VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

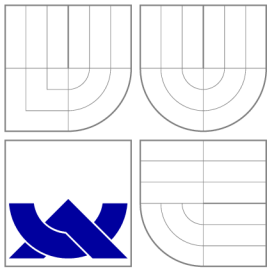
SNÍŽENÍ NÁROČNOSTI VÝPOČTŮ V LIBSVM S POUŽITÍM ŘETĚZCOVÝCH FUNKCÍ

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

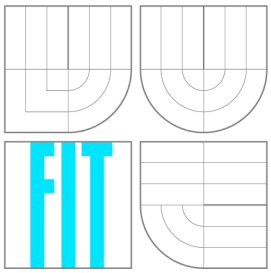
AUTOR PRÁCE
AUTHOR

TOMÁŠ KUBERNÁT

BRNO 2010



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

SNÍŽENÍ NÁROČNOSTI VÝPOČTŮ V LIBSVM S POUŽITÍM ŘETĚZCOVÝCH FUNKCÍ

REDUCTION OF COMPUTATION COST IN LIBSVM USING STRING KERNEL FUNCTIONS

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

TOMÁŠ KUBERNÁT

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. ZBYNĚK MICHLOVSKÝ

BRNO 2010

Abstrakt

Cílem práce bylo implementovat čtyři řetězcové funkce do knihovny *libSVM*. Za pomoci této knihovny a výše zmíněných řetězcových funkcí poté provést sérii testování s různými hodnotami parametrů ovlivňujících výpočet samotných řetězcových funkcí. Pomocí experimentů byla porovnána rychlost a úspěšnost klasifikace mojí implementace řetězcových funkcí v knihovně *libSVM* s implementací řetězcových funkcí v programu *kernels*. V práci jsou také popsány průběhy všech testování i s naměřenými hodnotami a grafy pro grafické znázornění výsledků.

Abstract

The goal of this thesis was to implement four string functions into the library *libSVM*. Then apply series of tests with variable parameters values affecting the individual string functions using the library and string functions. Using the results of experiments the speed and success of clasification of my implementation of string functions in library *libSVM* was compared with the implementation of string functions in program *kernels*. In this thesis there are also described procedures of all tests along with measured data and their graphical representation.

Klíčová slova

strojové učení, klasifikace, support vector machine, kernel trick, teorie jader, řetězcové funkce, vícetřídní klasifikace, lineární klasifikátor, knihovna libSVM

Keywords

machine learning, classification, support vector machine, kernel trick, kernel theory, string kernels, multi-class classification, linear classifier, libSVM library

Citace

Tomáš Kubernát: Snížení náročnosti výpočtů v libSVM s použitím řetězcových funkcí, bakalářská práce, Brno, FIT VUT v Brně, 2010

Snížení náročnosti výpočtů v libSVM s použitím řetězcových funkcí

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Zbyňka Michlovského

.....
Tomáš Kubernát
18. května 2010

Poděkování

Děkuji vedoucímu své bakalářské práce, panu Ing. Zbyňku Michlovskému, za cenné rady, připomínky a celkové vedení bakalářské práce.

Obsah

Kapitola 1

Úvod

Bez internetu a síťové komunikace si dnešní život dokáže představit jen málokdo. Slouží ke sdělování a vyhledávání informací, komunikaci, seznamování a je prakticky spojen se vším, čím se dnes člověk zabývá. Je pro člověka každodenním prostředkem komunikace, práce a také zdrojem zábavy.

Globální komunikace by bez internetu byla ochuzena o významný komunikační kanál, díky kterému se prakticky zkrátily vzdálenosti napříč celou zeměkoulí. Se zvyšující se datovou komunikací po internetu hrozí stále větší nebezpečí útoků na tuto komunikaci. Útočníci se snaží napadnout koncové stanice uživatelů, různé aplikační servery nebo aktivní síťové prvky a zařízení zajišťující provoz sítě.

Zaznamenáváním datového pohybu na těchto zařízeních získáme data, která obsahují informace o potenciálních útocích. Klasifikací se potom pokoušíme tato data roztřídit podle toho, jestli jsou škodlivá nebo neškodná. Nesnažíme se klasifikovat jednotlivé fragmenty síťové komunikace, ale už celé zprávy, jimiž mohou být logy síťových provozů, emailové zprávy či řetězcová data. Tato práce je zaměřena na klasifikaci řetězcových dat, která slouží pro detekci obsahu kolujícího po internetu.

V kapitole 2 jsou vysvětleny teoretické pojmy a výrazy úzce související s danou problematikou. Jsou teoreticky objasněny postupy klasifikace, jednotlivé způsoby provádění klasifikace a blíže popsány programy *kernels* a *libSVM* [4]. Čtenář se obeznámí s řetězcovými funkcemi, které byly implementovány do knihovny *libSVM*. Dále je zde popsána úprava dat a formáty vstupních a výstupních souborů, ve kterých jsou tato data uložena.

Kapitola 3 podrobněji popisuje způsoby provádění klasifikace. Jsou v ní podrobně interpretovány postupy pro získání výsledků, a to jak procentuálních, tak i časových. Uvedeno je i testovací prostředí a faktory ovlivňující zejména časové výsledky našich pokusů.

Detailní popis jednotlivých fází testování je následně uveden v kapitole 4. Pro každou řetězcovou funkci jsou přesně popsány jednotlivé kroky, které vedly k výsledkům. Tyto hodnoty jsou potom pro každou metodu zobrazeny v tabulkách nebo grafech. Je zde uvedeno použití skriptů, které zprostředkovávaly všechny běhy klasifikací a počítaly veškeré výsledky.

Poslední kapitola je věnována komplexnímu porovnání naměřených výsledků.

Kapitola 2

Teoretická východiska

2.1 Strojové učení

Strojové učení(*machine learning*) je podoblastí umělé inteligence, zabývající se algoritmy a technikami, které umožňují počítačovému systému učit se. Zde je učení myšleno jako změna vnitřního stavu systému, která zefektivní schopnost přizpůsobení se změnám okolního prostředí [14]. Strojové učení je dovednost inteligentního systému měnit svoje znalosti tak, že příště bude vykonávat stejný nebo podobný úkol efektivněji [15].

Vytváří inteligentnější rozhodnutí založené na přijatých datech. Tato data bývají v rámci učení obsahem trénovací množiny dat. Po natrénovaní našeho systému musíme ověřit, jestli tento systém máme dobře zkonstruovaný a jestli dává uspokojivé výsledky. Tento proces se nazývá *testování* a děje se na testovacích datech, která ještě klasifikátor neviděl. Na testovací množině si ověříme správnost a úspěšnost našeho učícího algoritmu.

Díky učení by se měla zvyšovat výkonnost našeho systému. Strojové učení lze rozdělit do čtyř základních kategorií [14]:

- Učení s učitelem (*Supervised learning*)
- Učení bez učitele (*Unsupervised learning*)
- Posilované učení (*Reinforcement learning*)
- Kombinace učení s učitelem a bez učitele (*Semi-supervised learning*)

Učení s učitelem

Prvním principem strojového učení je učení s učitelem, tzv. *supervised learning* [15]. Strojové učení s učitelem spočívá v tom, že pro každý vstupní prvek, který přichází do učícího algoritmu, je předem znám očekávaný výstup. Díky tomuto přístupu je učící se systém ihned informován o tom, jak dobře se právě přijatý vzorek dat dokázal naučit. Do této kategorie učení patří následující metody učení s učitelem:

- Rozhodovací stromy *decision trees*
- Prohledávání prostoru verzí *version-space search*
- Rozpoznávání obrazů *pattern recognition*

Učení bez učitele

Učení bez učitele probíhá stejně jako učení s učitelem, ale s tím rozdílem, že vstupní data nejsou nijak ohodnocená. Učící algoritmus rozděluje data podle společných rysů. Většinou se jedná o kritérium kvality, které může být reprezentováno např. vzdáleností bodů v prostoru. Používá se několik typů vzdáleností a za zmínku stojí *Euklidovská vzdálenost*, *Cosinova vzdálenost* či *Manhattan distance*. Tomuto způsobu učení se také říká *shlukování* nebo anglicky *clustering* [2].

Posilované učení

Posilované nebo též motivované učení je další z principů strojového učení. Systém provádí určité akce, o kterých si myslí, že povedou ke správnému výsledku. Ke každé akci, kterou provede, dostane náležité ohodnocení. Většinou se jedná o číselné ohodnocení akce a jde buď o kladnou nebo zápornou odměnu, podle toho, jestli aktuální krok vedl k lepšímu nebo horšímu výsledku. Cílem systému je maximalizovat výši ohodnocení v čase, po který učení probíhá [15].

Posilované učení je typ strojového učení a zároveň také větví umělé inteligence. Toto učení umožňuje strojům a softwarovým agentům automaticky určit ideální chování uvnitř specifického kontextu maximalizováním svého výkonu.

Kombinace učení s učitelem a bez učitele

Kombinace učení s učitelem a bez učitele je další z technik strojového učení. Tato metoda používá ohodnocená i neohodnocená data pro trénování. Většinou se jedná o malé množství ohodnocených dat a velké množství dat neohodnocených. Mnoho výzkumníků v oblasti strojového učení zjistilo, že jsou-li neohodnocená data použita ve spojení s malým počtem dat ohodnocených, vytvářejí významné zlepšení v úspěšnosti učení. Získání ohodnocených dat pro problém učení často vyžaduje zásah lidského pracovníka s určitými znalostmi a schopnostmi, který manuálně klasifikuje trénovací příklady. Cena spojená s procesem ohodnocení tedy může učinit plně ohodnocený trénovací dataset neuskutečnitelným, zatímco získání neohodnocených dat je relativně levné. V takové situaci může mít tato technika učení velké praktické využití [12].

2.2 Klasifikace

Dalším výrazem pro strojové učení s učitelem je *klasifikace* [2]. Pod pojmem klasifikace se nerozumí jenom samotný proces třídění vstupních dat. Jedná se o sofistikovaný systém několika různých procesů, které na sebe navazují. Základní klasifikační systém se skládá z částí, kterými jsou získávání dat, extrakce příznaků a samotné klasifikace. Prvky rozšířené klasifikace jsou vstup, snímání, segmentace, extrakce příznaků, klasifikace, post-processing a rozhodnutí. Klasifikace se využívá napříč veškerou lidskou činností, ale strojová klasifikace např. v průmyslu nebo medicíně.

2.3 Klasifikátor

Je to algoritmus, který dostane ukázky, jak řešený problém vypadá. Respektive jak vypadají jednotlivé třídy, které se snažíme rozlišit. Klasifikátor se snaží naučit, jak příchozí data

rozdělit do jednotlivých tříd. Fázi učení také jinak nazýváme *trénování* a v této fázi se snažíme najít rozhodovací linii, která co nejlépe od sebe oddělí prvky dvou odlišných tříd. Rozhodovací linie musí být zvolená tak, aby klasifikátor dobře generalizoval na datech, která ještě neviděl [2].

Druhou fází je fáze testování, která se provádí na odlišných datech než trénování. Testovací data jsou taková data, která ještě nebyla klasifikátorem viděna a slouží k ověření toho, že náš klasifikátor na vstupních datech, která budou přicházet v ostrém provozu, bude dobře fungovat. Po procesu testování klasifikátoru musíme dosažené výsledky analyzovat, a pokud jsou dobré, tak můžeme klasifikátor nasadit do ostrého provozu.

2.4 Generalizace

Jde o schopnost klasifikátoru správně zpracovat data, která nebyla použita v procesu učení [6]. Když máme špatně určenou rozhodovací linii, tak nám sice může dobře rozdělovat trénovací data, ale nová testovací data klasifikovaná dobře nebudou. Špatně zvolená rozhodovací linie nám prakticky znemožní nebo drasticky sníží generalizaci. Když klasifikátor dobře generalizuje, tak správně rozpoznává to, co ještě neviděl.

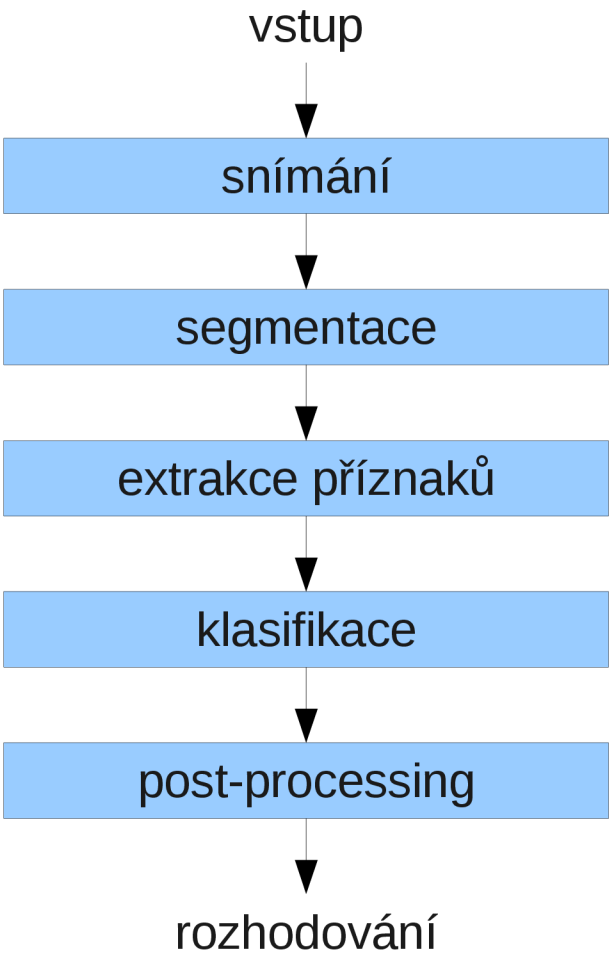
2.5 Přetrénování

Přetrénování nebo také přeučení či *over-fitting*. Při příliš velkém množství trénovacích dat klesá chyba na trénovacích, ale roste chyba na testovacích datech [2]. Přetrénování může být také způsobeno příliš přesnými příznaky, které byly získány ze vstupních dat v rámci trénování. Systém potom velmi dobře nebo dokonale rozpozná již jednou viděná data, ale hůře klasifikuje data, která ještě nikdy neviděl.

