Could not compile user defined regex* ⇒ Regulární výraz definovaný uživatelem se nepodařilo zpracovat. Byl pravděpodobně špatně definován, tj. například porušoval syntaxi rozšířené sady regulárních výrazů. Jelikož nelze očekávat, že uživatel stojí o výpočet bez filtru, který vědomě, ale špatně, zadal, je výpočet ukončen.

- *Wrong format* ⇒ V rámci konfigurace některého z výstupních parametrů nastala chyba. Další průběh programu to neovlivní, pouze tento chybný příkaz bude ignorován. Je na uživateli zvážit, zda je to problém, a program po konfiguraci případně ukončit.

- *Line n: wrong format* ⇒ Na n-tém řádku v konfiguračním souboru nastala chyba při parsování, neboť tento řádek neodpovídal žádnému z parsovacích regulárních výrazů. S největší pravděpodobností byla na tomto řádku porušena předepsaná syntaxe konfiguračního souboru.

- *Content structure specification error* ⇒ Struktura podadresářů v rámci příkazu DataRoot byla zadána špatně. Protože v konfiguračním souboru má být uveden právě jeden zdroj dat, jehož zpracování právě selhalo, nemá smysl v konfiguraci pokračovat a program je tedy ukončen.

- *Reading CSV file error* ⇒ Při zpracovávání vstupních dat z CSV souboru došlo k chybě. Specifika vyvolané výjimky program vypsal na další řádek.

- *The given address was NOT a directory* ⇒ V rámci zpracování příkazu `DataRoot` došlo k chybě při čtení adresářů.

- *Reading directory error* ⇒ Během průchodu obsahu adresáře při zpracování příkazu `DataRoot` došlo k přístupu na neplatnou adresu. O konkrétním významu vyvolané výjimky je uživatel informován na dalším řádku.

- *Input file address error* ⇒ Zařazení datového souboru do struktury určené ke zpracování během extrakce selhalo. Chyba je s největší pravděpodobností na straně programu, nikoli uživatele, což lze ověřit na následujícím řádku výstupu.
- *Missing a valid *-gram definition* ⇒ V rámci celého konfiguračního souboru nebyl korektně zadán ani jeden n-gram či skip-gram a tedy nemá smysl ve výpočtu pokračovat.
- *Missing a valid input data definition* ⇒ Uživatel nespecifikoval zdroj dat a program tedy nemá odkud extrahovat příznaky.
- *Wrong format of n-gram or skip-gram definition* ⇒ Při zpracování zadání n-gramu či skip-gramu došlo k chybě. Tento *-gram nebude zařazen do zpracování a je na uživateli zhodnotit, zda i bez něj chce s výpočtem pokračovat. Konkrétní výjimka je vypsána na následujícím řádku.

5.4.3 Extrakční třída

- *Extraction fault* ⇒ Během extrakce došlo k nezotavitelné chybě.
- *Saving index file error* ⇒ Během ukládáná indexového souboru na disk došlo k chybě. Konkrétnější informace o vyvolané výjimce následují chybovou hlášku.
- *Loading index file error* ⇒ Při načítání dat z indexového souboru nastala chyba. V závislosti na chybě může bý výpočet ukončen. Podrobnosti jsou na následujícím řádku.
- *Features vector creation error* ⇒ Při zpracovávání extrahovaných dat do vektoru příznaků nastal problém. Lze očekávat nekonzistenci a neúplnost výstupních dat. Specifika jsou vypsána na dalším řádku.

6. Alternativy k FAFEFI

FAFEFI není jediným programem k extrakci n-gramů. Existují další, z nichž každý oproti němu poskytuje trochu jiné možnosti a výhody i nevýhody. V této kapitole se s nimi postupně seznámíme, dozvíme se, co umí a jak zhruba fungují. Na konci kapitoly tyto poznatky shrneme a v několika hlavních kategoriích porovnáme, jak si proti nim FAFEFI stojí.

