VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

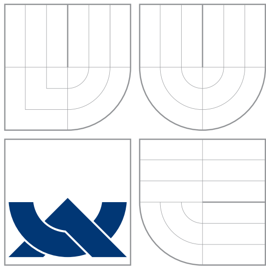
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

ROZPOZNÁVÁNÍ RUČNĚ PSANÝCH ČÍSLIC POMOCÍ SUPPORT VECTOR MACHINES

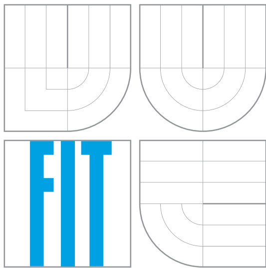
BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE JOZEF HRICKO
AUTHOR

BRNO 2010



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

ROZPOZNÁVÁNÍ RUČNĚ PSANÝCH ČÍSLIC POMOCÍ SUPPORT VECTOR MACHINES

HANDWRITTEN DIGITS RECOGNITION USING SUPPORT VECTOR MACHINES

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — JOZEF HRICKO
AUTHOR

VEDOUCÍ PRÁCE — Ing. OLDŘICH PLCHOT
SUPERVISOR

BRNO 2010

Abstrakt

Práce se zabývá možností rozpoznávání ručně psaných číslic a znaků pomoci volně dostupných knihoven. Pro rozpoznávání je použitá jádrová klasifikační metoda support vector machines. Práce také uvažuje různé algoritmy zpracování obrazu a jejich implementace. Dále je zde navrhnuto, jak je možno aplikaci vytvořit co nejefektivněji vzhledem ke znovupoužitelnosti zdrojového kódu.

Abstract

Thesis deals with the options of the hand-written digit and character recognition using open-source libraries. The kernel-based classifiers (support vector machines) are used for the recognition. Various algorithms of image processing and their implementation are shown in this work together with suggestions, how to effectively write reusable source code.

Klíčová slova

rozpoznávání textu, podpůrné vektory, OCR, zpracování obrazu, extrakce příznaků, grafické uživatelské rozhraní, lineární klasifikátor

Keywords

pattern recognition, support vector machines, OCR, image processing, feature extraction, graphical user interface, linear classifier

Citace

Jozef Hricko: Rozpoznávání ručně psaných číslic pomocí support vector machines, bakalářská práce, Brno, FIT VUT v Brně, 2010

Rozpoznávání ručně psaných číslic pomocí support vector machines

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Oldřicha Plchota. Další informace mi poskytl Ing. Michal Hradiš. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

....
Jozef Hricko
14. května 2010

Poděkování

Především bych chtěl poděkovat panu Ing. Oldřichu Plchotovi za jeho odborné rady a názory. Dále panu Ing. Michalu Španělovi, který mi umožnil na jeho přednášce rozdat formuláře pro vyplnění. A v neposlední řadě chci poděkovat své rodině, která mi byla psychickou oporou.

© Jozef Hricko, 2010.
Tato práce vznikla jako školní dílo na Vysokém učení technickém v Brně, Fakultě informačních technologií. Práce je chráněna autorským zákonem a její užití bez udělení oprávnění autorem je nezákonné, s výjimkou zákonem definovaných případů.

Obsah

1 Úvod **3**

2 Teoretická část **4**
2.1 Support vector machines . . . 4
2.1.1 Korigující míry . . . 6
2.1.2 Lineárně neseparovatelná úloha . . . 6
2.2 OCR . . . 7
2.3 Algoritmy při pracování obrazu . . . 7
2.3.1 Skeletonizace . . . 8
2.3.2 Prahování . . . 8
2.3.3 Eroze a dilatace . . . 9
2.3.4 LIBSVM formát . . . 9

3 Návrh řešení **10**
3.1 Databáze . . . 10
3.2 GUI . . . 10
3.3 Knihovny SVM . . . 11
3.4 Zpracování formulářů . . . 12

4 Extrakce příznaků a klasifikace **14**
4.1 Příprava dat . . . 14
4.2 Hledání příznaků . . . 15
4.2.1 Paprsky . . . 15
4.2.2 Hodnoty pixelů . . . 16
4.3 Klasifikace . . . 16

5 Implementace **17**
5.1 OpenCV . . . 17
5.1.1 Použité struktury a funkce . . . 18
5.1.2 Zpracování formulářů . . . 18
5.2 LIBSVM . . . 18
5.3 Qt framework . . . 19
5.4 Další informace . . . 20

6 Testování **21**
6.1 Nároky na hardware . . . 21
6.2 Úspěšnost rozpoznávání . . . 23
6.3 Ukázka v praxi . . . 24

7 Závěr **26**

A Obsah CD **29**

B Manuál **30**
B.1 instalace 30
B.2 Ovládání 31

C Formulář **33**

Kapitola 1

Úvod

Snažíme se naučit počítač vidět svět očima člověka už od samého počátku vzniku počítačů. Je to dáno tím, že společnost má snahu vyvíjet prostředky pro ulehčení práce, a počítače jsou velice vhodným kandidátem. V první řadě je to jeho dostupnost - ve většině evropských zemí jsou personální počítače rozšířené nejen ve sféře obchodu a telekomunikace, ale i ve většině domácností. Například Policie ČR [4] by se bez počítačového vidění jen těžko obešla při odhalování zločinu. Odhalování silničních pirátů pomocí státní poznávací značky a také pokud možno obličeje řidiče lze provést automaticky velice dobře právě díky výpočetní technice. Další případ počítačového vidění v kriminalistice je daktyloskopie [15]. Také v oblasti průmyslu se kontroluje velikost nebo tvar výrobků pomocí strojů [13].

Zkoumat počítačové vidění a klasifikační postupy je jistě přínosné i pro rozvoj v robotice. Podle [7] je vyvíjen superpočítač Neurogrid, který je schpen učit se sám od sebe. Většina běžných počítačů na stejný matematický problém získá stejný výsledek. Neurogrid výsledků ale nabídne více, i když mohou být chybné. Roboti se tedy budou moci v budoucnosti sami zdokonalovat tak, že se automaticky poučí ze své předchozí chyby, podobně jako člověk. To povede k individualitě počítačů. Roboti budou schopni vidět svět očima člověka, pravděpodobně však dokonaleji. Rozvoj v této vědecké oblasti proto považuji za velmi důležitý a klasifikace vzorů zde hraje velice důležitou roli.

Tato kapitola uvádí čtenáře do oblasti počítačového vidění, a také vysvětluje, proč jsem si zvolil tuto práci pro zpracování. V kapitole 2 budou objasněny některé postupy a metody, které jsou při manipulaci s obrazem využívány. Také je zde vysvětlen princip support vector machines, který je pro tuto práci zásadní. Kapitola 3 popisuje postupy, které byly uvažovány při vývoje aplikace. Kapitola 4 vysvětluje, jakým způsobem jsou převáděny obrazové materiály pro klasifikátor. Kapitola 5 objasňuje, jak byly algoritmy a návrhy přeneseny do zvoleného programovacího jazyka. Kapitola 6 testuje různé nastavení aplikace a odhaluje chyby implementace. Také jsou zde již uvedené výsledky rozpoznávání. Závěrečná kapitola 7 shrnuje celkový průběh vývoje projektu a zmiňuje možné vylepšení.

Kapitola 2

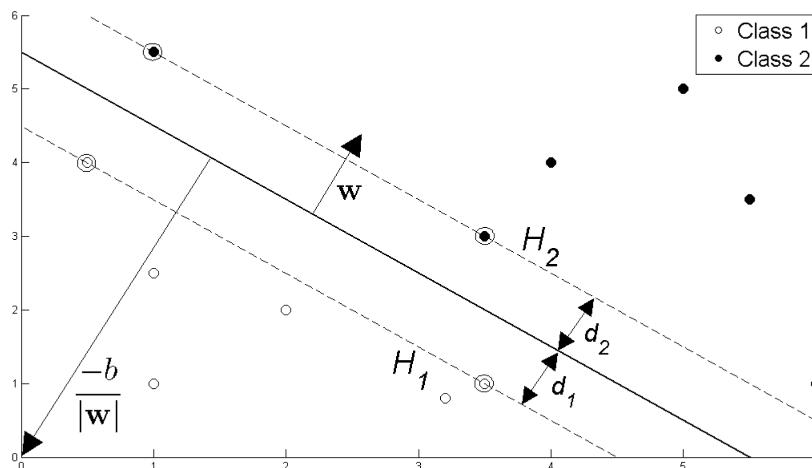
Teoretická část

Tato kapitola se zabývá potřebnou teorií pro vývoj aplikace. Dále jsou vysvětleny některé důležité pojmy, které mohou být zásadní pro pochopení obsahu dalších kapitol.

2.1 Support vector machines

Informace v této kapitole jsou volně převzaty z [13], [16], [20] a [1].

Podle zadání je pro klasifikaci doporučeno použít metodu support vector machines. Support vector machines je jádrová klasifikační metoda, která hledá optimální rozhodovací linii mezi jednotlivými třídami dat. Lineární jádrovou funkci pro tento typ učení představil světu Vladimir Vapnik v r. 1963 [17], nelineární až v r. 1992. Ve své základní verzi pracuje pouze se dvěma třídami, kde jedna třída je pozitivní a druhá negativní. Tyto třídy by navíc měly být lineárně separovatelné a nepřekrývající se[1].