2.6 Vlastní práce klasifikátoru

Činnost klasifikátoru není jenom o tom, nějakým způsobem klasifikovat data, ale na celkovém průběhu klasifikace se podílejí ještě další procesy mimo samotné klasifikace [2]. Musíme data, které budou klasifikována nějak získat a upravit tak, aby mohla být zpracována samotným klasifikátorem. Postup klasifikace je znázorněn na obrázku 2.1 a jednotlivé etapy celého procesu jsou :

- vstup dat (*input*)
- snímání (*sensing*)
- segmentaci (*segmentation*)
- extrakci příznaků (*feature extraction*)
- klasifikaci (*classification*)
- post-processing (*post-processing*)
- rozhodování (*decision*)



Obrázek 2.1: Znázornění práce klasifikátoru

Vstup

Vstupem do klasifikátoru jsou data, která chceme roztřídit do nějakých tříd, pokud se jedná o klasifikaci nebo do shluků, pokud se jedná o shlukování. Vstupní data můžeme rozdělit na trénovací, testovací a *ostrá data*.

Trénovací data slouží k učení klasifikátoru. Na těchto datech klasifikátor vidí, jak by asi mohla vypadat data, která dostane v rámci testování a nebo už v ostrém provozu. Klasifikátor si zapamatuje význačné vlastnosti těchto dat a tyto informace potom využívá k rozřazování ještě neviděných dat. Tato data mohou být testovací nebo data přicházející do klasifikátoru v ostrém provozu.

Snímání

Snímání slouží pro vlastní získání dat. Data, která dále pokračují do dalších částí klasifikátoru musí být nějak získána. Způsob, kterým data získáváme je závislý na tom, která data budeme klasifikovat. Když chceme klasifikovat obrazová data, tak snímacím zařízením

bude kamera nebo fotoaparát. Pro záznam zvukových dat použijeme mikrofon. Existuje ještě mnoho speciálních detektorů všemožných veličin pro různá použití. V našem případě budeme snímat řetězcová data z datasetu *Reuters*, a tak jako snímač nám poslouží jednoduchý textový editor pro úpravu a zobrazení řetězcových dat.

Segmentace

Segmentace nám z celkových dat vyřeže jenom takové části, které potřebujeme a které jsou důležité. Ostatní části dat jsou pro náš účel zbytečné a mohly by např. zpomalit klasifikaci nebo zkreslit její výsledky. Proces segmentace může být složen z více klasifikátorů, které hledají relevantní informaci např. obličej v obrázku, odstraní tiché pasáže z audio nahrávky nebo vstupní řetězec transformují na malá písmena.

Extrakce příznaků

Když chceme klasifikovat obrázky nebo zvuk, tak potřebujeme z jednotlivých souborů dostat takové informace, díky kterým budeme schopni udělat vlastní klasifikaci. Z velkého objemu dat se snažíme vybrat co nejmenší počet parametrů, které pořád budou dobře popisovat to, co naše data reprezentují. Jde vlastně o redukci vlastností daného problému. Příznaky nám umožňují rozlišovat mezi jednotlivými třídami. Tento proces je předzpracování vstupů do samotného klasifikátoru.

Klasifikace

V tomto procesu se snažíme vstupní data zařadit do určité třídy nebo je přiřadit do určitého shluku podobných dat. Při klasifikaci máme třídy určeny ve fázi trénování klasifikátoru a v testování nebo ostrém provozu už třídy musí klasifikátor určit sám. Ve shlukování právě příchozí prvek můžeme přiřadit právě k tomu shluku, se kterým má aktuální prvek nejvíce sousedů nebo ke kterému středu, některé ze skupin, má menší vzdálenost. Metod pro určení příslušnosti ke shluku je ovšem daleko více.

Post-processing

Post-processing je místo, kde můžeme dolaďovat výsledky klasifikace v rámci kontextu využitím dalších informací. Je to konečná úprava výsledků, které přicházejí z procesu klasifikace. I když máme data klasifikovaná a rozdělená do skupin, tak jejich zařazení ještě nemusí být naprosto finální. V tomto procesu můžeme dát našim výsledkům pravou váhu, podle které se nakonec budeme rozhodovat. Většinou se používá nějaké ohodnocení (*cost*) toho, co nám přišlo z klasifikace. Například může jít o pravděpodobnostní vyjádření.

Rozhodování

V tomto procesu již napevno rozhodneme, kam daný vzorek zařadíme a co s ním následně budeme provádět. Kvůli rozhodnutí děláme všechny předešlé sekce.

My se ovšem budeme zajímáme především o proces klasifikace, protože všechny předcházející procesy jsou již vyřešeny.

2.7 Lineární klasifikátor

Hlavním úkolem jakéhokoliv klasifikátoru je roztřídit data do určitých skupin. Lineární klasifikátor klasifikuje data do dvou tříd a odděluje je pomocí lineárního oddělovače např. přímka, rovina nebo nadrovina. Vzorec lineárního klasifikátoru je:

$$y(x) = w^T x + w_0, \tag{2.1}$$

kde w je prostor, ve kterém se vyskytují data, x jsou data, která chceme klasifikovat a w_0 je práh, podle kterého rozhodneme, do jaké třídy aktuální prvek patří.

Používá se také zobecněný lineární klasifikátor, který má vzoreček:

$$y(x) = f(w^T x + w_0) \tag{2.2}$$

kde f je tzv. *aktivační funkce*. Aktivační funkce mapuje čísla tříd na výstupní hodnoty klasifikátoru podle následujícího vzorečku.

$$f(x) = \begin{cases} +1 & x \geq 0 \\ -1 & x < 0 \end{cases} \tag{2.3}$$

Pokud jsou data, která klasifikujeme, lineárně oddělitelná, tak lineární klasifikátor vždy řešení nalezne. Jedná se o iterační algoritmus a záleží na prvotním nastavení rozhodovací linie, jak dlouho a kolikrát bude algoritmus iterovat a jakou separační linii klasifikátor nalezne. V opačném případě, kdy data není možné lineárně separovat, bude algoritmus iterovat pořád dokola a nikdy nezkonverguje. Při nalezení první separační linie, která odděluje data jedné třídy od druhé, se algoritmus lineárního klasifikátoru zastaví. Řešení je sice použitelné a odděluje data obou tříd, ale nemusí být ani zdaleka optimální.

Při vstupu dalších dat do klasifikátoru už nemusí dříve určená linie data oddělovat a klasifikátor se musí přeučit.

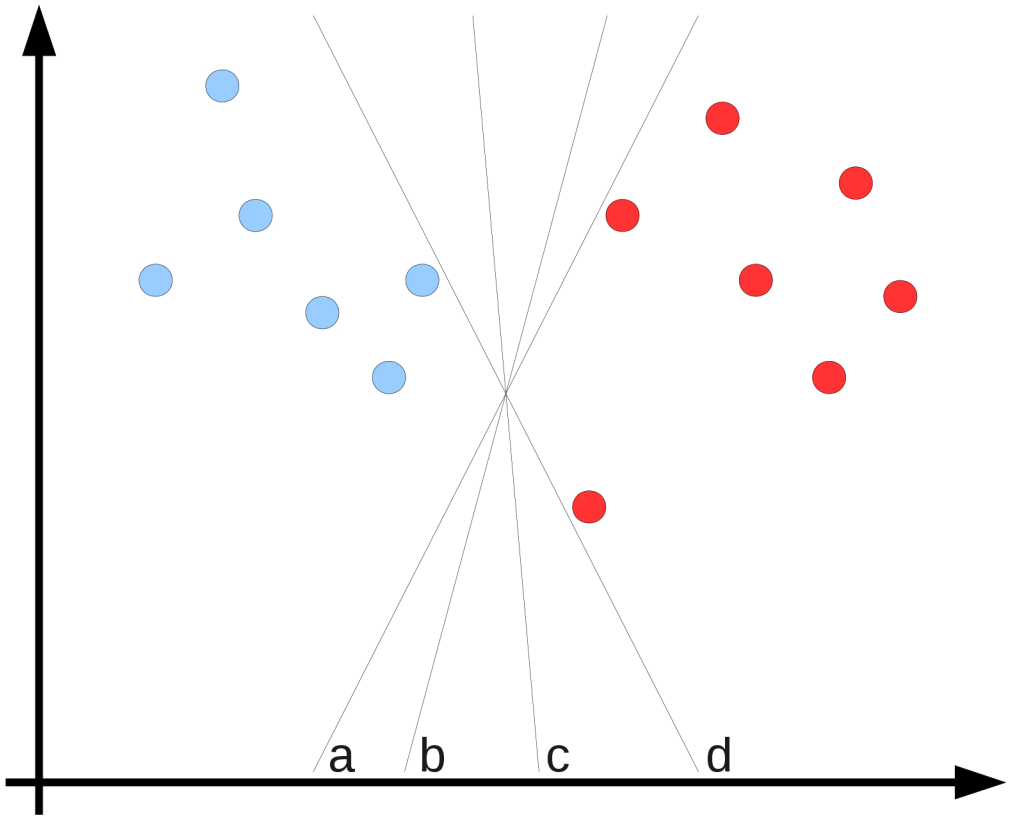
Jak je vidět na obrázku 2.2, všechny separační přímky dobře oddělují data různých tříd, ale ani o jedné z nich nemůžeme říct, že je naprosto optimálním řešením. Každá z nich vždy nějakou ze tříd zvýhodňuje tím, že je od této třídy ve větší vzdálenosti. Tudíž do ní může být klasifikováno větší množství dat. Tento problém se snaží vyřešit metoda *Support Vector Machine*, která hleda právě takovou přímku, která v klasifikaci nezvýhodňuje ani jednu ze tříd.

Přímky a, b, c a d na obrázku 2.2 vždy procházejí počátkem souřadnic, tedy bodem $[0, 0]$. Pokud ovšem data nemáme rozprostřena právě kolem počátku souřadnic musíme přepočítat jejich souřadnice tak, aby rozhodovací linie procházela tímto bodem [3].

2.8 SVM(support vector machine)

Jedná se o skupinu příbuzných metod strojového učení s učitelem, které se používají pro klasifikaci a regresi. SVM konstruuje nadrovinu nebo množinu nadrovin ve vícedimenzionálním prostoru, který může být použit pro klasifikaci, regresi nebo jiné úlohy [13]. Klasifikace se snaží zařadit daný objekt do určité výstupní třídy na základě poznatků, které byly získány ve fázi učení. Regrese určuje hodnotu jedné proměnné v závislosti na jedné nebo více dalších proměnných. Metoda SVM se využívá pro kategorizaci textů, rozpoznávání obrazů nebo také v lékařství.

Základní princip SVM je převod klasifikovaných prvků původního prostoru do vícedimenzionálního prostoru, ve kterém už je možné jednotlivé třídy prvků oddělit lineárně. Tomuto principu se říká *Kernel Trick* [1].



Obrázek 2.2: Určení separační linie

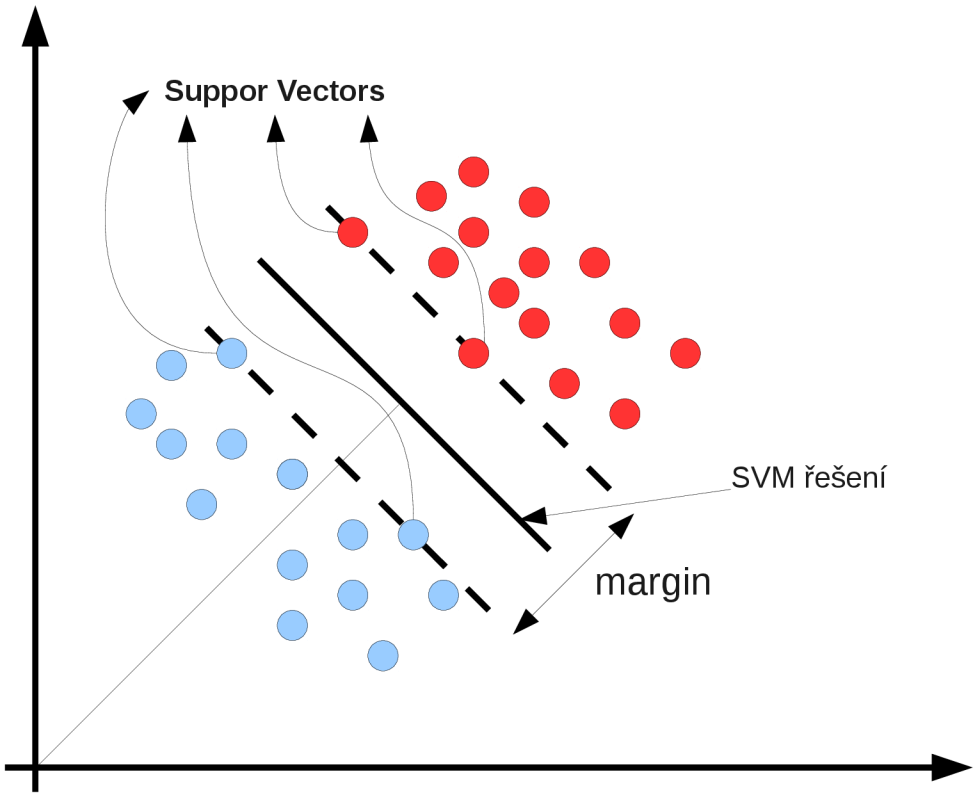
Díky použití dalšího prostoru přibude každému prvku další souřadnice, která tyto prvky posune. Tímto posunem se oddělí klasifikované prvky, které už lze lineárně oddělit pomocí nadroviny. Při transformaci dvojrozměrného prostoru do trojrozměrného závisí třetí souřadnice prvků na prvních dvou souřadnicích.