6.1 scikit-learn

Scikit-learn je velmi komplexní knihovna pro strojové učení. Je naprogramována v Pythonu a extrakce příznaků tak, jak ji provádí FAFEFI, je jen její malou částí. Je jednoduchým a efektivním nástrojem pro data mining a analýzu dat. Je všeobecně dostupná jako open source pod licencí BSD a široce využitelná — umožňuje předpřípravu dat a je schopna řešit různé typy úloh strojového učení.

Nás ale pro srovnání zajímají především její extrakční schopnosti. Jelikož pro ukládání příznaků používá seznamy standardních pythonovských *dict* objektů, které jsou velmi pohodlné pro práci, zato ale nepříliš rychlé, je ve srovnání s FAFEFI pomalá. Navíc model, kterým knihovna pracuje s extrahovanými n-gramy (práci se skip-gramy neumožňuje) je nevhodný pro zachování vnitřní struktury textu či libovolného kontextu v něm.

Program nalezneme zde: `http://scikit-learn.org/stable/index.html`.

6.2 N-Gram Extraction Tools

Tento open source program distribuovaný pod licencí MIT je sadou nástrojů pro práci s n-gramy z neznačkovaných dat. Používá algoritmus extrakce n-gramů popsaný v [6] a zvládá zpracování jak slovních n-gramů (např. čeština) tak znakových n-gramů (např. japonština) až do rozsahu $n=255$. Pracuje s textem v kódování Unicode (UCS-2), což umožňuje snazší práci s jazyky jako je čínština či korejština. V těchto jazycích slova nemají explicitní hranici a mnohdy mohou být zpracovávány jednotlivé znaky.

Zajímavou součástí jsou algoritmy statistické redukce podřetězců. Jedná se o proces odstranění redundantních n-gramů pomocí statistických údajů. Vyskytnou-li se v textu například řetězce „People's Republic of China“ a „Peoples's Republic“, každý desetkrát, druhý z nich bude odstraněn, jelikože se jedná o podřetězec prvního. Detaily o tomto procesu je možné nalézt v článku [7].

Oproti FAFEFI tato sada umožňuje extrakci pouze z jednoho souboru a nabízí jen základní možnost filtrace — na základě nastavení minimálního počtu výskytů n-gramu v souboru. Není jím možné extrahovat skip-gramy.

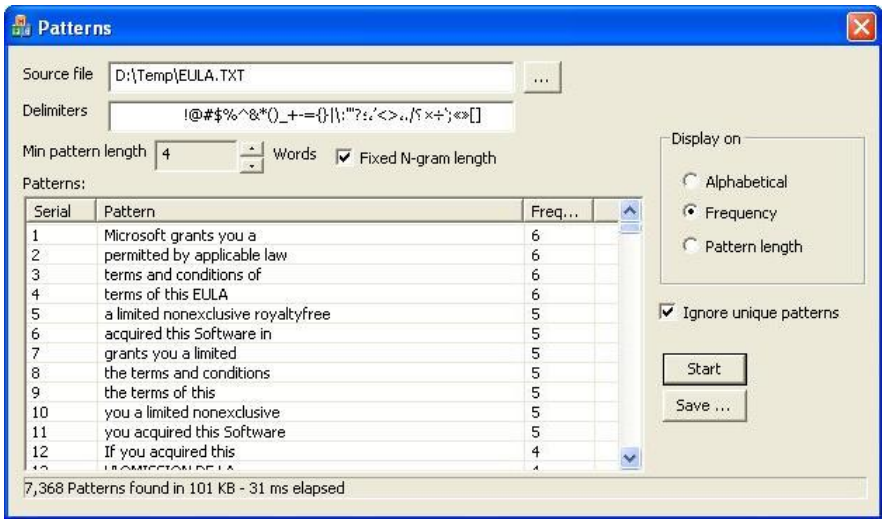
Program je pro stažení dostupný na adrese
`http://homepages.inf.ed.ac.uk/lzhang10/ngram.html`.