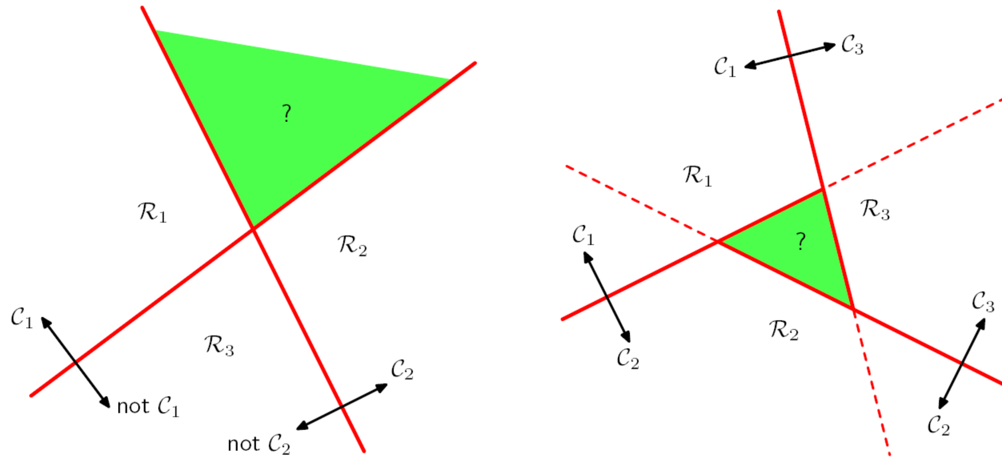


Obrázek 2.1: Optimální nalezená separační linie. Převzato z [5].

Pokud máme problém, kde se vyskytuje N tříd, kde $N > 2$, máme dvě možnosti řešení. Klasifikace do N tříd lze dosáhnout tak, že postupně natrénujeme pro každou třídu sa-

[1] Ve skutečnosti tomu tak většinou není.

mostatný klasifikátor takovým způsobem, že se data dané třídy postaví proti všem datům z ostatních tříd (jeden proti všem). Druhou možností je natrénovat klasifikátor pro každou dvojici tříd (každý s každým).



Obrázek 2.2: Demonstrace řešení problému více tříd. Vlevo je postavena třída proti všem ostatním. Vpravo je použita metoda každý s každým. Převzato z [13].

Support vectors, neboli podpůrné vektory, jsou prvky nacházející se nejblíže rozdělovací nadrovině. Pomocí jádrové funkce proto tyto body hledáme, abychom mohli tuto oblast vymezit. Jednoduše řečeno jde o maximalizaci mezery mezi trénovacími daty. Ostatní body pro nás následně již nehrají roli, což je na rozdíl od většiny ostatních klasifikátorů velice efektivní i při opravdu velkém počtu vstupních dat.

Předpokládejme, že máme body:

$$\{(\langle x_1 \rangle, y_1), (\langle x_2 \rangle, y_2), \ldots, (\langle x_n \rangle, y_n)\} \tag{2.1}$$

kde$\langle x_i \rangle$ je vstupní vektor a y_i může nabývat hodnot 1, nebo -1.

Lineární funkce je zadána rovnicí:

$$y(\mathbf{x}) = \mathbf{w}^T \phi(\mathbf{x}) + b \tag{2.2}$$

Kde $\phi(\mathbf{x})$ představuje transformaci příznakového prostoru. Parametr b je nazýván biasem.

Podstatou této úlohy je ve výsledku maximalizovat kritérium

$$\text{argmin} \frac{1}{2} ||\mathbf{w}||^2 \tag{2.3}$$

Přepis do tvaru Lagrangeových násobitelů:

$$\max \sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i,j} \alpha_i \alpha_j c_i c_j \mathbf{x}_i^T \mathbf{x}_j \tag{2.4}$$

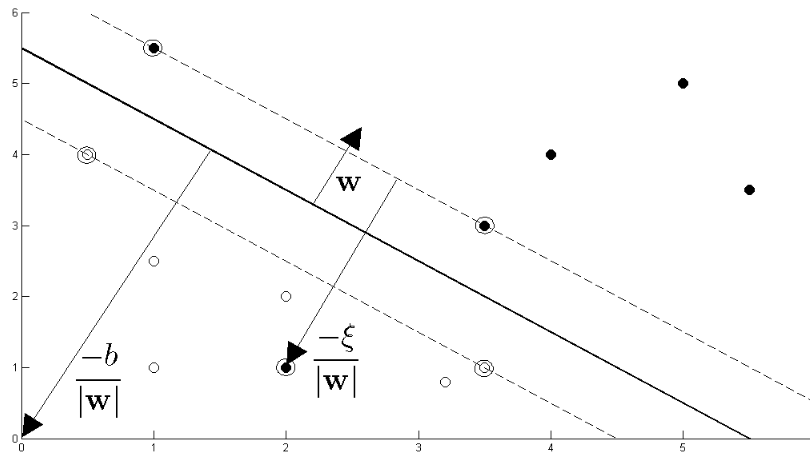
, kde $\alpha_i \geq 0$ a $\sum_{i=1}^{n} \alpha_i c_i = 0$

2.1.1 Korigující míry

Data však větnou nejsou lineárně separovatelná. Proto existuje tolerance pomocí korigujících mír (slack variables). Minimalizuje se tedy:

$$\frac{1}{2}||\mathbf{w}||^2 + C \cdot \sum_i \xi_i \tag{2.5}$$

ξ_i představuje korigující míru. C rozumíme konstantu určující velikost nevhodnosti špatně umístěných dat.



Obrázek 2.3: Korigující míry umožňují ignorovat neseparovatelnost dat v prostoru. Převzato z [5]

2.1.2 Lineárně neseparovatelná úloha

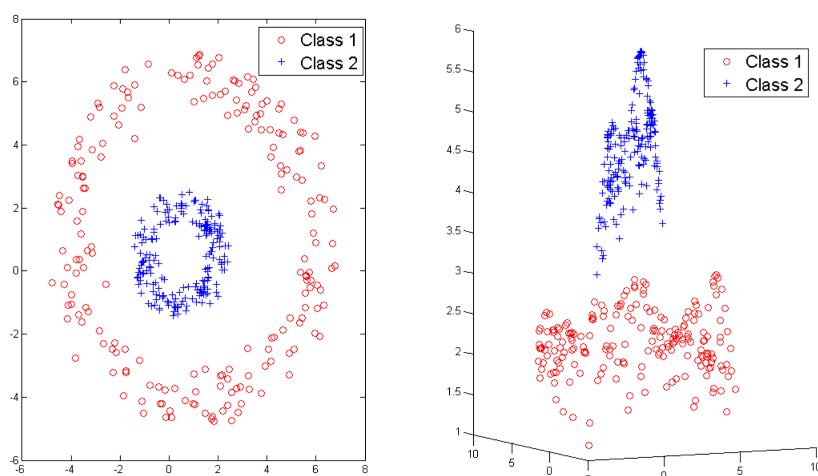
Pokud se jedná o složitější problém (lineárně neseparovatelný), jádrová funkce musí převést data do jiného prostoru (někdy se i zvýší dimenzionalita), kde budou lineárně separovatelná.

Tomu se říká jádrový trik (kernel trick). Stačí tedy znát skalární součin dat v daném prostoru. Tento součin počítá právě jádro. Pokud se použije vhodné jádro, budou data lineárně separovatelná.

V SVM se jádro (kernel) označuje: $K(x_i, x_j)$.

Rozlišuje se více typů jádrových funkcí:

- lineární
- polynomiální
- radiální
- sigmoidní



Obrázek 2.4: Transformace prostoru tak, aby bylo možno data separovat. Zde byla použita radiální transformace. Převzato z [5].

2.2 OCR

Pod termínem OCR (Optical Character Recognition) si lze představit soubor metod, které dokáží z textu získaného jakýmkoliv pořizovacím zařízením (tedy např. rastrový obrázek pořízený scannerem, fotografie, video) získat textový výstup.

Bohužel, počítačové vidění nikdy nebude na takové úrovni, aby byly rozponány veškeré znaky na 100%. Ovšem nejdokonalejší systémy dosahují úspěšnosti až 99% [12], záleží však na kvalitě i množství vstupních dat, množství přítomného šumu, a také na klasifikátoru.

Existují dva druhy OCR, a to **offline** a **online**. Liší se pouze tím, že offline rozpoznávání se děje až poté, co byly znaky vytištěny nebo napsány, kdežto online rozpoznávání pracuje tak, že kontroluje směry tahů autora přímo při psaní[2]. My se budeme zabývat offline typem OCR.

2.3 Algoritmy při pracování obrazu

Tato kapitola se věnuje velmi důležitému procesu pro další zpracovávání vstupních dat. Především se jedná o metody pro odstranění šumu a disharmonií které nejsou pro další průběh data miningu žádoucí. Důležité jsou především thresholding (z důvodu různého přítlaku pera), převod na stupně šedi (z důvodu různé barvy pera) a skeletonizace (z důvodu různé šířky pera).

Ať už potřebujeme rozpoznávat jakékoliv objekty v jakémkoliv obrazu, potřebujeme před samotnou klasifikací provést několik shodných kroků. Ty jsou podle [8] a [4]:

1. **Snímání a uložení obrazu(digitalizace)**: Tento krok se provádí většinou scannerem, fotoaparátem, nebo kamerou. Jde o to zachytit skutečnost do obrazu a uložit ji v digitální podobě.

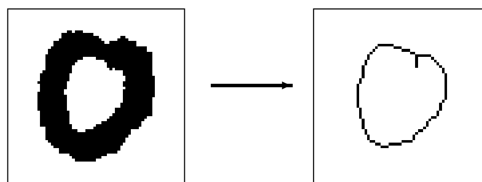
[2]Proto bude pořizování takovéto databáze také o něco obtížnější.