Pro klasifikaci používáme knihovnu *libSVM* [4]. Knihovna *libSVM* je jednoduchý, snadno použitelný software pro SVM klasifikaci. Cílem je pomoci uživatelům z jiných oblastí, používat tento software jako nástroj. *LibSVM* poskytuje jednoduché rozhraní, kde uživatelé mohou snadno propojit vlastní programy a poté graficky prokázat rozpoznávání vzorů.

Podrobněji je knihovna *libSVM* popsána v kapitole 2.12.

2.9 *Kernel trick*

Kernel trick je metoda, která umožňuje mapování prvků původního prostoru do prostoru s více dimenzemi. V původním prostoru nejsme schopni data od sebe oddělit lineární separační linií, proto je musíme pomocí této metody převést do vícerozměrného vektorového prostoru, kde již lineární oddělovač existuje a kterým může být například rovina nebo nadrovina [11]. Formální definice tranformace na lineárně separabilní úlohu je relativně komplikovaná a přesahuje rámec této práce.



Obrázek 2.3: Princip Support Vector Machines

2.10 Řetězcové funkce

Řetězcové funkce SVM poskytují schopnost implicitního mapování nelineárně separovatelných bodů do odlišného prostoru, kde už jsou tyto body lineárně separovatelné. Nejpoužívanější jádra vhodná pro SVM jsou např. *Polynomial Kernel*, *Gaussian Radial*, *Basis Kernel* či *Hyperbolic Tangent Kernel* [10].

Většina jádrových funkcí pro SVM pracuje pouze s číselnými daty, která jsou ovšem nevyhovující pro internetovou bezpečnost, kde je prezentováno obrovské množství řetězcových dat. Za účelem rozšíření SVM pro zpracování řetězcových dat jsme implementovali následující řetězcové algoritmy.

- Gap-Weighted Subsequence kernel
- Levenshtein distance
- Bag of Words kernel
- N-Gram kernel

2.10.1 Gap-Weighted Subsequence kernel

Teorie *Gap-Weighted Subsequence kernel* je popsána v knize *Kernel Methods for Pattern Analysis* [9]. Hlavní myšlenka, která stoji za touto metodou, je v porovnání řetězců pomocí počtu shodných podsekvencí, které tyto řetězce obsahují - čím více podsekvencí a méně mezer obsahují, tím podobnější si jsou. Prostor vlastností této metody je definován jako

$$\phi_u^p(s) = \sum_{i:u=s(i)} \lambda^{l(i)}, u \ \in \Sigma^p, \tag{2.4}$$

kde $\lambda \in (0,1)$ je chybový faktor, i je index výskytu podsekvence $u = s(i)$ v řetězci s a $l(i)$ je délka řetězce s. My zatěžujeme výskyt u s exponenciálním chybovým faktorem $\lambda^{l(i)}$. Přidružený kernel je definován jako

$$\kappa(s,t) = \langle \phi^p(s), \phi^p(t) \rangle = \sum_{u \in \Sigma^p} \phi_u^p(s) \phi_u^p(t). \tag{2.5}$$

V rovnici (2.5) je nutné provést pomocnou dynamickou programovací tabulku DP_p, jejíž položky jsou:

$$DP_p(k,l) = \sum_{i=1}^{k} \sum_{j=1}^{l} \lambda^{k-i+l-j} \kappa_{p-1}^S(s(1:i), t(1:j)). \tag{2.6}$$

Potom výpočetní(aritmetická) složitost je ohodnocená jako:

$$\kappa_p^S(sa, tb) = \begin{cases} \lambda^2 DP_p\left(|s\ |, |t|\right) & \text{když } a = b; \\ 0 & \text{jinak} \end{cases} \tag{2.7}$$

kterak vyplývá, že pro samotnou hodnotu p je výpočetní složitost rovna $O(|s|\,|t|)$. Tudíž celková výpočetní složitost $\kappa_p(s,t)$ je $O(p\,|s|\,|t|)$.

2.10.2 Levenshtein distance

Metoda *Levenshtein distance* nebo také *Edit distance* počítá rozdíly mezi dvěma řetězci. Rozdílem je myšlen počet všech nahrazení, mazání nebo vložení potřebných ke změně řetězce s s délkou n na řetězec t s délkou m. Formální definice *Edit distance* je dána takto: Je dán řetězec s, nechť $s(i)$ zastupuje jeho $i^{-\text{tý}}$ znak. Pro dva znaky a a b, je definováno

$$r(a,b) = 0 \text{ když } a = b.\ A\ r(a,b) = 1 \tag{2.8}$$

Za předpokladu dvou řetězců s a t s délkami n respektive m, pak pole $(n+1)(m+1)$ d poskytuje požadované hodnoty *Edit distance* $L(s,t)$.

Výpočet d je rekurzivní procedura. První sada $d(i,0) = i$, $i = 0,1,...,n$ a $d(0,j) = j$, $j = 0,1,...,m$; Poté pro ostatní páry i a j máme

$$d(i,j) = min(d(i-1,j)+1,\ d(i,j-1)+1,\ d(i-1,j-1)+r(s(i),\ t(j))). \tag{2.9}$$

V naší implementaci používáme $D = e^{-\lambda \cdot d(i,j)}$ pro získání lepších výsledků. Analogicky k výše uvedené řetězcové funkci je výpočetní složitost *Edit distance* $O(|s|\,|t|)$. V případě, že s a t mají stejnou délku, je složitost $O(n^2)$.

2.10.3 Bag of Words kernel

Tato metoda je reprezentována jako neuspořádaná kolekce dat přehlížející gramatiku a slovosled. Slova jsou sekvence písmen ze základní abecedy oddělené interpunkcí nebo mezerami(bílými znaky). My tuto metodu reprezentujeme jako vektor v prostoru, ve kterém je každá dimenze asociována s jedním termínem ze slovníku.

$$\phi : d \mapsto \phi(d) = (tf(t_1, d), tf(t_2, d), ...tf(t_N, d)) \in \mathbb{R}^N, \tag{2.10}$$

kde $tf(t_i, d)$ je četnost termínu t_i v dokumentu d. Proto je dokument mapován do prostoru o N rozměrech. Tento prostor je velikosti slovníku, typicky velmi velké číslo [9].

2.10.4 N-Gram kernel

Tato metoda převádí dokumenty do vysoko rozměrových charakteristických vektorů, kde každá vlastnost odpovídá sousedícímu podřetězci. Prostor vlastností sdružený s N-Gram kernelem je definován jako

$$\kappa(s, t) = \langle \phi^n(s), \phi^n(t) \rangle = \sum_{u \in \Sigma^n} \phi_u^n(s) \phi_u^n(t) \tag{2.11}$$

kde

$$\phi_u^n(s) = |\{(v_1, v_2) : s = v_1 u v_2\}|, u \in \Sigma^n.$$

Pro výpočet N-Gram kernelu využíváme prostého přístupu, proto je časová složitost $O(n\,|s|\,|t|)$.

2.11 Program *kernels*

Program *kernels* slouží k vytváření *precomputed* matic obsahující míry podobnosti porovnání všech řetězců ze vstupního souboru. Jsou v něm implementovány řetězcové funkce, které počítají míry podobnosti dvou řetězců. Tyto hodnoty jsou uloženy v *precomputed* matici, které jsou vstupem do programu *svm-train* z knihovny *libSVM*. Každému řádku odpovídá příslušný řetězec vstupního souboru.

Zdrojové kódy programu *kernels* jsem upravoval jenom minimálně a změny neměly výraznější vliv na běh programu. Z tohoto programu jsem vycházel při implementaci řetězcových funkcí do knihovny *libSVM*.

2.12 Knihovna *libSVM*

Knihovna *libSVM* je ucelený software pro *support vector machines* klasifikaci, který podporuje vícetřídní klasifikaci [4]. Pomocí této knihovny můžeme řešit nejrůznější druhy problémů, jako jsou např. klasifikaci obrazu, hlasu nebo textu. Využití prakticky není omezeno, ale musíme být schopni transformovat klasifikovaná data do formátu, který je vhodný pro knihovnu *libSVM*. Pro vykreslení výsledků klasifikace do grafu lze knihovnu *libSVM* propojit s programem *gnuplot*. Ke knihovně *libSVM* existuje mnoho rozšíření a vlastních implementací knihovny v různých programovacích jazycích. Za zmínku stojí např. rozhraní pro výpočetní prostředí Matlab či implementace pomocí jazyků Java, C#, Python nebo LISP.

Knihovna je také rozšířena o grafické uživatelské rozhraní, které je dostupné přímo na stránkách projektu, díky němuž si lze vyzkoušet základní funkčnost. Implementován je také

nástroj, který obyčejnému uživateli usnadní proces klasifikace. Provede ho od přípravy dat až po trénování a testování klasifikátoru a zobrazení výsledků.

V mojí práci jsem pracoval pouze s variantou knihovny *libSVM*, která umožňuje klasifikaci řetězcových dat. Řetězcová verze obsahuje tři programy, které vzniknou přeložením zdrojových souborů knihovny.

Program *svm-train* slouží k trénování modelu a vytváří soubor, který tento model popisuje. Druhý program s názvem *svm-predict* načítá soubor s popisem modelu generovaný předchozím programem a počítá procentuální úspěšnost klasifikace. Výstupem je již zmíněná procentuální úspěšnost a soubor s výsledky testování. Struktury výše popsaných výstupních souborů jsou popsány v kapitole 2.13.2. Posledním programem je program *svm-scale*, ale ten nebyl v náší práci využit.

Všechny tři výše zmíněné programy používají knihovnu, ve které je vlastní implementace řetězcových funkcí a ostatní objekty a funkce pro správný chod klasifikace.

Úpravy knihovny *libSVM*

Byl jsem nucen poupravit pouze zdrojové kódy vlastního knihovny, která je složena ze zdrojových souborů *svm.cpp* a *svm.h*. Hlavní úprava knihovny spočívala v iplementaci řetězcových funkcí, které doposud v knihovně *libSVM* implementovány nebyly. S tím byla spojená i úprava souvisejících datových struktur. Jednalo se zejména o datovou strukturu *svm_parameter*.

Musel jsem také upravit veškeré větvící konstrukce, které byly nezbytné pro správný běh programu. Dále bylo nutné poupravit datovou strukturu *svm_parameter*, ve které jsou uloženy hodnoty parametrů nezbytných pro správný běh trénování klasifikátoru. Jedná se o parametry *lambda*, *substr_lenght* a *str_lenght*, které ovlivňují výpočet pouze řetězcových funkcí a doposud nebyly v knihovně implementovány. Poslední úprava zdrojového kódu knihovny bylo ukládání výše zmíněných parametrů do souboru, který popisoval model. Tento soubor je vstupem programu *svm-predict*.

Programy *svm-train*, *svm-predict* a *svm-scale* nebyl důvod jakkoliv upravovat. Program *svm-scale* jsme v naší práci k ničemu nepotřebovali, a tudíž ani nebyl generován spustitelný binární soubor.

2.13 Klasifikovaná data a jejich normalizace

Aby bylo možné provádět klasifikaci, je zapotřebí získat vstupní data, která budou sloužit pro klasifikaci. Tato data jsou ve formátu prostého textu, logů síťových provozů nebo speciálních řetězcových sekvencí a jsou získávána z dříve vyjmenovaných zdrojů pomocí programů, které tyto data dokáží zpracovat a správně zobrazit(*wireshark*, *tcpdump*, *thunderbird*, ...).

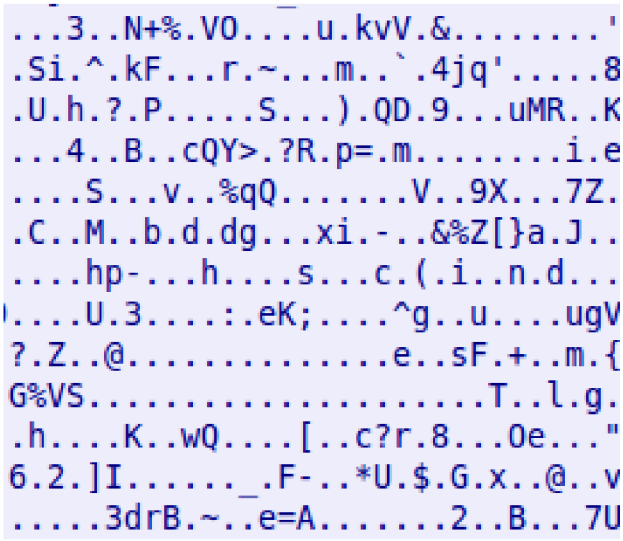
Jelikož vstupní data nejsou ihned vhodná pro klasifikaci, musí se provézt jejich normalizace. Například u textových dat normalizace spočívá ve změně velkých znaků abecedy na malé a úpravě velikosti řetězce. Příliš malé řetězce se na optimální velikost zvětšují kopírováním a příliš velké ořezáváním. U ostatních typů vstupních dat se jedná o vymazání zbytečných znaků nebo sekvencí, úpravě velikosti, tak jako u textových dat nebo nahrazení některých znaků za jiné znaky. V této práci se věnuji jenom klasifikaci řetězcových dat získaných z *Reuters* datasetu.