6.3 N-gram and Fast Pattern Extraction Algorithm

Jedná o program řešící úlohu extrakce vzorů, v níž je extrakce n-gramů vedlejším efektem. Vzor může mít fixní, podobně jako n-gram, nebo proměnnou délku. Vzory proměnné délky mohou být direktivami k určitým pravidlům jako například regulární výrazy. Také mohou být náhodné a záviset na kontextu a opakování vzorů ve slovníku vzorů.

Stejně jako FAFEFI je tento program napsán v C++. Jeho algoritmus je úpravou algoritmu Lempel-Ziv-Welch 84 určeného k bezeztrátové kompresi dat. Díky těmto faktům je extrakce pomocí tohoto programu rychlá, na druhou stranu program neposkytuje žádné nadstavby pro strojové učení ani neumožňuje formátovat výstup rovnou do vektoru příznaků.

Program je možné nalézt na `http://www.codeproject.com/Articles/20423/`.



Obrázek 6.1: Příklad ukončeného výpočtu

6.4 N-gram — Tool for n-gram extraction from xml files

N-gram je nástroj k extrakci n-gramů ze souborů ve formátu XML. Je napsán v jazyku C# s použitím .Net Framework 1.2. Umožňuje především výběr uzlů pro konstrukci n-gramů a určení konečných uzlů (např. k vyloučení interpunkčních znamének) pomocí XPath[1] výrazu a filtraci výstupních dat pomocí XSL[2] stylesheet. Navíc je možné přidat k n-gramům ID a HASH atributy či rozdělit výstupní soubor na více částí. Má poměrně přívětivé uživatelské rozhraní, jak lze nahlédnout na obrázku 6.2.

[1]Pomocí XPath lze adresovat části XML dokumentu, vybírat z něj jednotlivé elementy a pracovat s jejich hodnotami a atributy.

[2]Extensible Stylesheet Language; rodina jazyků umožňující popsat, jak se mají XML soubory formátovat a převádět.

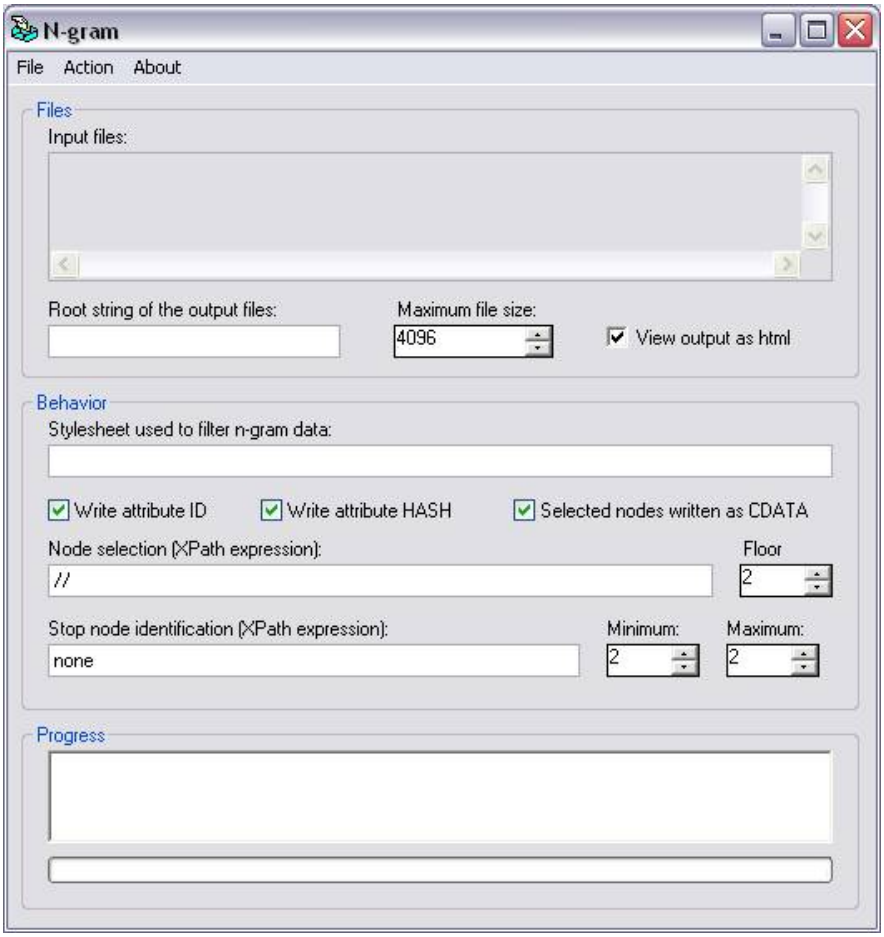
Algoritmus:

1. Po jednom naparsuje vstupní XML soubory a vybere uzly, o něž měl uživatel zájem dle XPath, který nastavil.

2. Rozdělí sérii uzlů na konečných uzlech určených XPath. Tato procedura je užitečná především, když interpunkční znaménka nejsou potřebná, ale reprezentují důležité informace pro optimalizaci konstrukce n-gramů.

3. Část dokumentu sestávající z řady vybraných uzlů je filtrována pomocí XSL stylesheet. Výsledky jsou přidány do hešovací tabulky.

4. Položky v hešovací tabulce, které přesahují předepsané minimum jsou zapsány do XML souboru dané maximální velikosti. Každý soubor má kořenovou položku nazvanou „N-GRAM“, která je unikátním identifikátorem daného souboru, a položku „COUNT“ k ukládání absolutního počtu n-gramů daného typu. Atribut COUNT není ovlivněn zadaným minimem výskytů. Každý n-gram může mít ID unikátní přes celý výstup programu a může být zapsán jako CDATA[3] kvůli urychlení parsování výstupních souborů.



Obrázek 6.2: Vzhled konfiguračního okna programu

[3]CDATA — běžné znaky odpovídající zhruba typu `char` například z jazyka C++.

Oproti FAFEFI přináší tento nástroj širší možnosti využití XML souborů, na jejichž zpracování je primárně zaměřen. Pro běžná data je ale jen velmi těžko použitelný. Také nedává žádnou možnost zpracování extrahovaých n-gramů do vektorů příznaků, což vede k nutnosti mít nějaký další program řešící tuto úlohu.

Program nalezneme zde: `http://n-gram.sourceforge.net/`.

6.5 Shrnutí

FAFEFI není jediným nástrojem pro extrakci n-gramů. Jak ale můžeme nahlédnout v tabulce 6.1, jelikož zvládá extrahovat nejen n-gramy, nýbrž i skip-gramy, poskytuje nemálo nadstavbových funkcí, dokáže extrahovaná data zpracovat do vektorů příznaků a lze jej spustit na clusteru, nemá přímou konkurenci. Ostatní nástroje jsou buď zaměřené jen na specifické výpočty, napsány v interpretovaném jazyce nebo velmi omezené co do rozšiřujících funkcí.

Program	Jazyk	n-gramy	skip-gramy	filtr	cluster
scikit-learn	Python	ano	ne	ano	ano
N-Gram Extraction Tools	C++	ano	ne	ano	ne
N-g and Fast Pattern Ext. Alg.	C++	ano	ne	ne	ne
N-gram Tool for ext. from XML	C#	ano	ne	ano	ne
FAFEFI	C++	ano	ano	ano	ano

Tabulka 6.1: Porovnání programů

Závěr

FAFEFI je nástroj sloužící k extrakci n-gramů a skip-gramů z velkého množství jazykových dat, který je široce, snadno a přehledně uživatelsky konfigurovatelný a podporuje spouštění na clusteru. Má řadu rozšiřujících funkcí. Zvládá různé formy předání vstupních dat v rozličných způsobech uložení, umožňuje filtraci a modifikaci řádků, dává uživateli šanci zredukovat množství příznaků ve výstupních vektorech a dokáže oddělit zpracování trénovacích a testovacích sad.