2. **Předzpracování**: Pro další operace s obrazovým materiálem je potřeba zbavit jej šumu, ořezat nepotřebné informace, případně jinak filtrovat nadbytečná data.
3. **Segmentace obrazu na objekty**: V této části hledáme v již předzpracovaném obraze důležité části. Nejčastěji to je objekt který nás zajímá, a jeho pozadí.
4. **Popis objektů**: V tomto kroku již hledáme detaily prvků a jejich příznaky.
5. **Porozumnění obsahu okna**: Tím se rozumí již samotná predikce.

2.3.1 Skeletonizace

Skeleton objektu bývá časo využíván při zisku příznaků. Využívá jej např. algoritmus pro porovnávání otisků prstů [4]. Skeletem objektu je tedy linie o šířce jednoho pixelu. K nalezení tohoto skeletu se používá nejčastěji metoda ztenčování (např *zhangův*, nebo *deutschův* algoritmus). Někdy se také používá metoda *chordal axis*, která je přesnější, zato však složitější na implementaci.

Podle [4] je velice přesný právě *deutschův* algoritmus, který jsem pro aplikaci použil. Ten funguje tak, že pro celý obraz se zkontroluje okolí bodu. Podle toho se vytvoří maska 3×3, kterou zkontrolujeme pomocí tabulky masek. Podle této tabulky poté zjistíme, zda bude bod invertován, nebo ne. Abychom docílili toho, že je skelet široký maximálně jeden pixel, je potřeba obvykle více průchodů. Deutschova tabulka masek je konstruována tak, že se obraz prochází dvakrát v jednom cyklu. Je to z toho důvodu, aby čára o šířce dvou pixelů nebyla zcela odstraněna.



Obrázek 2.5: Výsledek skeletonizace

2.3.2 Prahování

Prahování, neboli thresholding, je přiřazování výslednému obrazu hodnoty buďto 0, nebo 1. Rozhodování se děje podle předem známého prahu (threshold) T. Podle [8], prahování je děleno dále na lokální a globální.

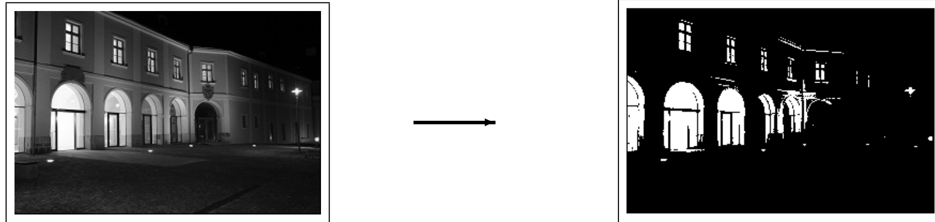
U globálního prahování je práh T dán předem a je neměnný. aplikuje se tedy na celý obraz g.

$$g(x, y) = \begin{cases} 1 & \text{pro } f(x, y) \geq T \\ 0 & \text{pro } f(x, y) < T \end{cases} \tag{2.6}$$

Lokální prahování se využívá v případě nerovnoměrného osvětlení obrazu. Základem lokálního prahování je postupně prozkoumat okolí každého bodu v obraze tak, abychom

pomocí tohoto okolí zvolili hodnotu práhu T pro tento bod. Velikost masky okolí si můžeme zvolit sami, nebo ji odhadujeme metodou *p-procentní prahování* [4].

Pro naši práci budeme používat pouze globální prahování, především pro zpracování formulářů.



Obrázek 2.6: Výsledek globálního prahování. Hodnota práhu T byla zvolena 128. Původní obrázek převzat z [19].

2.3.3 Eroze a dilatace

Eroze redukuje světlé objekty, dilatace zvětšuje objekty a zaplňuje díry.

Obrázek 2.7: Vlevo je demonstrována dilatace, vpravo eroze. Původní obrázek je umístěn uprostřed. Obrázek převzat z [18].

2.3.4 LIBSVM formát

Jelikož bude aplikace používat knihovnu LIBSVM (více v kapitole 3.3), je potřeba objasnit formát, který tato knihovna zpracovává. LIBSVM používá pro trénování tzv. sparse formát [6]. Jedná se o vypuštění nulových hodnot na vstupu. Příjimači tedy stačí pouze informace o pozicích nenulových hodnot[3].

Pokud tedy máme řadu čísel

```
7 0 6 0 0 0 0 2 0
```

převod do sparse formátu bude vypadat takto:

```
1:7 3:6 8:2
```

Výhoda se jeví především u řady čísel, kde je mnoho nulových hodnot. Upravený řetězec bude tedy podstatně kratší, než původní.

[3]Na ostatních pozicích se tedy budou předpokládat nuly.

Kapitola 3

Návrh řešení

Po domluvě s vedoucím práce jsem se rozhodl, že aplikace bude schopna rozpoznávat nejen číslice, ale i písmena. V této kapitole jsou podávány různé návrhy pro řešení podproblémů projektu.

3.1 Databáze

Součástí semestrálního projektu bylo obstarat vhodnou databázi číslic a písmen. V příloze je ukázka vyplňeného formuláře C.1. Existuje 10 různých verzí takovýchto formulářů, pro ukázku však stačí jeden. Po domluvě s konzultantem měly formuláře také obsahovat slova, aby byly znaky napsány co nejpřirozeněji. Dále měly formuláře mít v každém rohu černý čtverec pro automatické zarovnání formuláře. Bylo vyplněno celkem 258 formulářů různými osobami různého pohlaví. Vyplňující osoby byli studenti, tedy lidé jedné věkové kategorie, přesto není databáze nijak jednotvárná. To je vzhledem k povaze této práce důležité[1].

Velice obsáhlou databází číslic je databáze **MNIST** [11]. MNIST je rozšířením původní NIST databáze. Trénovací sada obsahuje 60000 a testovací 10000 číslic. Přestože tato databáze je velice kvalitně zpracovaná (normalizovaná), bohužel neobsahuje znaky písmen, proto jsem ji při trénování nezahrnul. Navíc vhodnou sadu již mám díky vyplněných formulářů k dispozici (celkem 41749 znaků).

I když sada není implicitně pro trénování zahrnuta, je důležité mít referenční trénovací a testovací sadu, proto jsem otestoval chybovost použitého klasifikátoru i na databázi MNIST. Výsledky testování jsou uvedeny v kapitole 6.2.

3.2 GUI

V dnešní době je vytváření rozsáhlejších programů bez grafického rozhraní velice neprofesionální. V první řadě je to dáno tím, že počítač již ovládá široká veřejnost, která nezvládne efektivně ovládat aplikaci pomocí terminálu, tedy příkazového řádku. Dalším důvodem vytvoření GUI pro tento projekt je ten, že se bude pracovat s obrazovým materiálem, přestože výstup programu je v textové podobě. Implementovat grafické rozhraní by přesto neměl být problém především díky vhodné literatuře [14]. Rozhodl jsem se proto vytvořit výslednou aplikaci tak, aby byla použitelná pro nejširší spektrum uživatelů. Jedním z předpokladů je také možnost napsat do určeného okna slovo pomocí myši.

[1] Přestože by znaky měly být různorodé, neměly by být vadné.

Rozhodoval jsem se mezi dvěma toolkity, které by splňovaly několik zásadních požadavků. Knihovna musí být multiplatformní (především Winodws a Linux/UNIX). Knihovna by měla být určená primárně pro jazyk C++, který podle mého názoru poskytuje nejširší spektrum vnitřních struktur, je objektově orientovaný, a navíc výsledné programy jsou relativně rychlé při běhu. Do konečného výběru se tedy dostaly pouze **wxWigdets** a **Qt**.

wxWidgets mé sympatie získal především proto, že se jedná o multiplatformní framework. Bohužel mě však zklamal v ne příliš jednotném zobrazení pod různými platformami. Zdrojový kód se zdá ve výsledku dost nepřehledný a podpora také není dostatečná. Vývoj tohoto projektu začal v r. 1992. Od té doby prošel mnoha změnami, bohužel se zdá, že jeho sláva upadá a wxWidgets upadne brzy v zapomění[2].

Qt má vývojové prostředí od verze 4.5 – *QtCreator*. QtCreator přes své nedostatky především při práci se signály dokáže vývojářům celkem usnadnit vývoj aplikací. QtCreator však není příliš intuitivní, navíc trvá poněkud déle pochopit některé funkce a nastavení. Také by bylo vhodné, kdyby byl programový soubor *pro*[3] nastavitelný přes tento framework poněkud pohodlněji.

Nakonec jsem se rozhodnul pro knihovnu **Qt**. Predevším z toho důvodu, že tento toolkit má zcela jistě perspektivnější budoucnost narozdíl od wxWidgets.

3.3 Knihovny SVM

Existuje několik knihoven pro klasifikaci, které používají ve svém jádru metodu support vector machines. Po doporučení jsem zvolil knihovnu LIBSVM [3], přesto by bylo vhodné uvést alternativní knihovny.

- SVM-light
- SVMTorch
- SVM-Struct
- mySVM

Velice rozsáhý seznam knihoven lze nalézt také zde [9].

Knihovna LIBSVM je velice dobře nastavitelná. Pro trénování je možno použít různé parametry podle potřeby. Jelikož je možné tyto parametry v rámci aplikace měnit pomocí nastavení, je vhodné tyto parametry popsat.

- **-s** : nastavuje typ support vector machines
  0 – C-SVC
  1 – nu-SVC
  2 – one-class SVM
  3 – epsilon-SVR
  4 – nu-SVR

[2]Především soudě dle počtu aktualit na oficiálních internetových stránkách a počtu vydaných aktualizací pro knihovnu.