Trénovací a testovací dataset jsem měl již předpřipravený, a tudíž jsem nemusel provádět žádné úpravy týkající se normalizace dat. Příklady typů vstupních dat jsou na obrázcích

2.4, 2.5 a 2.6.

```
Message-ID: <4B6C41BD.8080702@stud.fit.vutbr.cz>
Date: Fri, 05 Feb 2010 17:05:17 +0100
From: =?ISO-8859-2?Q?Tom=E1=B9_Kubern=E1t?= <xkuber00@stu
User-Agent: Thunderbird 2.0.0.23 (X11/20090817)
MIME-Version: 1.0
To: "xkuber00@fit.vutbr.cz" <xkuber00@stud.fit.vutbr.cz>
Subject: BP zprava
Content-Type: text/plain; charset=ISO-8859-2; format=flow
Content-Transfer-Encoding: 8bit
X-Spam-Score: 0 ()
X-Scanned-By: MIMEDefang 2.64 on 147.229.176.14
```

Obrázek 2.4: Emailová zpráva zobrazená v programu Mozilla Thunderbird



Obrázek 2.5: Data z programu Wireshark

```
7 gencorp st qtr shr cts vs cts gencorp
7 gelco corp nd qtr shr cts vs cts gelc
8 twa confirms ownership of pct of usai
8 waste management ends tender offer fo
9 eia says distillate stocks off mln ga
9 reagan says u s must do more to lesse
5 u s corn soybean acreage estimates co
5 u s export inspections in thous bushe
```

Obrázek 2.6: Obyčejná textová data kolující po internetu

2.13.1 Formát vstupních dat

Pro oba dva způsoby klasifikace jsem měl jako vstup stejné soubory jak v rámci trénování, tak i v rámci testování. Vstupní data byla reprezentována textovým souborem, kde na každém řádku by jeden řetězec uvozený číslem třídy, do které patřil. Číslo třídy a příslušný řetězec byly od sebe odděleny mezerou. Soubor s trénovacími daty má 380 řetězců a soubor s testovacími daty má 90 řetězců. Data v každém souboru jsou rozdělena do čtyř tříd. Ukázka souboru s řetězcovými daty je vidět na obrázku 2.6.

V mojí práci pracuji s daty patřícími do tříd 1, 2, 3 a 4, ale použití tohoto číslování není žádným pravidlem.

2.13.2 Formáty výstupních dat

Precomputed matice

Soubor generovaný programem *kernels* je tzv. *precomputed* matice. Jedná se o textový soubor, který obsahuje matici s výsledky porovnání všech řetězců. Míry podobnosti jednotlivých porovnání jsou uloženy ve *sparse* formátu.

Sparse formát je způsob uložení dat, ve kterém není potřeba uchovávat nulové hodnoty. Například následující datová sekvence

1 0 2 0

je ve *sparse* formátu reprezentována jako

1:1 3:2

Popis tohoto formátu je vysvětlen na stránkách knihovny *libSVM* [4]. V mé práci jsou data v *precomputed* matice uložena ve *sparse* formátu, ale jsou uloženy i nulové hodnoty.

Precomputed matice má na každém řádku uloženu informaci o porovnání jednoho řetězce s ostatními řetězci. První částí každého řádku je číslo třídy, do které patří řetězec související s aktuálním řádkem. Druhá část je už ve *sparse* formátu a značí pořadové číslo tohoto řetězce ve vstupní množině dat. Za touto částí následuje sekvence hodnot znamenající míry podobnosti právě testovaného řetězce se všemi ostatními. Veškeré hodnoty jsou odděleny mezerami a kromě čísla třídy, která je ihned na začátku, jsou ve *sparse* formátu.

Jako ukázka poslouží následující příklad,

1 0:5 1:0.081 2:0.133

kde 1 je příslušnost do třídy, 0:5 znamená, že se jedná o 5. řetězec, 1:0.081 značí míru podobnosti 5. řetězce s 1. řetězcem a 2:0.133 je míra podnobnosti 5. a 2. řetězce.

Model file – soubor s popisem modelu

Soubor je vytvořen programem *svm-train* z trénovací množiny. Na prvních řádcích tohoto souboru jsou uloženy hodnoty parametrů, které přímo ovlivňují to, jakým způsobem bude klasifikace prováděna. Jedná se např. o typ svm, typ kernel funkce, typ dat, počet a označení tříd nebo naše nově implementované parametry.

Za výčtem hodnot všech parametrů následuje seznam zpracovaných řetězců tak, jak jsou uloženy v trénovacím souboru a jak byly zpracovány. Narozdíl od vtupního souboru jsou zde řetězce uvozeny čísly, kterých je o jedno méně než je celkový počet tříd. Tato čísla označují *support* vektory mezi třídou příslušného řetezce a ostatními třídami. Z tohoto důvodu je počet čísel *support* vektorů o jedno menší, než je celkový počet tříd.

***Predict* file – výsledný soubor klasifikace**

V tomto souboru není uloženo nic jiného, než čísla tříd, do kterých byly zařazeny řetězce v rámci testování. Řádek *predict* souboru odpovídá řetězci, který je na stejném řádku v souboru obsahujícím testovací data.

2.13.3 Charakteristika současného řešení

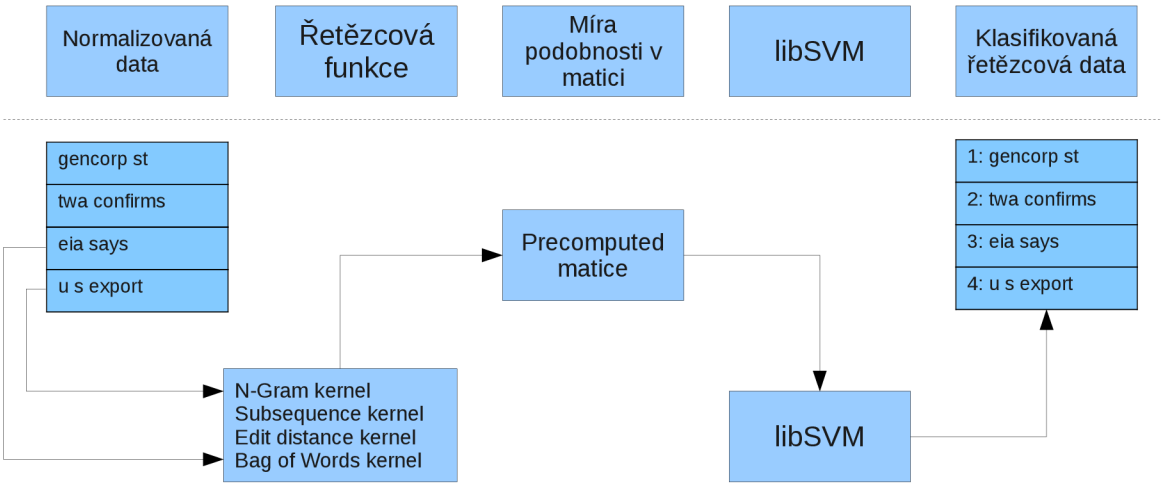
Po sběru dat a jejich normalizaci jsou příchozí data zpracována natolik, že je možné přistoupit k samotnému procesu klasifikace. Všechna data jsou nejprve načtena do operační paměti a poté je prováděno jejich porovnávání.

V mém případě se pro 100 řetězců volá řetězcová funkce 5050x, protože se neporovnává každý řetězec s každým, ale vždycky aktuální řetězec se všemi přecházejícími a se sebou samým. Na pořadí řetězců při porovnání nezáleží, a proto může být výsledek zapsán na obě strany diagonály *precomputed* matice, čímž se zajistí porovnání každého řetězce s každým. Poté již závisí jenom na rychlosti výpočtu dané metody, jak dlouho bude celý proces trvat. V závislosti použité řetězcové funkce jsem se pohyboval od 30 vteřin do 2 hodin v rámci trénování a od 2 vteřin do 7 minut v rámci testování.

Trénovací datový soubor obsahoval 380 řetězců, tudíž bylo zapotřebí 72390 porovnání. Na testovacím souboru, který obsahuje 90 řetězců se provádělo 4095 porovnání.

V každé iteraci cyklu se vezmou dva řetězce, které jsou právě na řadě, a jako parametr je oba předám ke zpracování do námi zvolené řetězcové funkce. Výsledkem této funkce je tzv. *míra podobnosti*, která udává jak moc si jsou dva řetězce podobné. Program, který nám nakonec provádí vlastní klasifikaci vezme tuto *precomputed* matici a provede testování nebo *cross* validaci a tím vypočítá procentuální úspěšnost klasifikace.

Vlastní princip klasifikace je znázorněn na obrázku 2.7.



Obrázek 2.7: Vlastní princip klasifikace

2.13.4 Charakteristika nového řešení

Pro časovou náročnost předešlého řešení jsem byl nucen implementovat holé řetězcové funkce do programu *libSVM* [4]. Toto úskalí se snažím obejít tím, že výsledky z řetězcové funkce neukládám do matice, ale dávám je přímo na vstup programu *libSVM*, který už vrátí klasifikovaná řetězcová data. V této variantě klasifikace volám řetězcovou funkci 161120x v rámci trénování a 30780x v rámci testování. Oproti klasifikaci programem *kernels* je zde vidět několikanásobný nárůst počtu porovnávání. Je to markantní zejména v testovací fázi, kdy při zpracovávání 90 řetězců, je prakticky 7,5x větší počet porovnávání.

Zrychlení u tohoto způsobu klasifikace nebylo naměřeno pouze při použití řetězcové metody *Gap-Weighted Subsequence kernel*, kde se časová náročnost zvyšovala spolu se zvyšováním parametru *substr_lenght*, tedy délky podřetězce, který je nezbytný pro samotný výpočet této řetězcové funkce. Nejvýznamnější podíl na rychlosti metody má ovšem délka zpracovávaných řetězců. Horší časové hodnoty tato metoda vykazovala pro veškeré hodnoty parametru *subst_lenght* a doba provádění se zvyšovala spolu se zvyšováním hodnoty *substr_lenght*. V tomto případě měla rychlejší běh klasifikace prováděná programem *kernels*. Byl to ovšem ojedinělý případ a v pokusech na ostatních řetězcových metodách klasifikace programem *libSVM* vykazovala lepší časové výsledky, než starší způsob klasifikace programem *kernels*.

Úkolem bylo zjistit, jestli se vyplatí implementace řetězcových funkcí použitých v programu *kernels* do knihovny *libSVM*. Jediný způsob jak toto ověřit spočíval v doplnění řetězcových funkcí do výše zmíněné knihovny a provedení všech potřebných testování a měření. K vytvoření konečného závěru poté stačilo analyzovat jednotlivá měření a vyhodnotit výsledky.

Testování na jiných než textových datech je mimo rozsah této práce.

Kapitola 3

Princip testování

3.1 Experimenty

Pro klasifikaci dat, které mají více tříd používám software pro *support vector machines* s názvem *libSVM* [4]. Ve svých pokusech používám klasifikaci pomocí předpočítané (*precomputed*) matice, ale i klasický přístup s trénováním a testováním na odlišných datových sadách. Výsledkem je jak procentuální úspěšnost klasifikace, tak i čas potřebný pro výpočet. Jelikož je cílem této práce snížení náročnosti výpočtů, tak se právě prioritně zaměřuji na časové výstupy jednotlivých metod klasifikace. Výsledky procentuální úspěšnosti procesu klasifikace jsou sice až na druhé koleji mého zájmu, ale rozhodně nejsou opomíjeny. Rychlý způsob by byl k ničemu, kdyby dával špatné výsledky klasifikace.

Všechny hodnoty v *precomputed* matici jsou škálovány do intervalu [-1,1]. *Precomputed* matice byly použity jako vstup do *libSVM* a jsou vytvářeny programem *kernels* [8].

3.1.1 Klasický způsob testování

Klasický způsob klasifikace je tvořen dvěma běhy programu. V prvním běhu se musí na určitých datech natrénovat a v druhém běhu se již naučený klasifikátor otestuje. Pro obě fáze klasifikace jsou určeny odlišné vzorky dat, kterými jsou trénovací a testovací dataset. Trénovací dataset vždy obsahuje větší množství prvků než dataset testovací. Proces dělení dat se neděje automaticky, ale tuto činnost musí zajistit uživatel programu. Tento způbob testování je použit jak pro klasifikaci programem *kernels*, tak i pro klasifikaci knihovnou *libSVM*.

Při provádění klasifikace programem *kernels* nejprve vytvořím *precomputed* matici z trénovacích dat, dále následuje vytvoření *precomputed* matice z testovacích dat a nakonec provedu testování programem *svm-train* s využitím *cross* validace. *Precomputed* matici tvořenou z testovacích dat k vlastní klasifikaci nepoužívám a slouží mi jenom pro výpočet celkové doby trvání klasifikace.

V případě použití knihovny *libSVM* spustím program *svm-train* na trénovací množinu a výsledkem je soubor s ohodnocením všech řetězců, které byly zpracovány. Formát tohoto souboru je blíže popsán v kapitole 2.13.2. Dalším krokem je spuštění programu *svm-predict*, jehož parametrem je soubor s testovacími daty, potom soubor, který je výsledkem trénování a nakonec výstupní soubor.

Výsledkem obou přístupů ke klasifikaci je procentuální vyjádření úspěšnosti klasifikace.

3.1.2 *Cross* validace

Jedná se o další způsob testování a klasifikace. Především se *cross* validace používá pro nelezení optimálních hodnot parametrů při trénování klasifikátoru a k testování přesnosti modelu.

Cross validace spočívá v tom, že jsou trénovací data rozdělen na určitý počet skupin. Vždy se jedna skupina dat vybere a označí se jako testovací. Zbylé skupiny jsou seskupeny do jedné, čímž vznikne trénovací množina. Poté je trénovací množina využita na natrénování klasifikátoru. Po natrénování se ještě klasifikáto musí otestovat, což je učiněno zbylou testovací množinou. Tento proces s opakuje tak, aby každá ze skupin byla právě jednou skupina, se kterou se provádí testování. Výsledkem této metody je procentuální úspěšnost správně klasifikovaných dat.

Počet skupin, do kterých jsou vstupní data dělena, je zadáván před spuštěním vlastní *cross* validace. Tento počet udává kolikanásobná *cross* validace se bude provádět.