Jsa implementován v jazyce C++, využívá jen součásti standardní knihovny a knihovny jazyka C POSIX definované před standardem C++11, takže je spustitelný na širokém spektru UNIXových strojů. V jeho datových strukturách jsou použity pouze kombinace kontejnerů ze standardní knihovny, což znamená drobné rezervy v časové náročnosti operací, zato ale garantovanou spolehlivost. Tyto struktury jsou voleny tak, aby se s nimi pohodlně pracovalo, a zároveň neplýtvaly operační pamětí, jejíž úspora je jedním z klíčových problémů extrakce z velkého množství dat.

Algoritimické řešení samotné extrakce n-gramů a skip-gramů i jejich následného zpracování do vektorů příznaků je přímočaré. Důležitými součástmi programu jsou hlavně rozšiřující funkce, jejichž rozsah jej v rámci implementací v kompilovaných programovacích jazycích kvalitativně staví nad konkurenci, která se zpravidla zabývá jen pouhou extrakcí n-gramů a základním filtrováním.

FAFEFI umožňuje uživateli pozitivní (řádek musí projít) i negativní (řádek nesmí projít) filtraci vstupu stejně jako redukci příznaků na výstupu na základě počtu jejich výskytů v jednotlivých souborech nebo v celém výpočtu. Díky modifikátoru řádku a vnitřnímu oddělovači v rámci tokenů umožňuje i práci s označkovanými daty či zobecnění některých tokenů v rámci řádků — kupříkladu převedení konkrétních čísel na obecné číslo oproštěné jakékoli hodnoty.

Další podstatnou podporovanou vlastností je oddělení různých fází výpočtu. Předně je možné zcela rozdělit a tedy i paralelizovat výpočty různých cyklů křížové validace. V rámci této paralelizace je součástí práce i shell skript[4], který při předání konfigurace takového výpočtu a celkového počtu foldů vygeneruje jednotlivá volání, která pak uživatel spustí například na clusteru.

Také je možné místně i časově rozdělit fáze zpracování trénovacích a testovacích dat. Datové struktury obsahující údaje o dokončené extrakci trénovacích dat lze uložit na disk v podobě indexových souborů a ty pak ve chvíli zájmu o extrakci testovacích dat opět načíst. Tuto variantu lze použít i při bezprostředně následující extrakci k tomu, aby uživatel mohl prozkoumat obsah a počty výskytů jednotlivých n-gramů a skip-gramů, které by se jinak na výstup nedostaly.

Při konfiguraci sázíme na jednoduchost a jednoznačnost. V rámci pevně a striktně definované syntaxe konfiguračního souboru se FAFEFI u nejasných či (jakkoli lehce) syntaxi neodpovídajících příkazů nesnaží hádat, co měl uživatel pravděpodobně na mysli. Pracuje s filosofií, že je lepší po dvou sekundách běhu skončit s chybou, než provést celý (mnohdy několikahodinový) výpočet a pak nechat uživatele zkoumat, proč dostal zcela odlišný výsledek, než jaký očekával.

Lze nalézt části programu, které by bylo možné zlepšit. Kupříkladu by bylo možné rozšířit uživatelské možnosti, především co se přívětivosti a doplňkových

[4]Naleznete jej na přiloženém CD v adresáři ./cluster_tools.

funkcí týče. Například by šlo generovat definovatelné statistiky z výpočtu. Tako by bylo možné ukládat konfigurace, informace o počtu souborů, jejich průměrné velikosti a na nich proběhlých výpočtech a na základě těchto údajů pak uživatele informovat, jak dlouho bude pravděpodobně nový výpočet trvat. Též by bylo možné více využívat nízkoúrovňových možností, které jazyk C++ programátorovi poskytuje, a na jejich základě zlepšit výkonnost. Tato rozšíření však již přesahují rozsah práce.