[3]Tento soubor obsahuje veškeré informace o způsobu kompilace aplikace, umístění externích knihoven a souborů, konfigurace parametrů (flags) při překladu, výčet zdrojových souborů atd.

- **-t** : nastavuje typ jádrové funkce
  0 – lineární: $u' * v$
  1 – polynomiální: $(gamma * u' * v + coef0)^{degree}$
  2 – radiální: $e^{-gamma*|u-v|^2}$
  3 – sigmoid: $tanh(gamma * u' * v + coef0)$
  4 – vlastní jádrová funkce
- **-g** gamma : nastavuje parametr γ (pokud nenastaveno, tak 1/k)
- **-c** cost : nastavuje korigující míry **C** pro penalizaci. Platí pouze pro C-SVC, epsilon-SVR, and nu-SVR
- **-m** cachesize : Horní hranice velikosti paměti v MB, kterou si LIBSVM může propůjčit ze systémvých prostředků.
- **-b** pravděpodobnostní výstup. Volby jsou buďto 1, nebo 0

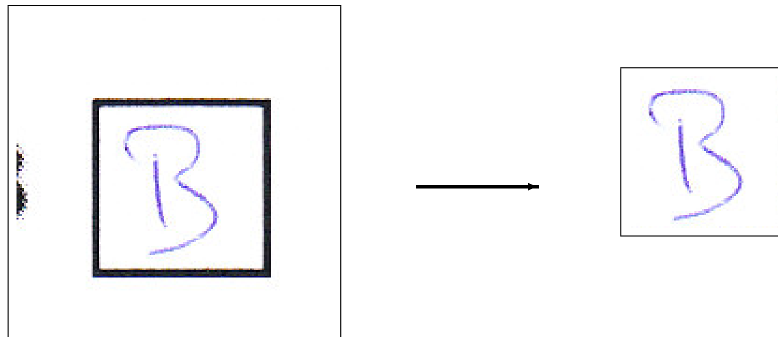
3.4 Zpracování formulářů

Vzhledem k tomu, že je několik verzí formulářů, je potřeba nejprve zjistit aktuální verzi. Ta je zjišťována tak, že pomocí souboru *look.dat* vyhledá informace o umístění černobílého obdélníku. Ten je poté prahován a rozdělí se na čtvrtiny. Pomocí funkce 5.1 se nakonec zjistí světelnost každé této části. Každá část má přidělenou váhu ω_i. Levá horní část má $\omega_1 = 2^1$, pravá horní $\omega_2 = 2^2$, levá dolní $\omega_3 = 2^3$ a pravá dolní $\omega_4 = 2^4$. Poté se sečtou váhy ω_i částí, kde byla světelnost menší, než práh $T = 200$. Výsledá hodota nám udává číslo vyšší verze (protoře jeden formulář obsahuje dvě verze, nižší verze je jednoduše o 1 menší).

Obrázek 3.1: Rozlišení formulářů do verzí. Tento formulář má verzi $2^2 + 2^3$ a $2^2 + 2^3 - 1$, tedy 12 a 11.

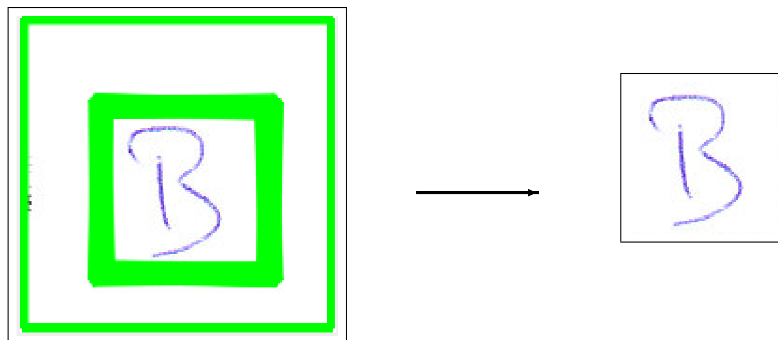
Poté, co známe verzi formuláře, můžeme postpně procházet formulář podle předem známých souřadnic. Tyto souřadnice se budou nacházet v souboru *look.dat*. Na těch se bude přibližně nacházet pro nás důležitý znak. Pokud bude funkce na pozici formuláře, kde se očekává slovo, funkce se podívá do pomocného souboru, kde jsou informace o aktuální verzi formuláře a jeho rozvržení. Všechny tyto soubory (jak look.dat, tak pomocné soubory) budou umístěny ve složce *DATA*, která nesmí být smazána.

Nyní, když víme, jaký znak by se měl v uvedené oblasti nacházet, musíme tento znak ořezat tak, aby neobsahoval žádná nadbytečná data. Tento krok popisuje následující obrázek:



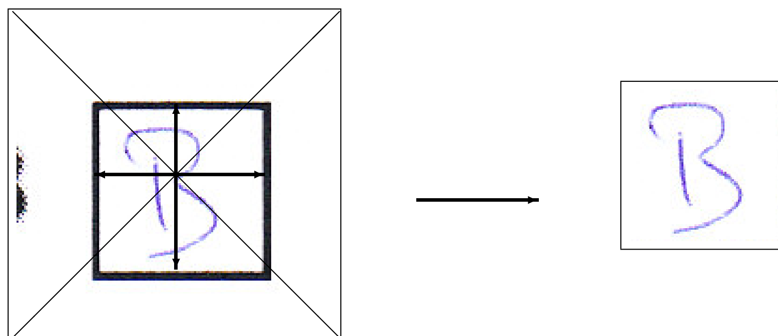
Obrázek 3.2: Předpokládané ořezání obrázku z nalezené oblasti.

Prvotní návrh pro zpracování formulářů bylo prohledávat oblast a hledat čtverce. Tento postup využívá ukázková aplikace *squares*, která je součástí knihovny openCV. Funkce nalezené čtverce zvýrazní zeleně. To se však ukázalo jako absolutně nevhodné, protože proces byl velice náročný na systémové prostředky,viz. 6.1.



Obrázek 3.3: Předpokládané ořezání obrázku z nalezené oblasti.

Druhý návrh byl již mnohem efektivnější, rychlejší a dokonce méně náročný na implementaci. Hlavní myšlenkou je procházet od středu obrazu vždy ke kraji tak, že hledáme černé linie, které zcela jistě jsou ohraničením. Procházíme jej tedy vždy od středu nahoru, dolů, doprava a doleva.



Obrázek 3.4: Předpokládané vyhledání obrázku.

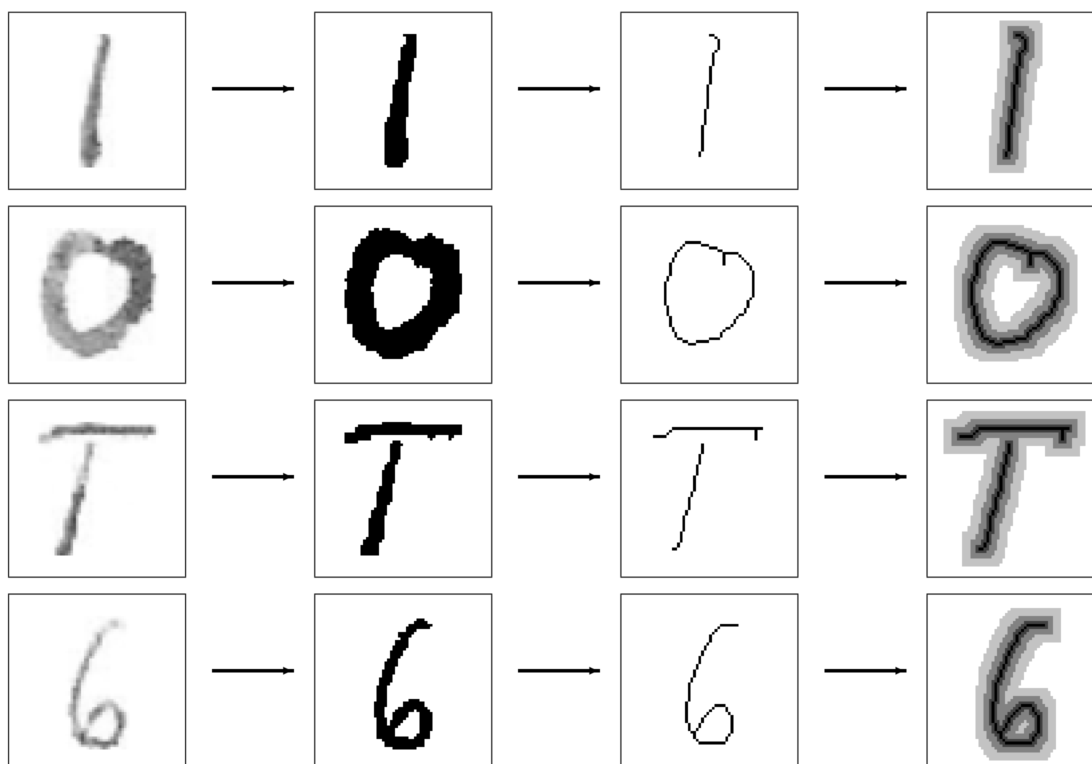
Kapitola 4

Extrakce příznaků a klasifikace

Tato kapitola se zabývá extrakci příznaků z již předzpracovaných vstupních dat. Pro klasifikaci byla využita knihovna LIBSVM.