Výběr prvků do skupin se řídí pseudonáhodným algoritmem, tudíž pro stejnou sadu dat budu mít vždy stejný výsledek, protože *cross* validační algoritmus rozdělí vstupní dataset vždy do naprosto stejných skupin. Tomuto lze předejít inicializací počátečního stavu, tzv. *seedu*, např. funkcí srand(time()). *Seed* potom nebude vždy stejný, ale v každém běhu programu se bude lišit. Díky tomu budu vždy dostávat jinačí výsledky *cross* validace. Ve své práci toto nastavování *seedu*, na doporučení vedoucího, nepoužívám.

Cross validaci spustím příkazem *svm-train* s parametrem *-v n*, kde *n* je celé číslo, které značí o kolikanásobnou *cross* validaci se jedná.

Tento způsob klasifikace většinou dává přesnější a hodnotnější výsledky než klasifikace běžným způsobem, kde mám pevně danou jednu sadu dat pro testování a druhou sadu dat pro trénování.

V této práci používám *cross* validaci pouze pro získání procentuální úspěšnosti z *precomputed* matice. Tento způsob má už míry pravděpodobnosti vypočítané a ani jednou samotnou řetězcovou funkci nevolá, díky čemuž trvá tento běh naprosto zanedbatelnou dobu. Opačný případ nastává při provádění cross validace přímo na souboru s trénovacími daty. Zde už program *svm-train* musí počítat míry podobnosti, a tudíž program trvá o mnoho déle, než pomocí klasického způsobu popsaného v kapitole 3.1.1. Při 5-ti násobné *cross* validaci prováděné na trénovací množině dat každý z pěti běhů volá řetězcovou funkci 125000krát, tudíž výsledný počet volání je 625000. Nejmarkantnější byl tento jev zřetelný u metody *Gap-Weighted Subsequence kernel*, u které se dosahované časy *cross* validace pohybovaly okolo 6,5 hodiny.

Namísto *cross* validace používám výše zmíněný, klasický způsob klasifikace uvedený v kapitole 3.1.1.

3.1.3 Sledovaná data

Úkolem této práce bylo implementovat řetězcové funkce do programu *libSVM* [4] a změřit časové rozdíly mezi původním a novým způsobem klasifikace dat.

K měření času, po který běžel program, jsem použil shellovou utilitu *time*. Vždy jsem příkaz *time* spouštěl s parametrem *-p*, což vypíše na standardní chybový výstup časy *real*, *user* a *sys*.

Čas *real* reprezentuje celkový běh programu od jeho spuštění po jeho ukončení a zahrnuje i dobu, kdy program čeká a nezatěžuje systém. Čas *user* znázorňuje opět dobu běhu programu, ale už bez čekání. Je to právě tento čas, na který se zaměřujeme. Jedná se

o dobu, která znázorňuje čistý čas procesu na procesoru. Poslední z naměřených časů, čas *sys*, udává, jakou dobu strávil proces v režimu jádra.

Spolu s mojí aplikací nebyly na pozadí spuštěny žádné jiné kromě procesů zajišťující samotný chod systému. Čím déle program běžel, tím větší byly rozdíly časů *real* a *user*.

Zároveň s měřením času jsem sledoval i procentuální úspěšnosti obou způsobů klasifikace. Ty jsou spolu s pomocnými výpisy vypisovány na standardní výstup. Pro záznam těchto hodnot stačí tento výstup přesměrovat do souboru.

Režim jádra (*kernel mode*)

Režim jádra je způsob běhu programu v operačních systémech unixového typu. Když běží program v tomto režimu, tak má k dispozici rozšířenou sadu příkazů pro práci s operační pamětí a operačním systémem. V režimu jádra lze přistupovat ke všem funkcím, lze provádět veškeré instrukce nebo zasahovat kamkoliv do paměti [5].

Uživatelský režim (*user mode*)

Narozdíl od režimu jádra má uživatelský režim spoustu omezení. Tento režim má sadu instrukcí omezenou a již nemůže přistupovat kamkoliv do operační paměti. Má privilegia pracovat pouze se svým adresním prostorem, který dostal přidělen. Programy spuštěné obyčejným uživatelem pracují většinu času právě v uživatelském režimu [5].

3.1.4 Testovací sestava

Jako testovací sestava posloužil asi půl roku starý počítač s nově nainstalovaným operačním systémem GNU/Linux. Jednalo se o linuxovou distribuci Ubuntu 9.04 s kódovým označením Karmic Koala. Jádro linuxového systému bylo verze 2.6.31-10-generic v x86_64 variantě. Programy byly přeloženy překladačem GCC ve verzi 4.4.1(program *kernels*) a 4.2.4(programy *svm-train* a *svm-predict*). Musely být použity dvě verze překladače, protože verze GCC překladače vyšší než 4.2 špatně překládaly program *svm-train*. Konkrétně se jednalo o výpisy varování při načítání dat ze souboru, u kterých nebyly ošetřeny návratové hodnoty funkce *fscanf*. Jednalo se bohužel jenom o systém, na kterém běželo testování. Na jiných systémech tento problém nenastal. Naštestí tato komplikace neznamenala vážné problémy a všechna testování proběhla v pořádku a bez komplikací.

Procesor od firmy AMD je sice dvoujádrový, ale oba programy nejsou tvořené pro paralelní zpracování a tak tuto výhodu bohužel nemohly využít. Tudíž zrychlení není dáno počtem jader, ale spíše výkonem a rychlostí procesoru jako celku. Pokud procesor běží naplno, tak pracuje na kmitočtu 2800 MHz, kdežto v klidu procesor běží na taktu 800 MHz.

Operační paměť také nijak výrazně naše experimenty neovlivnila. Většina pokusů měla minimální využití operační paměti co se týče procentuálního zaplnění. Toto bylo dáno její dostatečnou velikostí, která byla 4GB a tento fakt je podpořen i tím, že při testování nebyl spuštěn žádný další proces, který by potřeboval ve větší míře jak procesor, tak i operační paměť.

V opačném gardu naopak pracoval na mém procesu procesor. Ten byl vždy mým procesem vytížen nad 90 % a mému procesu byl procesor odebrán jenom tehdy, když se střídal s procesy zajišťujícími samotný běh operačního systému.

Pevný disk pro moje testování sloužil jenom jako instalační medium pro operační systém a bezprostředně testování neovlivnil, protože velikosti ukládaných matic programem *kernels*

byly prakticky zanedbatelné. Z tohoto důvodu parametry pevného disku nejsou v tabulce 3.1 uvedeny.

Veškerá testování byla spouštěna jako jediná aplikace v systému. Tímto bylo zařízeno větší vytížení procesoru právě našemi procesy. Můj program se na procesoru nemusel nikdy střídat s žádným dalším, významně zatěžujícím procesem, ale střídal se jen s procesy zajišťujícími samotný běh systému.

Detailnější popis parametrů počítače je uveden v tabulce 3.1.

	Název	Takt[MHz]	Velikost
Procesor	AMD Athlon X2 240	2800	2048kB L2 cache
Operační paměť	A-Data	800	2x 2GB

Tabulka 3.1: Důležité parametry testovacího počítače.

Klasifikace programem *kernels*

Vlastní průběh klasifikace pomocí programu *kernels* probíhá ve třech bězích. Prvním během je generování trénovací *precomputed* matice. Data jsou načítána naprosto stejně jako v programu *svm-train*. Program po řádcích načte data z trénovacího souboru a provádí porovnání řetězců. Ve výsledku je porovnán každý řetězec s každým, ale implementace je odlišná. Princip porovnání je podrobněji popsán v kapitole 2.13.3. Stejné řešení je použito i pro druhý běh. Tentokráte se používá testovací dataset a opět je vygenerována *precomputed* matice programem *kernels*.

Třetí finální fáze je už vlastní klasifikace. Jako vstup je použita *precomputed* matice vytvořená ve fázi trénování. Na této matici provedeme programem *svm-train* z knihovny *libSVM* 5-ti násobnou *cross* validaci, čímž získáme procentuální úspěšnost klasifikace.

Protože mi jde především o měření časové náročnosti jednotlivých přístupů, tak si musím časové hodnoty jednotlivých běhů měřit. Pro každou kombinaci parametrů jsem měření opakoval 5krát, abych tyto hodnoty nakonec mohl zprůměrovat. Průměrné hodnoty všech tří fází jsem potom sečetli a vyšla nám celková doba potřebná pro klasifikaci programem *kernels*.

Pro měření času mi posloužila shellová utilita *time*, která s parametrem *-p* vypíše počet sekund, po které proces běžel na procesoru. Popis jednotlivých částí výpisu je uveden v předcházející kapitole.

Klasifikace pomocí knihovny *libSVM*

Klasifikace programy *svm-train* a *svm-predict* probíhá obdobně jako klasifikace programem *kernels*, ale s tím rozdílem, že etapy už nejsou tři, ale jenom dvě.

První etapa je opět proces trénování. Děje se na trénovací množině dat a provádí se programem *svm-train*. Výstupem z tohoto programu je soubor s popisem modelu a generuje se právě z trénovacích dat. Tento soubor má svoji koncovku implicitně nastavenou na *.model*, ale jedná se o obyčejný textový soubor. Ovšem je na uživateli, jestli koncovku souboru změní nebo ji nechá implicitně nastavenou. Formát tohoto souboru je detailněji popsán v kapitole 2.13.2.

Program *svm-predict* potom načítá soubor s testovacími daty a soubor s popisem modelu vygenerovaný programem *svm-train* v rámci trénování. Výstupem je úspěšnost klasifikace vyjádřená poměrem dobře klasifikovaných dat a všech testovacích dat a procentuální

úspěšností. Dalším výstupem programu *svm-predict* je soubor s výsledky klasifikace. Jedná se o soubor obsahující na řádcích oddělená čísla tříd, které určil klasifikátor. Soubor má implicitně nastavenou koncovku na *.predict*, ale uživatel ji může pohodlně změnit. Jako v případě souboru popisující model, je i tento soubor obyčejný textový soubor a formát tohoto souboru je blíže popsán v kapitole 2.13.2. *Svm-predict* volá řetězcové funkce, které jsou implementovány v knihovně *libSVM*. Jak je zmíněno výše, tak tyto funkce potřebují hodnoty parametrů, které byly uvedeny v rámci trénování. Ty se programem *svm-predict* načítají právě ze souboru, kterým je popsán model trénovacích dat.

Časy obou fází opět sečtu a dostanu dobu trvání klasifikace knihovnou *libSVM*. Tento čas potom porovnám s časem, který jsem získal změřením doby běhu klasifikace pomocí programu *kernels*. Takto porovnám časy všech kombinací parametrů u obou programů. Analýzou výsledků se zabývám v následující kapitole.

Kapitola 4

Výsledky a porovnání experimentů

V následujících testech jsem vycházel z již naměřených výsledků, které jsou uvedeny v článku *String Kernel Based SVM for Internet Security Implementation* [8]. Hlavním úkolem bylo porovnat mou vlastní implementaci programu *kernels* s řetězcovými funkcemi implementovanými do *libSVM*. Pro každou kombinaci parametrů jsem měření času prováděl 5x, abych mohl dosažené hodnoty způměrovat. Dosáhnu tímto přesnějších a hlavně věrohodnějších výsledků, které nejsou nijak vážně zkresleny různými a předem nepředvídatelnými jevy. Tyto hodnoty jsou znázorněny v tabulkách umístěných v následujících kapitolách. Abych mohl dosažené výsledky porovnat, potřeboval jsem vytvořit referenční řešení. Již jednou naměřené hodnoty se použít nedaly, protože byly prováděny na počítači, na který jsem neměl přístup. Z tohoto důvodu jsem měřil hodnoty poskytované i během starého způsobu klasifikace. Výsledky nového způsobu jsem poté měl možnost porovnat.

Veškeré testování probíhalo pomocí skriptů napsaných v *bashi*, protože usnadňovaly práci a šetřily čas. Díky tomuto přístupu programy nemusejí čekat na spuštění od uživatele, ale jsou spouštěny ihned za sebou automaticky. Navíc se předejde chybám, které by na příkazovou řádku mohl zadat uživatel. Tento přístup také umožňuje lepší zaznamenávání výsledků a to jak časových, tak i procentuálních, protože časové výsledky jsou tisknuty na standardní chybový výstup, kdežto procentuální výsledky jsou tisknuty na standardní výstup. Pro zaznamenávání výsledků jsou oba dva výstupy přesměrovány do jednotlivých souborů.

Pro ulehčení práce jsem si v programu *C++* s pomocí knihovny *QT4* napsal program, který převádí počet sekund zaznamenaný programem *time* do formátu čitelným pro člověka. Tento program jsem nazval *casy* a jako vstup slouží počet milisekund. Program vezme tuto hodnotu a vypíše ji ve formátu *hh:mm:ss*.

V grafech 4.2, 4.4, 4.5, 4.6 a 5.3 jsou vyneseny časy potřebné pro vykonání klasifikace programy *kernels* a *libSVM*, kterým je měněm parametr lambda. Jelikož v programu libsvm měním zároveň parametr C, musím zprůměrovat všechna měření, ve kterých se mění parametr C, abych dostal čas klasifikace pro jednotlivé parametry lambda.

4.1 Experimenty metodou *Gap-Weighted Subsequence kernel*

Klasifikace metodou *Gap-Weighted Subsequence kernel* probíhala ze všech nejdelší dobu a jako jediná měla rychlejší průběh, když byla prováděna programem *kernels*, než když byla prováděna přes knihovnu *libSVM*. Časové hodnoty byly menší pro fázi trénování i pro

fázi testování. Je to dáno vlastním algoritmem řetězcové funkce *Gap-Weighted Subsequence kernel*, kde nejvýraznější podíl na čase jednoho porovnání má délka řetězce. Hlavním důvodem je ovšem počet volání řetězcové funkce, který má knihovna *libSVM* daleko vyšší než program *kernels*, což je popsáno v kapitolách 2.13.3 a 2.13.4.