FAFEFI řeší extrakci *-gramů z jazykových dat a jejich okamžité zpracování do vektorů příznaků. K tomu přidává široké spektrum jednoduše uživatelsky konfigurovatelných položek, nezanedbatelné množství nadstavbových funkcí a možnost paralelizace výpočtu na clusteru. Touto kombinací zaujímá mezi konkurencí unikátní místo, takže jistě nalezne řadu uživatelů, pro něž bude optimální volbou.

Seznam použité literatury

[1] Hladká, Barbora, Holub, Martin, Kríž, Vincent. *Feature Engineering in the NLI Shared Task 2013: Charles University Submission Report.* Proceedings of the Eighth Workshop on Innovative Use of NLP for Building Educational Applications, pages 232–241, Atlanta, Georgia, June 13 2013. http://www.aclweb.org/anthology/W13-1730

[2] Indurkhya, Nitin, Damerau, Fred J. (eds). *Handbook of Natural Language Processing.* 2. vydání Chapman and Hall/CRC, 2010. ISBN-10: 1420085921. ISBN-13: 978-1420085921.

[3] Alpaydin, Ethem. *Introduction to Machine Learning.* 2. vydání. Massachusetts: The MIT Press, 2010. ISBN 978-0-262-01243-0.

[4] Eckel, Bruce. *Thinking in C++.* 2. vydání. Bruce Eckel, MindView, Inc., 1999. http://mindview.net/Books/TICPP/ThinkingInCPP2e.html

[5] Josuttis, Nicolai M. *C++ Standardní knihovna a STL.* 1. vydání. CP Books, a.s., 2005. ISBN 80-251-0511-1.

[6] Nagao, Leslie, Shinsuke, Mori. *A New Method of N-gram Statistics for Large Number of n and Automatic Extraction of Words and Phrases from Large Text Data of Japanese.* The 15th International Conference on Computational Linguistics (COLING 1994). http://acl.ldc.upenn.edu/C/C94/C94-1101.pdf

[7] Lv, Xueqiang, Zhang, Le, Hu, Junfeng. *Statistical Substring Reduction in Linear Time* IJCNLP-04, Hai Nan island, P.R.China. http://homepages.inf.ed.ac.uk/lzhang10/paper/linearssr.pdf

Seznam tabulek

Seznam obrázků

Seznam použitých zkratek

- **FAFEFI** — Fast Feature Extraction and Filteringů název programu popsaném v této práci
- ***-gram** — Libovolný typ gramu, v rámci této práce vždy buď n-gram nebo skip-gram
- **CSV** — Comma-separated values; souborový formát sestávající z řádků, ve kterých jsou jednotlivé položky odděleny znakem „ , “
- **RAM** — Random Access Memory; paměť s přímým přístupem
- **XML** — Extensible Markup Language; obecný značkovací jazyk, definuje formát dokumentů
- **XSL** — Extensible Stylesheet Language; rodina jazyků umožňující popsat, jak se mají XML soubory formátovat a převádět

Přílohy

Veškerá data v elektronické podobě jsou k práci přiložena na CD. Adresářová struktura a obsah CD vypadá takto:

- `/cluster_tools`
  - `/cluster.sh` — shell skript sloužící ke generování příkazů pro spuštění na clusteru
- `/documentation` — automaticky generovaná programátorská dokumentace
- `/examples`
  - `/fafefi` — zkompilovaný program FAFEFI
  - `/config.txt` — vzor konfiguračního souboru
  - `/config_tag.txt` — vzor konfig. souboru pro označkovaná data
- `/makefile`
  - `/faf_make` — makefile pro kompilaci FAFEFI
- `/src` — zdrojový kód FAFEFI
  - `main.cpp`
  - `configuration.cpp`
  - `configuration.h`
  - `extraction.cpp`
  - `extraction.h`
- `/text`
  - `/bakalarska_prace.pdf` — text práce
- `/README.txt` — informační soubor o obsahu CD