4.1 Příprava dat

Zřejmý problém při extrakci příznaků (feature extraction) je různorodost vlastností znaků získaných od různých osob. Každý použil jakýkoliv psací nástroj jakékoliv barvy. Také šířka pera se různí. Proto je vhodné náležitě promyslet předzpracování. Níže je zobrazen postup, který byl použit v tomto projektu. Bylo použito prahování, poté skeletonizace, a nakonec obalení vrstvami.



Obrázek 4.1: Průběh úpravy obrázku pro klasifikátor.

Stojí za povšimnutí, že znak O a 6 mají znatelně odlišnou tloušťku, což by bylo pro klasifikátor nevhodné. Ve výsledku je ovšem díky skeletonizace tento problém vyřešen.

Jistě zajímavou transformací je poslední úprava, tedy obalení skeletu vrstvami. Tento postup byl zvolen proto, že každý znak má svůj specifický tvar, jelikož byl většinou napsán jedinečnou osobou. I za předpokladu, že by znaky psala jediná osoba, byly by zde jisté odchylky. Tato úprava tedy zmenšuje rozdíly mezi znaky jedné třídy, protože tam, kde je např. jen svrchní vrstva, může mít jiný znak přímo skelet. Tímto nedojde k absolutnímu zmatení trénování, kdy za použití pouze skeletu bylo rozpoznávání relativně nepřesné, viz. 6.4.

Použil jsem tedy 3 vrstvy, každá je o určitý koeficient k světlejší, než ta předchozí. Původně bylo zvoleno $k = 0.75$, po několika testování však vyšlo najevo, že nejlepší výsledky dává $k = 0.65$.

Obalení vrstvami funguje tak, že vždy na předchozí vrstvu je použita eroze. Tím dostaneme rozsáhlejší plochu tmavých pixelů. Poté nově vytvořený obraz zesvětlíme o zmíněné k. Nakonec vrstvy spojíme tak, že z puvodní vrstvy použijeme pouze pixely, které nejsou totožné s pozadím, a těmi překryjeme vrstvu nově vytvořenou. Tuto operaci provedeme v d iteracích. Jak je míněno výše, v tomto případě bylo $d = 3$.

4.2 Hledání příznaků

Hledání příznaků bývá největší problém pro následovnou klasifikaci, protože je vyžadován individuální přístup pro různé objekty [21]. Je rozdíl mezi tím, pokud potřebujeme klasifikátor pro ručně psané znaky, obličeje, otisky prstů, nebo jakosti vín. Poslední uvedený dokonce nesouvisí s počítačovým viděním, ale je zde uvedený záměrně. Klasifikovat můžeme tedy i objeky reprezentované nám známými skutečnostmi. Jelikož ale budeme rozpoznávat znaky ze vstupního obrazu, potřebujeme ze vstupního obrazového materiálu získat vektor, který bude pokud možno co nejpřesněji popisovat objekt.

Tento problém byl pro mě nejtěžším úkolem této práce. Support vector machines mají naštěstí tu výhodu, že pokud potřebujeme klasifikovat mnohomístné vektory, nebude příliš velký problém přispůsobit data potřebám klasifikátoru, viz. 2.1. Příznaky by přesto měly být diskriminativní a invariantní proti rotaci a pokud možno i deformaci [13].

4.2.1 Paprsky

Příznaky paprsek nejlépe vystihuje následující funkce. Obrázek vždy procházím od středu ke kraji tak, že počítám průměrnou hodnotu pixelů, kterým paprsek prochází.

```
for (int i=0;i<POCET_PAPRSKU; i++)
        {
        double sinx=sin(i*(360/ss));
        double cosx=cos(i*(360/ss));

        xu = img->height/2 ;
        yu = img->width/2 ;

        count=0;
```

```
        do{

            xu += cosx;
            yu += sinx;

            count += 255 - (clarityRGB (img, (int)xu, (int)yu));

        } while ((xu>.0) && (xu < img->height) &&
            (yu>.0)&&(yu<img->width) );

        if (count > 0)  // používáme sparse, nulové hodnoty
            zahodíme
        {
                //zapiš paprsek "count" do souboru
                ...
        }
}
```

Kód 4.1: Funkce paprsky. Funkce neobsahuje část kódu pro zapsání do souboru. Pro pochopení algoritmu paprsky je však tento kus kódu irelevantní.

4.2.2 Hodnoty pixelů

Pro trénování použijeme hodnoty všech pixelů vstupního obrazu. Jistým vylepšením je použití síta tak, že je použitý každý druhý pixel.

```
for (int h = 0; h < img->height; h++)
        for (int j = 0; j < img->width; j++)
        {
            if(sito)
                if ((j+h)%2)
                    continue;

            count = 255 - (clarityRGB (img, j, h));
        if (count > 0)  // používáme sparse, nulové hodnoty
            zahodíme
        {
                //zapiš hodnotu "count" do souboru
                ...
        }
```

Kód 4.2: Funkce pro zisk hodnot pixelů. Funkce neobsahuje část kódu pro zapsání do souboru. Pro pochopení algoritmu je však tento kus kódu irelevantní.

4.3 Klasifikace

Pro klasifikaci je použita knihovna LIBSVM, kterou jsem zvolil po doporučení vedoucího práce. Její použití je uvedeno v podkapitole 3.3 a nutné úpravy jsou popsány v sekcu 5.2.

Kapitola 5

Implementace

Tato kapitola se zabývá implementací a testováním. Rozhodl jsem se celou aplikaci implementovat pomocí programovacího jazyka C++.

5.1 OpenCV

Knihovna OpenCV (Open Computer Vision library) je velice vhodná pro práci s obazem a videem, což je pro tuto práci zásadní. Samozřejmě existuje mnoho jiných knihoven, dokonce i Qt framework umožňuje základní operace s obrázky, přesto OpenCV je pravděpodobně nejlepší volba pro svou variabilitu a obsáhlost dostupných algoritmů a struktur.

Jednou z mnoha výhod je nejen kvalitní dostupná literatura, např [2], ale také oblíbenost u programátorů, tím pádem probíhá neustálý vývoj této robustní knihovny. V současné době (březen 2010) existuje verze 2.0, která obsahuje několik rozšíření oproti předchozím verzím, přesto jsem použil verzi OpenCV 1.1, která dostatečně zvládne veškeré operace, které jsem požadoval.

Níže je ukázka zdrojového kódu obsahující OpenCV struktury.

```
int hueSQ (IplImage * img, int begx, int begy, int endx, int
    endy)
{
  double soucet = 0;

  for (int x = begx; x < endx; x++)
    for (int y = begy; y < endy; y++)
      soucet += clarityRGB (img, x, y);

  return (int) (soucet / (endy - begy) / (endx - begx));
}
```

Kód 5.1: Funkce zjišťuje jas obdélníkové oblasti obrázku vymezeného pomocí dvou rohů(levý horní a pravý dolní).

Tuto funkci jsem uvedl především proto, že je využívána pro zjištění verze formulářů. Jak je vidět, tato funkce dále využívá následující definice:

```
#define pixel(img, x, y, color)
    (((uchar*) (img->imageData + img->widthStep*y))[x*3+color])

#define clarityRGB(img, x, y)
    (int)( (pixel(img, x, y, 0)+
            pixel(img, x, y, 1)+
            pixel(img, x, y, 2) )/3)
```

Kód 5.2: Definice pro přístup k barvě jednoho pixelu obrázku. Definice clarityRGB získává intenzitu pixelu. Jedná se o přímý přístup do paměti. Tento způsob je jednoznačně nejrychlejší.

5.1.1 Použité struktury a funkce

- `IplImage*` : Struktura, představující obrazový materiál. Využívá ji většina funkcí, které operují s obrazem. Výhoda tkví v tom, že knihovna OpenCV alokuje pro tento obrázek automaticky paměť, takže programátor jej pouze poté uvolňuje funkcí `cvReleaseImage(&img)`.

- `CvThreshold(src, dst, threshold, maxValue, thresType)` : Tato funkce provádí prahování obrazu. Demonstrována je v sekci 2.3.2, kde je použito `threshold = 128` a `maxValuje = 255`. Proměnná `thresType` určuje typ prahování, jelikož je možno obrázek prahovat více metodami.

- `CvErode(src,dest,element, iterations)` : Tato funkce provádí erozi obrázku. Ta je demonstrována v sekci 2.3.3, kde jsou použity 4 iterace: `iterations = 4`. Proměnná `element` určuje typ eroze, nejčastěji se však využívá matice 3×3, kdy platí: `element = 0`.

- `IplImage * cvLoadImage(file, depth)` : Tato funkce načítá obrázek podle zadaného umístění `file`. proměnná `depth` určuje barevnou hloubku obrazu.

  Samozřejmě nebylo možno zde uvést všechny použité funkce a struktury, protože jich je opravdu mnoho [2]. Proto jsem uvedl jen nejdůležitější položky, které se při programování pod OpenCV používají.

5.1.2 Zpracování formulářů

Velice nepříjemné bylo zjištění, že program při načítání formulářů potřebuje velké množství operační paměti. Také doba zpracování byla nepřijatelná. Po mnoha testování jsem zjistil, že paměť je vytížená především funkcí `cvCloneImage()`. Neúměrná doba zpracování byla způsobena použitím `cvGetPixel()` a `cvSetPixel()`. Tyto funkce jsou opravdu pomalé, což potvrzují i testy, viz. 6.1.

Další úprava se týkala vylepšení algoritmu pro hledání rohů formulářů (podle čtyř černých čtverců). Výsledky různých postupů jsou v podkapitole 6.1.