V porovnání s hodnotami ostatních měření, je zde vidět nárůst doby potřebné pro průběh celého procesu a je jedno, jestli se jedná o trénování nebo testování. Čas potřebný ke klasifikaci touto metodou se zvyšuje spolu se zvyšováním parametru *substr_lenght*, jehož hodnoty byly 4, 8, 12, 16 a 20.

Řetězcová metoda *Gap-Weighted Subsequence kernel* trvala sice nejdelší dobu, ale měla nejlepší výsledky procentuální úspěšnosti klasifikace.

4.1.1 Klasifikace programem *kernels*

Jak už bylo uvedeno v kapitole 4, žádnou klasifikaci uživatel nespouští sám, ale běh programů je zprostředkován pomocí skriptů.

Pro tuto klasifikace jsem použil skripty *subseq_precomputed_run.sh* a *casove_vysledky.sh*. Prvně jmenovaný skript zajišťuje vlastní průběh klasifikace a generuje soubor s hodnotami časových měření. Tento soubor jsem nazval *casy_subseq_precomputed.txt* a slouží jako vstup do skriptu *casove_vysledky.sh*, který vypočítá časové průběhy všech částí klasifikace.

Ukázka použití skriptů:

sh subseq_precomputed_run.sh train00.txt test00.txt 2>casy_subseq_precomputed.txt
sh casove_vysledky.sh casy_subseq_precomputed.txt

Podřetězec	**Čas**		
	real	user	sys
4	26m 27,360s	23m 22,180s	2m 57,348s
8	48m 15,016s	45m 17,076s	2m 49,274s
12	1h 10m 26,516s	1h 7m 34,902s	2m 42,580s
16	1h 32m 29,368s	1h 29m 40,372s	2m 40,004s
20	1h 55m 5,060s	1h 52m 10,378s	2m 45,520s

Tabulka 4.1: Celkový čas klasifikace prováděné programem *kernels* s proměnnou délkou podřetězce pro metodu *Gap-weighted subsequence kernel*.

4.1.2 Klasifikace pomocí knihovny *libSVM*

V tomto způsobu klasifikace jsem využil služeb skriptů *subseq_svm_run.sh*, *casove_vysledky.sh* a *klas_vysledky.sh*. Po skončení vlastní klasifikace pomocí *subseq_svm_run.sh* potřebujeme zpracovat její výsledky. Výpočet časových údajů zajistí *casove_vysledky.sh* a výpis procentuální úspěšnosti *klas_vysledky.sh*.

Použití výše zmíněných skriptů je následující:

sh subseq_svm_run.sh train00.sh test00.sh 2>casy_subseq_svm.txt > vystup_subseq_svm.txt
sh casove_vysledky.sh casy_subseq_svm.txt
sh klas_vysledky.sh vystup_subseq_svm.txt

Výsledky jsou ze skriptů *klas_vysledky.sh* a *casove_vysledky.sh* tisknuty na standardní výstup. Takto získané údaje jsou využity pro vytvoření tabulek 4.2 a 4.3. Graf pro vizuální reprezentaci výsledků klasifikace je uveden v obrázku 4.1.

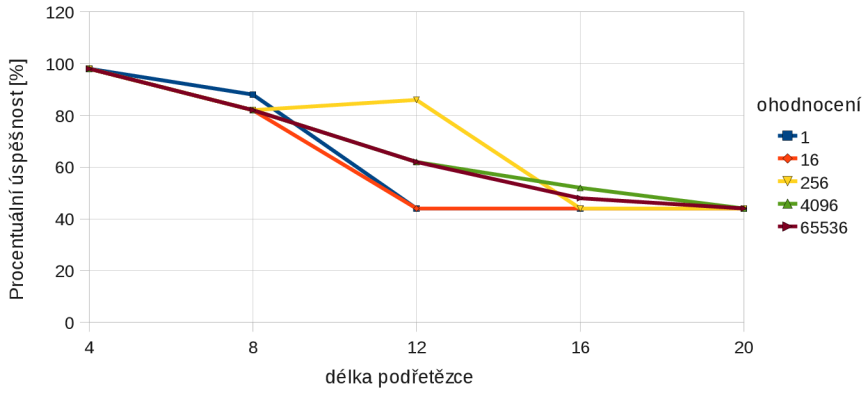
Podřetězec	Čas	**C parametr pro C-SVC** 1	16	256	4096	65536
4	real	34m 18s	33m 9s	31m 58s	31m 59s	33m 23s
	user	27m 18s	26m 45s	25m 57s	25m 56s	26m 48s
	sys	7m	6m 24s	6m 2s	6m 3s	3m 34s
8	real	1h 18m	1h 20m 5s	1h 23m 16s	1h 20m 29s	1h 19m 56s
	user	1h 9m 4s	1h 11m 2s	1h 13m 22s	1h 11m 18s	1h 10m 51s
	sys	8m 56,742s	9m 3,490s	9m 53,736s	9m 10,708s	9m 5,120s
12	real	2h 1m 12s	2h 6m 24s	2h 5m 7s	2h 1m 13s	2h 0m 36s
	user	1h 52m 13s	1h 56m 39s	1h 55m 16s	1h 51m 58s	1h 51m 35s
	sys	8m 58,770s	9m 45,082s	9m 50,422s	9m 15,394s	9m 1,486s
16	real	2h 0m 15s	2h 28m 23s	2h 45m 11s	2h 44m	2h 42m 559s
	user	1h 52m 58s	2h 19m 13s	2h 35m 14s	2h 34m 12s	2h 33m 14s
	sys	7m 16,926s	9m 9,844s	9m 57,082s	9m 48,106s	9m 44,396s
20	real	2h 29m 22s	2h 29m 35s	2h 30m	3h 19m 14s	3h 25m
	user	2h 22m 2s	2h 22m 7s	2h 22m 34s	3h 9m 48m	3h 15m 3m
	sys	7m 19,710s	7m 28,300s	7m 25,724s	9m 25,366s	9m 56,726s

Tabulka 4.2: Celkový čas klasifikace metodou *Gap-Weighted Subsequence kernel* pomocí knihovny *libSVM*.

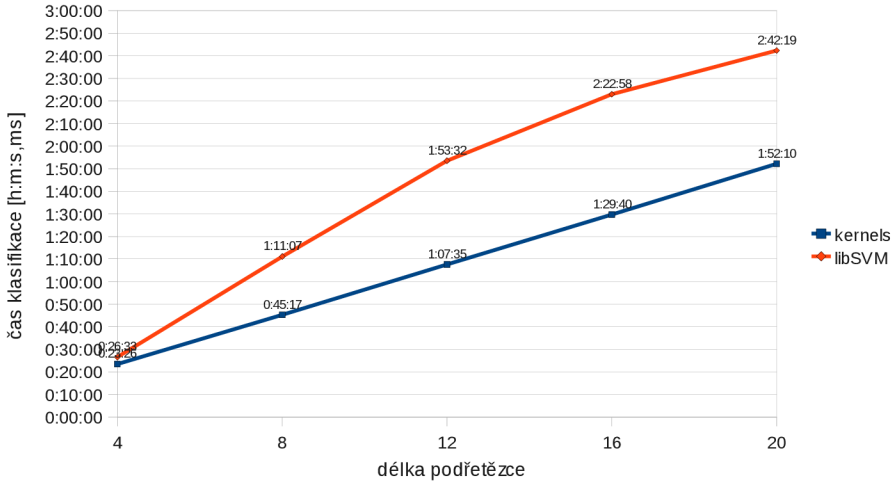
Tabulka 4.3 obsahuje hodnoty získané klasifikací pomocí knihovny *libSVM*. Procentuální úspěšnost pro klasifikaci programem *kernels* byla prováděna pomocí *cross* validace na *precomputed* matici, která byla vytvořena sloučením trénovacích a testovacích dat. Hodnoty vypočítané touto metodou byly prakticky totožné, a tudíž je zde uvedena pouze tabulka s lepšími hodnotami. Výsledky poskytnuté knihovnou *libSVM* jsou lepší v řádech desetin procent a z pohledu úspěšnosti klasifikace je prakticky jedno, který způsob klasifikace bude zvolen.

Podřetězec	**C parametr pro C-SVC** 1	16	256	4096	65536
4	97,8	97,8	97,8	97,8	97,8
8	87,8	82,2	82,2	82,2	82,2
12	44,4	44,4	85,6	62,2	62,2
16	44,4	44,4	44,4	52,2	47,8
20	44,4	44,4	44,4	44,4	44,4

Tabulka 4.3: Procentuální úspěšnost[%] klasifikace knihovnou *libSVM* s využitím metody *Gap-Weighted Subsequence kernel*.



Obrázek 4.1: Graf závislosti procentuální úspěšnosti klasifikace pomocí metody *Gap-Weighted Subsequence kernel* na změně délky podřetězce.



Obrázek 4.2: Porovnání časových průběhů obou způsobů klasifikace na metodě *Gap-Weighted Subsequence kernel*.

4.2 Experimenty metodou *N-Gram kernel*

Řetězcová metoda *N-Gram kernel* byla, co se rychlosti klasifikace týká, druhou nejrychlejší metodou. Vykazovala dobré výsledky i v procentuální úspěšnosti klasifikace, která se pohybovala v rozmezí 44–98%. Tato metoda používala parametr *substr_lenght*, který nabýval hodnot 4, 8, 12, 16 a 20.

4.2.1 Klasifikace programem *kernels*

Jak již výše uvedené postupy, byl i tento prováděn pomocí skriptů. V této fázi mi posloužili skripty *ngram_precomputed_run.sh* a *casove_vysledky.sh*, kterými se provedla klasifikace a spočítali výsledky. Výsledné hodnoty je možno vidět v tabulce 4.4.

Ukázka použití skriptů:

sh ngram_precomputed_run.sh train00.sh test00.sh 2>casy_ngram_precomputed.txt
sh casove_vysledky.sh casy_ngram_precomputed.txt

Podřetězec	**Čas**		
	real	user	sys
4	3m 9,866s	3m 0,880s	748ms
8	3m 8,696s	3m 1,086s	562ms
12	3m 8,058s	3m 0,918s	606ms
16	3m 8,368s	3m 0,944s	402ms
20	3m 8,062s	3m 1,032s	478ms

Tabulka 4.4: Celkový čas klasifikace prováděné programem *kernels* s proměnnou délkou podřetězce použitím řetězcové metody *N-Gram kernel*.

4.2.2 Klasifikace pomocí knihovny *libSVM*

Klasifikace knihovnou *libSVM* probíhala pro řetězcovou metodu *N-Gram kernel* naprosto stejně jako pro zbylé metody. Pomocí skriptů jsem zabezpečil automaticý běh veškerých částí klasifikace a zpracování výsledků. Pro testování tohoto způsobu klasifikace využívám skripty *ngram_svm_run.sh*, *casove_vysledky.sh* a *klas_vysledky.sh*.

Skriptem *ngram_svm_run.sh* provádíme vlastní klasifikaci, tudíž trénování a testování. Časové údaje jsou zaznamenávány do souboru *casy_ngram_svm.txt* přesměrováním standardního chybového výstupu. Výpisy programů *svm-train* a *svm-predict*, nesoucí údaje o vlastní klasifikaci, jsou uloženy v souboru *vystup_ngram_svm.txt*.

Oba dva výstupní soubory jsou zpracovány skripty *casove_vysledky.sh*, pro výpočet časových výsledků, a *klas_vysledky.sh*, pro získání procentuálních výsledků klasifikace.

Použití výše zmíněných skriptů je nýsledující:

sh ngram_svm_run.sh train00.txt test00.txt 2>casy_ngram_svm.txt > vystup_ngram_svm.txt
sh casove_vysledky.sh casy_ngram_svm.txt
sh klas_vysledky.sh vystup_ngram_svm.txt

V tabulce 4.5 jsou k vidění časové hodnoty klasifikace knihovnou *libSVM* naměřené během klasifikace a spočítané skriptem *casove_vysledky.sh*.

Procentuální úspěšnost této metody je znázorněna v tabulce 4.9 a byla tvořena pomocí skriptu *klas_vysledky.sh*.

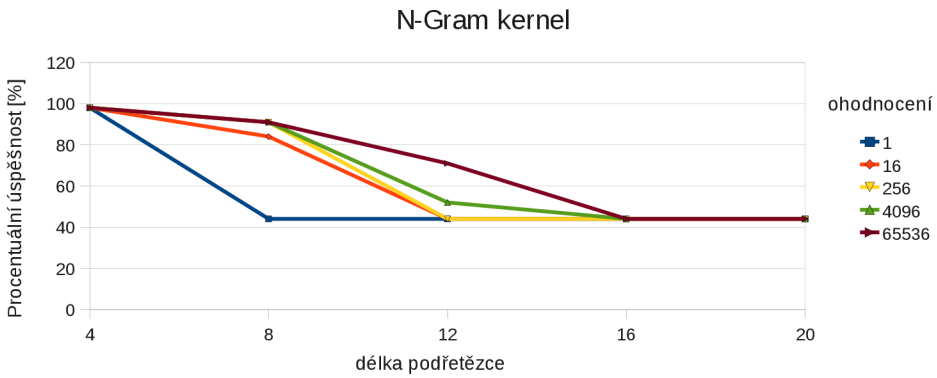
Pro grafické znázornění závislosti procentuální úspěšnosti klasifikace na změně délky podřetězce jsem vytvořil graf, který je uveden na obrázku 4.3.