5.2 LIBSVM

Jak již bylo řečeno v kapitole 3.3, pro klasifikaci byla použita knihovna LIBSVM. Přestože je knihovna velice kvalitně rozvržena, musel jsem pozměnit několik pomocných souborů,

aby bylo možno knihovnu do aplikace zakomponovat.

Základním souborem této knihovny je *svm.cpp*. Kód je strukturovaný objektově a v podstatě obsahuje veškeré potřebné algoritmy a struktury pro řešení, trénování a predikci.

Níže jsou uvedené úpravy, které bylo potřeba provést pro správný běh knihovny.

- odstranění main funkcí
- místo volání `exit()` voláno `return()`.

5.3 Qt framework

Jak jsem již zmínil v kapitole 3.2, rozhodnul jsem se pro Qt framework. Práce s Qt samotným, bez importovaných knihoven se jeví jako velmi pohodlná a intuitivní. Potřeboval jsem však zkombinovat Qt s knihovnou LIBSVM a OpenCv. První zmíněnou, tedy LIBSVM, není příliš náročné přilinkovat do samostatné složky a přikompilovat do zdrojových souborů v Qt, protože zdrojové soubory jsou volně k dispozici a navíc nejsou příliš velké (5 souborů, 5000 řádků). Větším problémem bylo nalézt řešení, jak připojit knihovny OpenCV. Tato knihovna je podstatně rozsáhlejší, navíc na různých platfomách by nemusely knihovny fungovat správně, pokud by byly součástí projektu. Proto jsem se rozhodnul situaci vyřešit tak, že projekt bude přeložitelný pouze na systémech, kde je OpenCV nainstalován. K tomuto kroku je ovšem nutný zásah do Makefile souboru, který je při spuštění programu `qmake` automaticky generován. Ten se generuje pomocí souboru *pro*. Proto jedna z mála možností, jak efektivně připojit tuto knihovnu, bylo přidat do projektového souboru následující řádek:

```
unix:CONFIG +=link_pkgconfig
unix:PKGCONFIG += opencv
```

Tímto byly vyřešeny veškeré problémy s knihovnou OpenCV v unixových systémech, navíc řešení je velice elegantní.

Přesto se tímto nevyřešily problémy překladu v operačních systémech Windows. Zde bylo nutné v projektovém souboru importovat knihovnu OpenCV explicitně, protože program `pkg-config` bohžel není součástí Windows.

```
win32:LIBS += -L,,D:\OpenCV\lib‘‘ -lcv -lcvaux -lhighgui -lcxcore
```

Zdá se tedy, že problém s knihovnou OpenCV je vyřešen. Přesto aplikace byla mimo linux nepřeložitelná. Důvodem byla funkce `mkdir()`, která vytváří novou složku. Tato funkce potřebuje v unix-like systémech jeden parametr navíc, a to informace o právech vytvářené složky. Řešení tohoto problému je následující:

```
#ifndef _WIN32                        // nejsme ve windows
        if (mkdir ("./DB", 0755))
   #else                              // jsme ve windows
        if (mkdir ("./DB"))
   #endif
        {
          showWarning ("nelze vytvorit slozku DB ");
        }
```

Kód 5.3: Úprava zdrojového kódu pro více platforem. Funkce mkdir() potřebuje totiž různý počet parametrů pod různými kompilátory.

V neposlední řadě bylo nutné nastavit projektový soubor tak, aby při zabalení programu zahrnoval i důležité programové soubory. Dejme tomu, že potřebujeme se zdrojovým kódem zahrnout do instalačních souborů i soubor *look.dat* a soubor *1.txt* v adresáři *DATA*. To se provede následujícím příkazem:

```
DISTFILES += DATA/look.dat \
    DATA/1.txt \
```

5.4 Další informace

Aplikace byla vytvářena tak, aby splňovala základní požadavky pro přehlednost kódu. Komentáře proto byly psány tak, aby bylo možno pomocí programu *doxygen* vytvořit programovou dokumentaci. Ta je součástí CD, viz. A.

Odsazení zdrojového kódu bylo upraveno programem *indent* pro lepší přehlednost a jednotnost.

Kód je částečně objektově orientovaný. Moduly pro práci s obrazem však byly implementovány jako funkce.

Aplikace byla od počátku navrhována tak, aby byla přeložitelná a spustitelná na více platformách. Testována byla na následujících sytémech, kde byla plně funkční.

- eva.fit.vutbr.cz (FreeBSD 8.0-STABLE)
- merlin.fit.vutbr.cz (GNU/Linux 2.6.32.11 x86_64)
- Windows 7 proffesional x86_64
- Ubuntu GNU/Linux 2.6.24-27-generic x86_64

Kapitola 6

Testování

Tato kapitola se zabývá testováním během vývoje projektu.

6.1 Nároky na hardware

Během testování byly odhaleny nedostatky v oblasti alokace paměti. Především při zpracovávání formulářů byly vytěžovány systémové prostředky velice zřetelně. Proto byly některé algoritmy během vývoje programu upraveny tak, aby aplikace fungovala bez problému. Tabulka zobrazuje náročnost na čas a paměťové zdroje při zpracování 12 formulářu.

Time(cpu) s	Top(mem) MB	poznámky
17,692	27,7	s původní implementace
16. 117	12,9	bez rotate, cvGet2D, cvSet2D, cloneImage
8, 426	1,1	Bez GET2D, SET2D, bez cloneImage
7,187	1,1	Po úpravě ořezávání

Tabulka 6.1: Tabulka znázorňuje, jaký vliv mají na běh programu různé funkce knihovny OpenCV.

Je tedy zřejmé, že funkce `cvSet2D` a `cvGet2D` jsou opravdu velmi nevhodné, především tedy časově náročné. Řešením tohoto problému je přímý přístup do paměti.

Další problém způsobuje funkce `cvCloneImage`, která v podstatě plní svou funkci dobře. Problém je ten, že při zpracování formulářů je potřeba získat mnoho výřezů, které jsou dále zpracovávány. Pokud je tedy `cvCloneImage` volána 256krát během zpracování jednoho formuláře(podle počtu znaků na formuláři), aplikace alokuje opravdu mnoho systémové paměti. Samozřejmě že takto alokovaná paměť se uvolňuje, a to pomocí cvReleaseImage, přesto jiné řešení je elegantnější, nepotřebuje tolik systémové paměti a dokonce mnohem rychlejší.

Tím řešením je použití `cvCreateImageHeader`. Takto se sice nevytvoří nový obrázek v paměti, ale pouze se vymezí oblast, se kterou budeme pracovat podle `cvRect`Ṗresto se takto vytvořený objekt chová jako obrázek a můžeme přistupovat k jeho pixelům zcela běžným způsobem, což my potřebujeme. Jediný problém je v tom, že pokud bychom v tomto obrázku změnili hodnoty pixelů, výsledek se samozřejmě projeví i u obrázku, ze kterého jsme výřez vytvářeli. To ale naštestí není potřeba při hledání čtverců, protože budeme pouze číst obrazové informace.

V neposlední řadě bylo také potřeba upravit algoritmus pro hledání čtverců. Ten byl upraven tak, že neprocházel již celý formulář, ale vždy pouze část, kde se tento čtverec předpokládal, tedy u rohů.

```
for (int smer = 0; smer < 4; smer++)
{                                    // budeme hledat čtyři černé
   čtverce a potom podle nich ořežeme formulář
   switch (smer)
   {
    case 0: it_X = 1; it_Y = 1;  break; //levý horní roh
    case 1: it_X = 1; it_Y = -1; break; //levý dolní roh
    case 2: it_X = -1; it_Y = 1; break; //pravý horní
    case 3: it_X = -1; it_Y = -1; break; //pravý dolní

      for (int h = 0; h < gray->height / 3; h += 2)
      {
         for (int j = 0; j < gray->width / 3; j += 2)
         {
               s_Y = h;
               s_X = j;

               if (it_Y < 0)
                   s_Y = gray->height - h - 1;
               if (it_X < 0)
                   s_X = gray->width - j - 1;

            if (clarityRGB (gray, (int) (s_X), (int) (s_Y)) > 1)
               continue; // dokud není aspon nějaký černý pixel,
                   přeskakuji zbytek těla

               .
               .
               .

         }
      }
   }
}
```

Kód 6.1: Úprava zdrojového kódu pro hledání černých čtverců. Algoritmus prochází pouze do třetiny z obrazu. Navíc je kontrolován každý druhý pixel – černé čtverce jsou dostatčně velké.

Výsledek použití algoritmu 6.1:

původně:	1687314 operací v cyklech
po optimalizaci:	274305 operaci v cyklech

6.2 Úspěšnost rozpoznávání

Během vývoje aplikace bylo potřeba průběžně testovat i správnost nastavení parametrů knihovny LIBSVM a vhodnost extrakce příznaků. Původně byly používány pouze příznaky plátno. Parametry pro trénování byly nastaveny následovně:

```
-b 1 -s 0 -t 0 -c 10 -m 20 -g 1
```

úspěch	TRAIN/TEST	Success/all	poznámky
61,4%	1500/500	307/500	základní matice
60,8%	1500/500	304/500	maskováno, prahováno
70 %	1500/500	350/500	prahováno
74 %	1500/500	370/500	prahováno, skelet, zvětšení
77,4%	1500/500	387/500	prahováno, skelet, eroze(2x)
76,6%	1500/500	383/500	prahováno, skelet, eroze(3x)
77,6%	1500/500	388/500	jako předchozí + slupky 75%
79,2%	1500/500	396/500	kalibrace slupek na 65%

Tabulka 6.2: Různé přístupy features. Datasety byly původně mírně nevyrovnané, proto jsou výsledky poněkud horší.