Podřetězec	Čas	C parametr pro C-SVC 1	16	256	4096	65536
4	real	2m 28,106s	2m 28,076s	2m 28,190s	2m 28,210s	2m 28,428s
	user	2m 28,060s	2m 28,066s	2m 28,108s	2m 28,154s	2m 28,250s
	sys	8ms	10ms	12ms	10ms	6ms
8	real	3m 12,240s	3m 12,752s	3m 22,850s	3m 23,184s	3m 23,014s
	user	3m 12,168s	3m 12,570s	3m 22,826s	3m 22,998s	3m 22,984s
	sys	16ms	6ms	6ms	8ms	14ms
12	real	2m 37,218s	2m 42,150s	3m 22,968s	3m 27,440s	3m 32,148s
	user	2m 37,176s	2m 42,022s	3m 22,804s	3m 27,154s	3m 32,014s
	sys	8ms	8ms	0ms	6ms	4ms
16	real	2m 37,318s	2m 37,172s	2m 36,936s	2m 43,792s	3m 31,790s
	user	2m 37,180s	2m 37,144s	2m 36,888s	2m 43,762s	3m 31,736s
	sys	6ms	6ms	4ms	2ms	4ms
20	real	2m 36,878s	2m 37,128s	2m 36,938s	2m 36,978s	2m 36,990s
	user	2m 36,848s	2m 37,096s	2m 36,876s	2m 36,908s	2m 36,954s
	sys	4ms	4ms	12ms	16ms	14ms

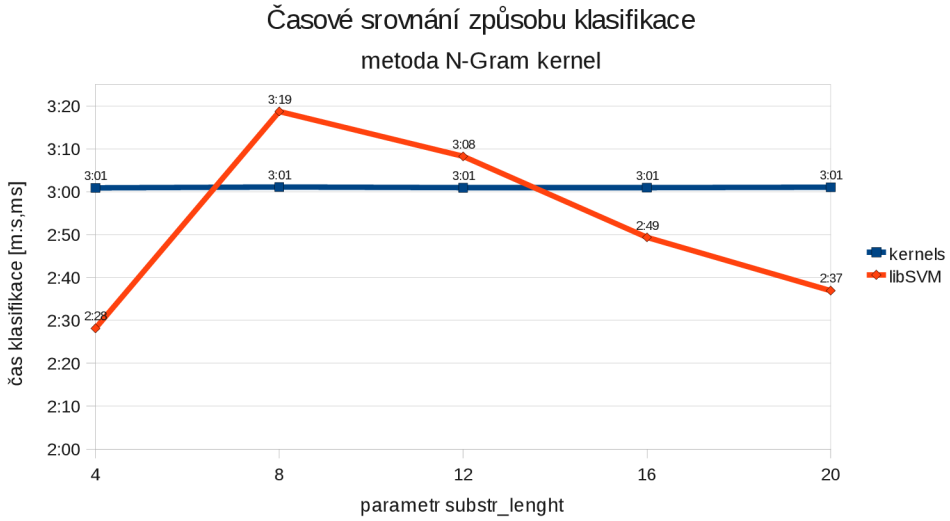
Tabulka 4.5: Celkový čas klasifikace prováděné programem *libSVM* a metodou *N-Gram kernel*.

Podřetězec	C parametr pro C-SVC 1	16	256	4096	65536
4	97,8	97,8	97,8	97,8	97,8
8	44,4	84,4	91,1	91,1	91,1
12	44,4	44,4	44,4	52,2	71,1
16	44,4	44,4	44,4	44,4	44,4
20	44,4	44,4	44,4	44,4	44,4

Tabulka 4.6: Procentuální úspěšnost[%] klasifikace knihovnou *libSVM* s využitím metody *N-Gram kernel*.



Obrázek 4.3: Graf závislosti procentuální úspěšnosti klasifikace na změně délky podřetězce.



Obrázek 4.4: Graf závislosti doby trvání klasifikace na změně délky podřetězce.

4.3 Experimenty metodou *Bag of Words kernel*

Řetězcová funkce *Bag of Words kernel* trvala ze všech řetězcových funkcí nejkratší dobu. V porovnání s ostatními řetězcovými metodami byly časové rozdíly poměrně obrovské. Klasifikace metodou *N-Gram kernel* byla, co se rychlosti týká, druhá nejrychlejší a i přesto byla metoda *Bag of Words kernel* přibližně 10krát rychlejší. V procentuální úspěšnosti už tato metoda tolik nevynikala a v pomyslném žebříčku obsadila až poslední místo mezi testovanými řetězcovými metodami. *Lambda* parametr nabýval hodnot 0.25, 0.5 a 0.75.

4.3.1 Klasifikace programem *kernels*

Pro klasifikaci programem *kernels* s využitím metody *Bag of Words kernel* byly připraveny skripty *bow_precomputed_run.sh* a *casove_vysledky.sh*. Skript *bow_precomputed_run.sh* generoval soubor *casy_bow_precomputed.txt*, který obsahoval časové údaje a byl zpracováván skriptem *casove_vysledky.sh*. Výstup z tohoto skriptu byl transformován do tabulky 4.7. Hodnoty z této tabulky jsou součtem dob trvání trénovací a testovací fáze.

Ukázka použití skriptů:

sh bow_precomputed_run.sh train00.txt test00.txt 2>casy_bow_precomputed.txt
sh casove_vysledky.sh casy_bow_precomputed.txt

Lambda	**Čas** real	user	sys
0.25	34,242s	26,794s	0,584s
0.5	33,872s	26,700s	0,526s
0.75	33,886s	26,910s	0,628s

Tabulka 4.7: Celkový čas klasifikace prováděné programem *kernels* s proměnným parametrem *lambda* využitím řetězcové metody *Bag of Words kernel*.

4.3.2 Klasifikace pomocí knihovny *libSVM*

V této části byly výstupem soubory *casy_bow_svm.txt* a *vystup_bow_svm.txt*. Skript *casove_vysledky.sh* zpracovával soubor *casy_bow_svm.txt* a výstupem byly hodnoty použité v tabulce 4.8. Procentuální výsledky klasifikace byly zpracovány skriptem *klas_vysledky.sh*, jenž vytiskl potřebné informace ze souboru *vystup_bow_svm.txt*. Výstupní soubory byly vytvořeny pomocí skriptu *bow_svm_run.sh*. Zde jsem kromě parametru *lambda* měnil i parametr C, který nabýval stejných hodnot jako v předešlých metodách.

Použití výše zmíněných skriptů je nýsledující:

sh bow_svm_run.sh train00.txt test00.txt 2>casy_bow_svm.txt > vystup_bow_svm.txt
sh casove_vysledky.sh casy_bow_svm.txt
sh klas_vysledky.sh vystup_bow_svm.txt

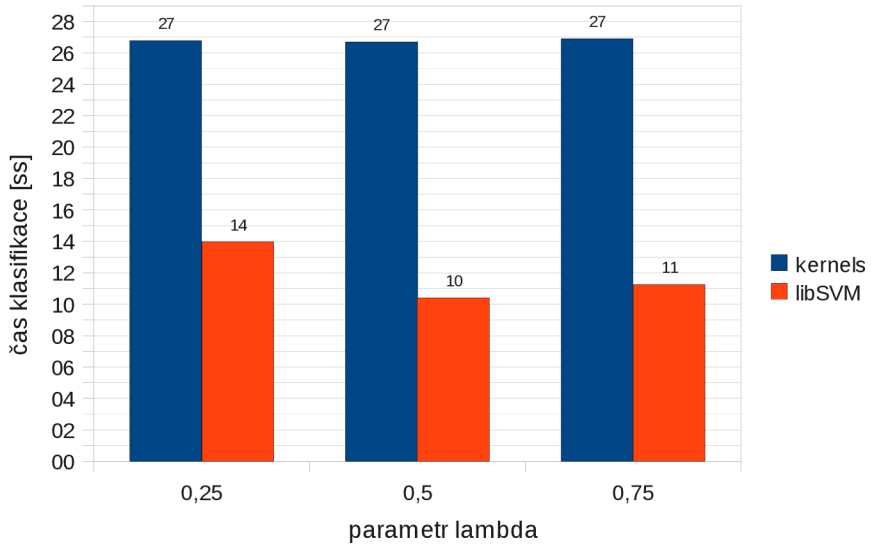
Pro tuto metodu jsem nevytvářel graf se zobrazením úspěšnosti klasifikace, protože naměřené hodnoty se prakticky nemění a graf by neměl žádnou vypovídací hodnotu. Místo grafu procentuální úspěšnosti jsem vytvořil graf, který znázorňuje rozdíly mezi klasifikací prováděnou programem *kernels* a knihovnou *libSVM*. V programu *svm-train* jsem měl pro každou hodnotu parametru C pět hodnot a tak jsem měření stejné hodnoty parametru *lambda* zprůměroval. Tento průměr jsem si mohl dovolit, protože parametr C nemá na dobu provádění žádný vliv.

		C parametr pro C-SVC				
Lambda	Čas	1	16	256	4096	65536
0.25	real	20,982s	14,968s	11,362s	11,244s	11,276s
	user	20,964s	14,974s	11,366s	11,248s	11,278s
	sys	2ms	6ms	2ms	6ms	2ms
0.5	real	11,266s	10,452s	10,340s	10,010s	9,904s
	user	11,262s	10,458s	10,344s	10,008s	9,908s
	sys	4ms	2ms	2ms	4ms	6ms
0.75	real	11,060s	11,102s	11,274s	11,390s	11,394s
	user	11,052s	11,106s	11,280s	11,400s	11,396s
	sys	8ms	4ms	0ms	0ms	6ms

Tabulka 4.8: Čas potřebný pro trénování i testování klasifikátoru metodou *Bag of Words kernel* a programem *libSVM*.

	C parametr pro C-SVC				
Lambda	1	16	256	4096	65536
0.25	27,8	27,8	27,8	27,8	27,8
0.5	27,8	27,8	44,4	44,4	44,4
0.75	27,8	27,8	27,8	27,8	27,8

Tabulka 4.9: Procentuální úspěšnost[%] klasifikace knihovnou *libSVM* s využitím metody *Bag of Words kernel*.



Obrázek 4.5: Porovnání časových průběhů obou způsobů klasifikace na metodě *Bag of Words kernel*.

4.4 Experimenty metodou *Edit distance kernel*

Poslední sérií experimentů byly experimenty pomocí řetězcové metody *Edit distance*. Pro tuto metodu jsem měnil parametr *lambda*, který nabýval hodnot 0.1, 0.01 a 0.001. Ke každé ze tří hodnot *lambda* jsem navíc měnil i parametr C. Jedná se o ohodnocení právě prováděného porovnání. Parametr C nabýval hodnot 1, 16, 256, 4096 a 65536 a byl použit až v programu *svm-train*.

4.4.1 Klasifikace programem *kernels*

Pro získání hodnot do tabulky 4.10 spouštím skript s názvem *leven_precomputed_run.sh*. Na standardní chybový výstup, který si zachytávám do souboru *casy_leven_precomputed.txt*, jsou vypisovány časové průběhy trénování a testování. Výpisy programu *kernels* k ničemu nepotřebuji, a tudíž standardní výstup není nikam přesměrován. Závěrečnou fází je výpočet tabulky 4.10 provedený pomocí skriptu *klas_vysledky.sh*, který má parametr příkazové řádky soubor s časovými měřeními.

Příklad použití:

sh leven_precomputed_run.sh 2>casy_leven_precomputed.txt
sh vypocet.sh casy_leven_precomputed.txt

4.4.2 Klasifikace pomocí knihovny *libSVM*

Stejně jako v předchozích částech provádím veškeré výpočty automaticky pomocí skriptů. Pro provedení klasifikace používám skripty *leven_svm_run.sh* a pro výpočet následujících tří tabulek skripty *casove_vysledky.sh* a *klas_vysledky.sh*, kterým si již zaznamenávám standardní výstup programů knihovny *libSVM*, protože tento výstup obsahuje procentuální

Lambda	Čas		
	real	user	sys
0.1	12m 31,660s	11m 11,180s	1m 12,680s
0.01	12m 17,916s	11m 06,942s	53,210s
0.001	12m 15,142s	11m 06,958s	1m 0,514s

Tabulka 4.10: Celkový čas klasifikace prováděné programem *kernels* s proměnným parametrem Lambda.

úspěšnost klasifikace.

Ukázka použití skriptů:

sh leven_svm_run.sh train00.txt test00.txt 2> casy_leven_svm.txt > vystup_leven_svm.txt
sh vysledky.sh casy_leven_svm.txt
sh uspesnost_klasifikace.sh vystup_leven_svm.txt

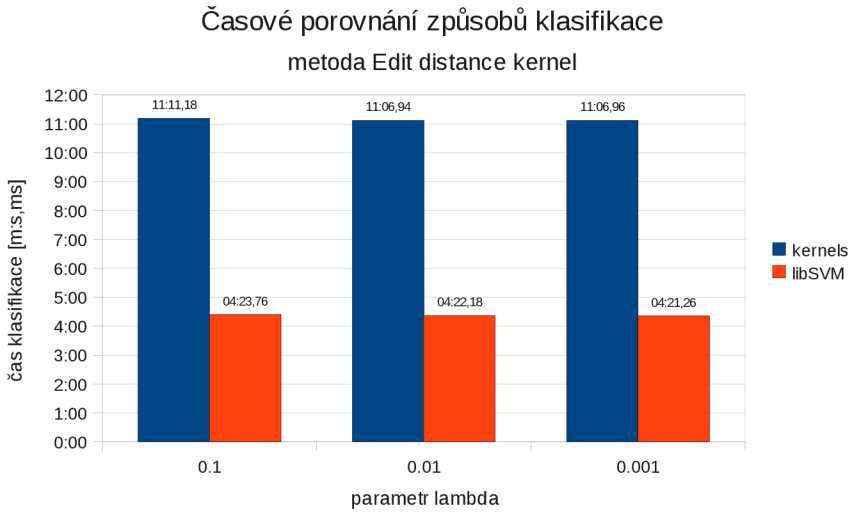
Celkový čas klasifikace pomocí knihovny *libSVM* je znázorněn v tabulce 4.11. Jak si můžeme povšimnout, tak celková doba je skoro 3krát rychlejší než klasifikace programem *kernels*.