Program je také schopen vytvořit trénovací sadu tak, aby byl ve všech třídách shodný počet položek. Následující tabulka demonstruje vliv vyrovnanosti trénovací sady na výslednou chybovost.

```
-b 1 -s 0 -t 1 -m 200
```

typ znaků	TRAIN/TEST	vyrovnanost sady	úspěšnost
číslice	6000/1500	ANO	**95,9%**
		NE	95,53%
písmena	6006/2002	ANO	89,01%
		NE	89,11%
oboje	6012/2016	ANO	82,54%
		NE	82,04%

Tabulka 6.3: Porovnání úspěšnosti rozpoznávání různých tříd znaků. Datasety jsou nastaveny podle potřeby. Je vidět, že zcela vyrovnané trénovací datasety nepatrně zlepšují výslednou predikci.

Jak bylo řečeno v sekci 3.1, vyzkoušel jsem úspěšnost testování také na referenční databázi ručně psaných číslic MNIST. Výsledek byl velice překvapivý. Parametry testování byly nastaveny následujícím způsobem:

```
-b 1 -s 0 -t 1 -m 200
```

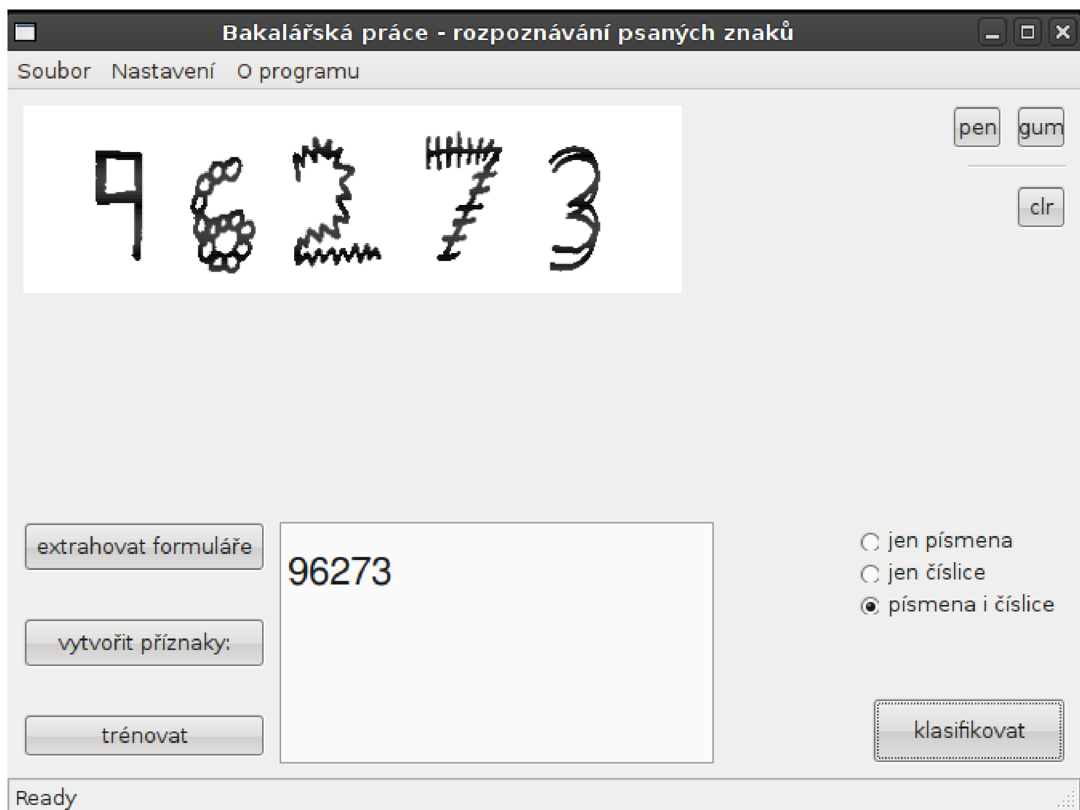
MNIST je velmi obsáhlou databází, a to se projevilo také u doby trénování, které trvalo 1 hodinu a 49 minut. Doba testování byla 4 minuty 38 sekund.

úspěch	TRAIN/TEST	Success/all	poznámky
95,05 %	60000/10000	9504/10000	MNIST databáze

Tabulka 6.4: Výsledek testování na databázi MNIST.

Protože však nebyl program původně konstruován pro MNIST, je potřeba do složky **DB** nejprve vložit trénovací data, vytvořit příznaky a výsledný soubor **dataset** přesunout z adresáře **TRAIN** do adresáře **TEST**. Poté je potřeba stejnou proceduru provést pro testovací sadu, s tím rozdílem, že nynější soubor **dataset** ve složce **TRAIN** je potřeba přejmenovat na **testset**. Poté jej můžeme umístit do adresáře **TEST** a můžeme spustit testování, ovšem je potřeba zatrhnout položku „jen trénovat“. V opačném případě bychom si smazali oba příznakové soubory.

6.3 Ukázka v praxi



Obrázek 6.1: Program dokáže zpracovat i nepřirozená vstupní data. Vstupní data převzata z [10].

Následující část zobrazuje schopnost rozpoznat některá slova.

→ INTELI6ENT

→ RESEARCM

→ 1S8

→ DVACETIKQRUNY

→ ALENANDR

→ 1H2C3F

Obrázek 6.2: Průběh úpravy obrázku pro klasifikátor

Ve čtvrtém a šestém případě je chyba způsobená tím, že pro predikci byla použita možnost, kdy se rozpoznávají číslice i písmena. Po zatrhnutí rozpoznávání jen čísel u třetí položky byla predikce již správně, **158**. Stejně tak, po zatrhnutí rozpoznávání jen písmen u prvního příkladu byla predikce již správně, **INTELIGENT**.

Kapitola 7

Závěr

Cílem práce bylo vytvořit software pro rozpoznávání ručně psaných písmen a číslic. Tento úkol byl splněn a vznikl multiplatformní program s grafickým uživatelským rozhraním. V rámci projektu byly navrženy formuláře pro jednoduchý sběr potřebných dat a následovně byly vytvořeny datové sady, na kterých jsou prezentovány výsledky rozpoznávání.

Důležité je, že výsledky rozpoznávání byly nad očekávání velice dobré. Úspěšnost pro samotné číslice byla **95,9%**, pro samotné znaky **89,11%**. Pro klasifikaci všech tříd, tedy číslic i znaků, byla výsledná úspěšnost **82,54%**. Chybovost byla způsobena především podobností číslice nuly a písmene O, nebo podobností písmene G a číslice šest. Program byl také otestován na databázi MNIST, kde úspěšnost rozpoznávání byla **95,05%**.

Projekt by šlo rozšířit o mnoho dalších vylepšení. Například rozpoznávání slov s diakritikou, tedy písmena s háčky a čárkami. Také by bylo možné implementovat rozšíření pro rozpoznávání malých tiskacích znaků. Stačilo by obstarat patřičnou databázi těchto znaků, v případě diakritiky přidat několik pomocných algoritmů. Také by bylo možno program rozšířit o možnost kontrolovat výsledný text podle slovníků. Také by se daly zkoumat i další příznaky, což by mohlo celkovou predikci nejen zlepšit, ale také případně zrychlit.

Problém by nastal při pokusu program orientovat pro psací písmo, kde by bylo potřeba detekovat tahy pera již přímo při psaní, protože každý jednotlivec píše velmi specifickým stylem, a rozdíly napsaných slov různými lidmi jsou opravdu velké. Tímto bychom dostali vektorovou reprezentaci v čase, což bychom pro klasifikátor mohli použít. Přesto by byl problém zjistit, kolik písmen slovo obsahuje (protože slovo je napsáno většinou jedním tahem) a také by bylo obtížné rozpoznávat slova z textů, které jsou již napsané (například nascanovaný dopis). Museli bychom totiž odhadovat co nejpřesněji, jakými tahy autor textu slovo napsal.

Literatura

[1] BISHOP, C. M.: *Pattern Recognition and Machine Learning*. Springer, 2006, 740 s., iSBN 978-0-387-31073-2.

[2] BRADSKI, G.; KAEHLER, A.: *Learning OpenCV: Computer Vision With The OpenCV Library*. O'reilly Media, 2008, 555 s., iSBN 978-0-596-51613-0.

[3] CHANG, C.-C.; LIN, C.-J.: LIBSVM – A Library for Support Vector Machines. [online], [cit. 2010-04-27].
URL http://www.csie.ntu.edu.tw/~cjlin/libsvm/

[4] DOBEŠ, M.; ZEMČÍK, P.: *Zpracování obrazu a algoritmy v C#*. BEN - technická literatura s.r.o., 2008, 144 s., iSBN 978-80-7300-233-6.
URL http://www.fit.vutbr.cz/research/view_pub.php?id=8763

[5] FLETCHER, T.: Support Vector Machines Explained. 2009.
URL www.tristanfletcher.co.uk/SVM%20Explained.pdf

[6] GOCKENBACH, M. S.: Creating a sparse matrix. [online], [cit. 2010-05-05].
URL http://www.math.mtu.edu/~msgocken/intro/node18.html

[7] HEIDE-HERMANN, T.: Myslící počítače. *100+1*, , č. 23, 2009: s. 14–15, ISSN 0332-9629.

[8] HLAVÁČ, V.; ŠONKA, M.: *Počítačové vidění*. Grada, 1992, 252 s., iSBN 80-85424-67-3.

[9] IVANCIUC, O.: SVM - Support Vector Machines Software. [online], [cit. 2010-05-11].
URL http://www.support-vector-machines.org/SVM_soft.html

[10] LECUN, Y.; BOSER, B.; DENKER, J. S.; aj.: Handwritten digit recognition with a back-propagation network. In *Advances in Neural Information Processing Systems 2 (NIPS*89)*, editace D. Touretzky, Denver, CO: Morgan Kaufman, 1990.

[11] LECUN, Y.; CORTES, C.: The MNIST database of handwritten digits. [online], [cit. 2010-04-27].
URL http://yann.lecun.com/exdb/mnist/

[12] LECUN, Y.; JACKEL, L. D.; BOTTOU, L.; aj.: Learning Algorithms For Classification: A Comparison On Handwritten Digit Recognition. In *Neural Networks: The Statistical Mechanics Perspective*, editace J. H. Oh; C. Kwon; S. Cho, World Scientific, 1995, s. 261–276.
URL http://leon.bottou.org/papers/lecun-95a

[13] SCHWARZ, P.; BURGET, L.; ČERNOCKÝ, J.; aj.: Klasifikace a rozpoznávání. [online], [cit. 2010-05-07].
URL http://www.fit.vutbr.cz/study/courses/IKR/public/prednasky/

[14] THELIN, J.: *Foundations of Qt Development*. Apress, 2007, 528 s., iSBN 978-1-59059-831-3.

[15] WWW stránky: Identifikace osob. [online], [cit. 2010-04-25].
URL http://www.prf.cuni.cz/documents/docFile.php?id=760

[16] WWW stránky: Support vector machine. [online], [cit. 2010-05-05].
URL http://www.statemaster.com/encyclopedia/Support-vector-machine

[17] WWW stránky: Vladimir Vapnik. [online], [cit. 2010-05-05].
URL http://www.statemaster.com/encyclopedia/Vladimir-Vapnik

[18] WWW stránky: Adobe Help Resource Center - Bitmaps. [online], [cit. 2010-05-11].
URL http://help.adobe.com/en_US/Director/11.0/help.html?content=06_bitmaps_01.html

[19] WWW stránky: Fakulta informačních technologií. [online], [cit. 2010-05-14].
URL http://www.fit.vutbr.cz

[20] WWW stránky: Úvod do strojového učení (v počítačové lingvistice). [online], [cit. 2010-05-14].
URL http://wiki.matfyz.cz/wiki/Úvod_do_strojového_učení_(v_počítačové_lingvistice)

[21] ZEMČÍK, P.; JURÁNEK, R.; LÁNÍK, A.: Počítačové vidění. [online], [cit. 2010-05-09].
URL http://www.fit.vutbr.cz/study/courses/POV/

Dodatek A

Obsah CD

Optické medium obsahuje všechny potřebné součásti programu:

README soubor popisující adresářovou strukturu a aktivity se souborem Makefile

Makefile kompilační soubor pro program make

src/ adresář obsahuje zdrojové soubory samotného programu a vyplněné formuláře

report/ adresář obsahuje soubory pro tvorbu technické zprávy pomocí LaTeXu

doxygen/ adresář obsahuje materiál pro tvorbu programové dokumentace

others/ adresář obsahuje ostatní soubory (nevyplněné formuláře)

Dodatek B

Manuál

Program byl testován především na Operačním systému Ubuntu 64bit. Dále byla jeho funkčnost testována na školním počítači **merlin** (merlin.fit.vutbr.cz), **eva** (eva.fit.vutbr.cz) a osobním systému Windows 7 (x86_64).

B.1 instalace

pokud bude aplikace spouštěna pomocí vzdáleného serveru merlin (eva), je potřeba aktivovat grafické rozhraní pomocí přepínače **-X**:

```
ssh -X uzivatel@server.fit.vutbr.cz
```

V případě, že se nacházíme v kořenové složce, můžeme spouštět následující příkazy.

1. vytvoříme Makefile pomocí programu `qmake`.

   Pro server **merlin** je potřeba nastavit korektní cestu na program `qmake`, jelikož implicitně je nastavená nevhodná verze Qt knihoven (3.3.6) i programu `qmake(1.0.7a)`. tímto dostaneme `qmake`verze 2.0.1a a Qt knihovny verze 4.5.0.

   ```
   PATH=/usr/local/share/Qt-4.5.0/bin:$PATH;
   cd src && qmake;
   ```

   Server **eva**:

   ```
   cd src && qmake-qt4;
   ```

2. Navrátíme se zpět do kořenové složky.

   ```
   cd ..;
   ```

3. Poté pro jistotu smažeme soubory vygenerované na jiném systému.

   ```
   make clean;
   ```

4. A přeložíme.

```
make;
```

5. Nyní již můžeme aplikaci spustit

   buďto pomocí programu `make`

```
make run;
```

   ,nebo přímo spuštěním binárního souboru

```
cd src && ./BC;
```

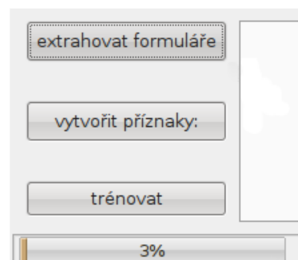
B.2 Ovládání

Program byl konstruován tak, aby nebyl problém pochopit jeho jednotlivé funkce. Po instalaci programu bude pravděpodobně nutné před samotnou klasifikací prověst několik důležitých kroků.

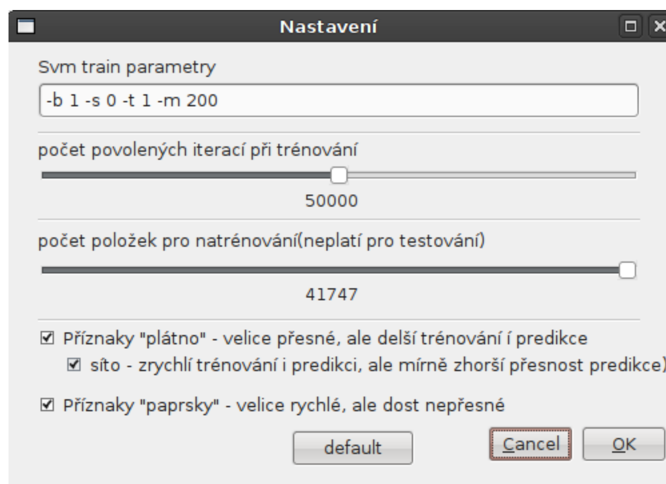
Podmínka je, aby ve složce **src** byly adresáře **SCANNED** a **DATA**

Adresář **SCANNED** by měl obsahovat vyplněné formuláře ve formátu `*.jpg.`Adresář **DATA** obsahuje konfigurační soubory.

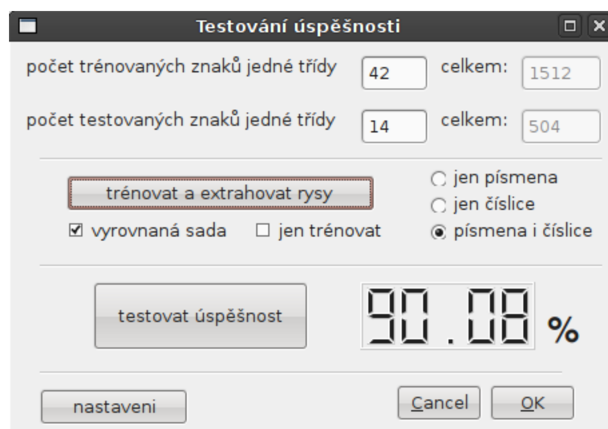
Pokud je vše v pořádku, je možno extrahovat z těchto formulářů jednotlivé znaky. Toho docílíme stisknutím tlačítka „extrahovat formuláře“. Poté je potřeba vytvořit příznaky. To se provede stisknutím tlačítka „vytvořit příznaky“. Nakonec je potřeba databázi natrénovat, což může trvat poněkud déle. To provede tlačítko „trénovat“.



Obrázek B.1:



Obrázek B.2: Obrazovka s nastavením programu. Program je možné nastavit podle svých vlastních potřeb, včetně parametrů pro trénování LIBSVM.



Obrázek B.3: Testovací obrazovka. Je možno nastavit různé předvolby pro testování.

Dodatek C

Formulář

Vyplňte prosím následující kolonky příslušnými znaky
Určeno pro 2 osoby

Verzion 9,10

A	A	J	J	S	S	1	1	GIN	GIN
B	B	K	K	T	T	2	2	KAZ	KAZ
C	C	L	L	U	U	3	3	HRAD	HRAD
D	D	M	M	V	V	4	4	PEPA	PEPA
E	E	N	N	W	W	5	5	HANKA	HANKA
F	F	O	O	X	X	6	6	SUM41	SUM41
G	G	P	P	Y	Y	7	7	202773	202773
H	H	Q	Q	Z	Z	8	8	ROZHLED	ROZHLED
I	I	R	R	O	O	9	9	UNIFORMA	UNIFORMA

A	A	J	J	S	S	1	1	RUS	RUS
B	B	K	K	T	T	2	2	LOV	LOV
C	C	L	L	U	U	3	3	5391	5391
D	D	M	M	V	V	4	4	IVAN	IVAN
E	E	N	N	W	W	5	5	POWER	POWER
F	F	O	O	X	X	6	6	MISKA	MISKA
G	G	P	P	Y	Y	7	7	HAVRAN	HAVRAN
H	H	Q	Q	Z	Z	8	8	REZERVA	REZERVA
I	I	R	R	O	O	9	9	DIRIGENT	DIRIGENT

Obrázek C.1: Vyplněný formulář. Jeden list vyplňují dvě osoby.