Lambda	Čas	C parametr pro C-SVC				
		1	16	256	4096	65536
0.1	real	4m 24,880s	4m 24,794s	4m 23,332s	4m 25,824s	4m 23,730s
	user	4m 24,272s	4m 23,956s	4m 22,634s	4m 25,052s	4m 22,894s
	sys	594ms	692ms	674ms	722ms	684ms
0.01	real	4m 27,148s	4m 22,746s	4m 21,474s	4m 21,414s	4m 21,510s
	user	4m 26,348s	4m 22,080s	4m 20,856s	4m 20,778s	4m 20,846s
	sys	696ms	646ms	576ms	598ms	640ms
0.001	real	4m 23,564s	4m 21,328s	4m 21,288s	4m 21,536s	4m 21,742s
	user	4m 22,942s	4m 20,718s	4m 20,646s	4m 20,864s	4m 21,142s
	sys	580ms	562ms	610ms	624ms	582ms

Tabulka 4.11: Celkový čas klasifikace knihovnou *libSVM* s využitím metody *Edit distance kernel*.

Lambda	C parametr pro C-SVC				
	1	16	256	4096	65536
0.25	48,9	48,9	48,9	48,9	48,9
0.5	48,9	48,9	48,9	48,9	48,9
0.75	48,9	48,9	48,9	48,9	48,9

Tabulka 4.12: Procentuální úspěšnost[%] klasifikace knihovnou *libSVM* s využitím metody *Bag of Words kernel*.



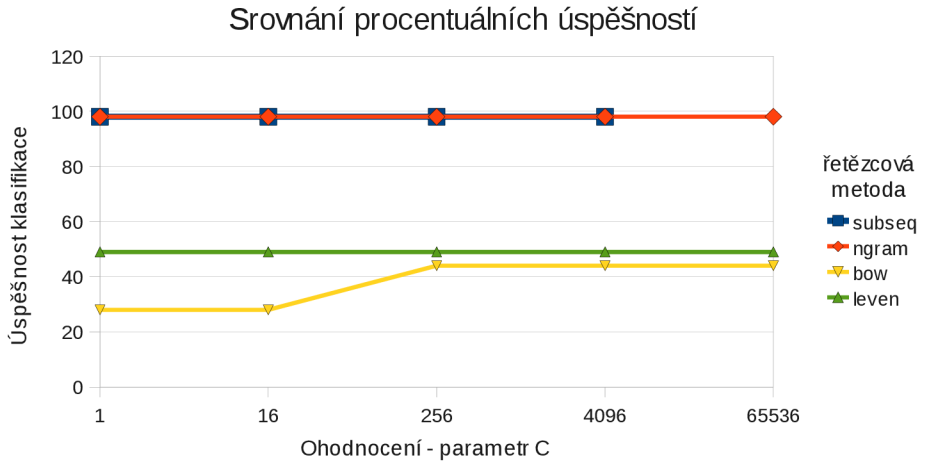
Obrázek 4.6: Porovnání dob trvání jednotlivých způsobů klasifikace pro metodu *Edit distance*.

Kapitola 5

Závěr

Úkolem této práce bylo přímo implementovat řetězcové funkce do knihovny *libSVM* a na provedených experimentech porovnat rychlost a úspěšnost klasifikace s předchozí implementací řetězcových funkcí v programu *kernels* [7].

Jelikož výše testované řetězcové metody mají málo společných parametrů, tak lze velmi těžko provádět nějaká bližší porovnání. Na obrázku 5.1 je znázorněno porovnání všech čtyř testovaných funkcí. V tomto porovnání není brána v potaz žádná konkrétní kombinace parametrů, ale je vynesena vždy ta nejlepší hodnota klasifikace pro každou metodu.

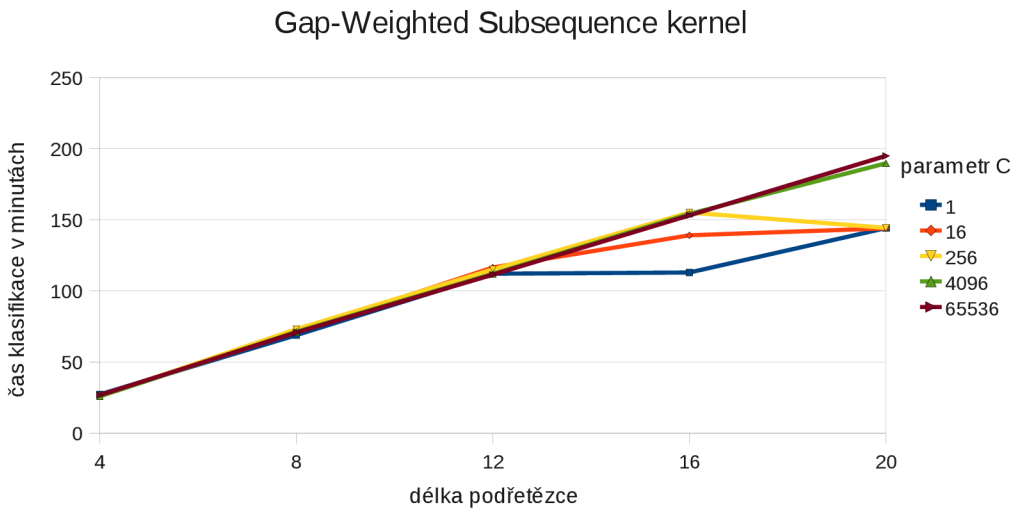


Obrázek 5.1: Graf nejlepších úspěšností všech řetězcových metod.

Jediné dvě metody, které lze mezi sebou detailněji porovnat jsou metody *Gap-Weighted Subsequence kernel* a *N-Gram kernel*, protože mají shodné hodnoty parametrů *substr_lenght*, *lambda* a *C*. Lépe z tohoto srovnání vychází metoda *N-Gram kernel*, která má sice nepatrně horší procentuální výsledky klasifikace, ale je oproti metodě *Gap-Weighted Subsequence kernel* několikanásobně rychlejší.

Časová závislost na některém parametru je nejlépe viditelná na metodě *Gap-Weighted Subsequence kernel*, kde se čas klasifikace zvyšoval se zvyšováním hodnot parametru *substr_lenght*. Tento jev je demonstrován v grafu na obrázku 5.2

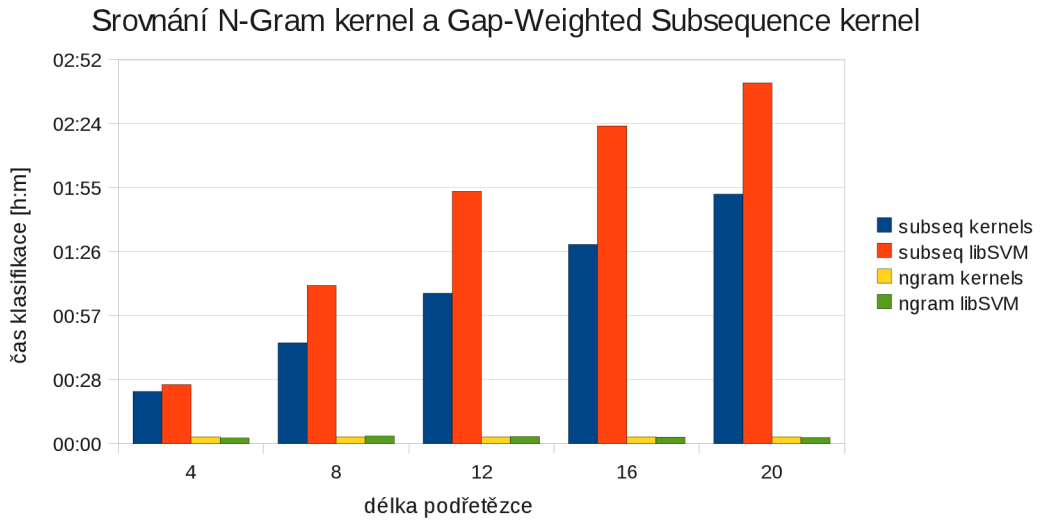
Naopak metoda *N-Gram kernel* se vyznačuje konstantní časovou náročností pro jakékoliv hodnoty parametrů. Doby jednotlivých průběhů se sice lišily, ale rozdíly byly nepatrné.



Obrázek 5.2: Vliv změny parametru *substr_lenght* na celkovou dobu trvání klasifikace.

I procentuální výsledky klasifikací jednotlivými metodami nejsou až tolik vypovídající. Všechny metody se nemusí ideálně hodit pro klasifikaci řetězcových dat, ale mohou vykazovat lepší výsledky v klasifikaci ostatních typů dat.

Ideálním kompromisem pro klasifikaci řetězcových dat je použití metody *N-Gram kernel* vyznačující se rychlým průběhem a dobrou úspěšností klasifikace, která v nejlepším případě pohybovala okolo 98%.



Obrázek 5.3: Porovnání dob trvání jednotlivých způsobů klasifikace.

Po analýze veškerých výsledků jsem zjistil, že implementace do knihovny *libSVM* je výhodnějším řešením než klasifikace přes *precomputed* matice programem *kernels* a to hned z několika důvodů.

Prvním důvodem je rychlost klasifikace. Klasifikace pomocí knihovny *libSVM* byla

oproti klasifikaci programem *kernels* pomalejší jen v několika málo případech. Výsledky jsou zobrazeny v grafem 4.2, 4.4, 4.5 a 4.6.

Dalším důvodem byl fakt, že pomocí knihovny *libSVM* můžeme vlastní průběh klasifikace ovlivnit několika dalšími parametry, jako v našem případě parametrem C. Možnosti ostatních parametrů knihovny *libSVM* získáme vypsáním nápovědy pro program *svm-predict* nebo prostudováním dokumentací či webových stránek [4].

Posledním důležitým faktorem je možnost použití *cross* validace, namísto klasifikace prováděné trénováním a testováním v samostatných bězích.

Literatura

[1] Aizerman, A.; Braverman, E. M.; Rozoner, L. I.: *Theoretical foundations of the potential function method in pattern recognition learning, Automation and Remote Control.* pp. 821-837,vol. 25, 1964.

[2] Burget, L.: *Klasifikace a rozpoznávání (kurs).* FIT na VUT v Brně, Brno, CZ, 2010.

[3] Burget, L.: *Lineární klasifikátory (přednáška).* FIT na VUT v Brně, Brno, CZ, 8.3.2010, předmět IKR - Klasifikace a rozpoznávání.

[4] Chang, C.-C.; Lin, C.-J.: *LIBSVM: a library for support vector machines.* 2001, software available at http://www.csie.ntu.edu.tw/~cjlin/libsvm.

[5] Jelínek, L.: *Vývoj jádra I. - Linux E X P R E S.* [Online; navštíveno 16. 05. 2010], URL: http://www.linuxexpres.cz/praxe/vyvoj-jadra-i.

[6] Kůrková, V.: *Strojové učení se schopností generalizace.* [Online; navštíveno 17. 04. 2010] URL: http://hilbert.chtf.stuba.sk/KUZV/download/kuzv-kurkova.pdf.

[7] Michlovský, Z.: *High-speed Systems for Identification of Attacks and Malware.* 2009, pojednání o disertační práci.

[8] Michlovský, Z.; Pang, S.; Kasabov, N.; aj.: *String Kernel Based SVM for Internet Security Implementation.* Springer Berlin Heidelberg, 2009, ISBN 978-3-642-10682-8.

[9] Shawe-Taylor, J.; Cristianini, N.: *Kernel Methods for Pattern Analysis.* Cambridge University Press, New York, NY, USA, 2004.

[10] Vapnik, V. N.: *The nature of statistical learning.* Springer-Verlag New York, Inc., 1995.

[11] Wikipedia: Kernel trick — Wikipedia, The Free Encyclopedia. 2010, [Online; accessed 16-May-2010].
URL http://en.wikipedia.org/w/index.php?title=Kernel_trick

[12] Wikipedia: *Semi-supervised learning — Wikipedia, The Free Encyclopedia.* 2010, [Online; accessed 16-May-2010].
URL http://en.wikipedia.org/w/index.php?title=Semi-supervised_learning

[13] Wikipedia: *Support vector machine — Wikipedia, The Free Encyclopedia.* 2010, [Online; accessed 16-May-2010].
URL http://en.wikipedia.org/w/index.php?title=Support_vector_machine

[14] Wikipedie: Strojové učení — Wikipedie: Otevřená encyklopedie. 2010, [Online; navštíveno 16. 05. 2010].
URL http://cs.wikipedia.org/w/index.php?title=Strojov%C3%A9_u%C4%8Den%C3%AD&oldid=5195739

[15] Zbořil, F.; Zbořil, F.: *Základy umělé inteligence, studijní opora*. FIT na VUT v Brne, Brno, CZ, 2007.

Dodatek A

Obsah CD

Na přiloženém CD jsou v adresářové struktuře, uvedené níže, uloženy veškeré soubory, které jsem potřeboval v této práci. Mimo programu *kernels* a knihovny *libSVM* jsou na médiu přiloženy i soubory se vstupními daty a skripty zajišťující průběh vlastní klasifikace.

Adresářová struktura přiloženého CD je následující:

```
CD
 |---- casy
 |        |-- README
 |        |-- main.cpp
 |        '-- Makefile
 |---- kernels
 |          |-- README
 |          |-- zdrojové soubory
 |          |-- skripty
 |          '-- soubory s výsledky
 '---- libsvm
           |-- README
           |-- zdrojové soubory
           |-- skripty
           '-- soubory s výsledky
```

Dále jsou na médiu v jednotlivých adresářích výstupní soubory, které byly vytvořeny samotnými programy *kernels* a *libSVM*. Tyto soubory slouží pro demonstraci činnosti všech skriptů. Druhým důvodem přiložení těchto souboru je i fakt, že jejich vytváření trvá značnou dobu.

Každý adresář obsahuje soubor *README*, ve kterém jsou podrobně popsány použití veškerých programů, skriptů a jsou v něm vysvětleny významy souborů, které byly vytvořeny výše zmíněnými skripty.