MASARYKOVA UNIVERZITA

FAKULTA INFORMATIKY

Podpora autodetekce obrázků pro systém G.A.T.E.

Diplomová práce

Bc. Radek Šťasta

2012

Strana obsahující v tištěné podobě kopii listu zadání práce

Prohlašuji, že tato práce je mým původním autorským dílem, které jsem vypracoval samostatně. Všechny zdroje, prameny a literaturu, které jsem při vypracování používal nebo z nich čerpal, v práci řádně cituji s uvedením úplného odkazu na příslušný zdroj.

……

Podpis autora práce

Chtěl bych zde také vyjádřit poděkování RNDr. Radku Ošlejškovi, Ph.D. za odborné vedení této diplomové práce.

Shrnutí

Tato diplomová práce se zabývá problematikou detekce objektů v obrázcích pomocí knihoven nebo programů, které jsou volně k dispozici na internetu. Vznikla z potřeby rozšíření projektu G.A.T.E., který je vyvíjen na Fakultě informatiky Masarykovy univerzity. V první části této práce jsou popsány základní teoretické znalosti potřebné k porozumění oblasti rozpoznávání objektů v obecně jakékoliv digitální obrazové informaci. V této části je také popsána funkcionalita konkrétních programů a knihoven, které jsou využity v následujících částech práce. Další rozsáhlejší částí diplomové práce jsou testy zvolených programů a knihoven, na základě kterých jsou následně vybrána konkrétní nastavení parametrů vyhovujících určitým podmínkám detekce. V poslední části je popsána funkcionalita aplikace, která je výsledkem této diplomové práce.

Klíčová slova

Detekce objektů, detekce tváří, detekce postav, detekce automobilů, detekce motocyklů, detekce v obrázcích, detekce ve fotografiích, faint, Betaface, Haarovské příznaky, OpenCV, Enterprise Java Beans.

Obsah

1. Úvod ... 6

2. Detekce objektů ... 8

2.1. Detekce tváře ... 10

2.1.1. Vlastní vektory a Eigenfaces ... 13

2.1.2. Haarovské příznaky ... 17

2.1.3. OpenCV ... 22

2.1.4. Betaface ... 24

2.2. Detekce ostatních objektů ... 26

2.3. Optické rozpoznávání znaků ... 29

2.3.1. GOCR ... 31

3. Java Beans ... 33

4. faint ... 39

4.1. Struktura faintu ... 41

4.2. Změny ve faintu ... 43

5. Betaface ... 45

6. Testy ... 47

6.1. Testování detekce tváří a jeho výsledky ... 50

6.1.1. Výběr vhodného nastavení ... 57

6.1.2. Dodatečné testy ... 62

6.1.3. Testování detekce Betaface a jeho výsledky ... 64

6.2. Testování detekce ostatních objektů ... 67

6.2.1. Testování detekce postav ... 69

6.2.1.1. Princip snížení počtu chybných detekcí ... 74

6.2.2. Testování detekce automobilů a motocyklů ... 77

7. Implementace ... 82

7.1. FaceDetection.java ... 82

7.2. BodyDetection.java ... 84

7.3. CarDetection.java a MbikeDetection.java ... 87

7.4. EJB rozhraní DetectionGateLocal.java ... 89

8. Závěr ... 91

9. Literatura ... 94

1. Úvod

Na internetu se v různých formách (ať již jako jednotlivé knihovny, tak hotové programy) vyskytuje mnoho nástrojů, které umožňují detekci objektů v obrázcích. Mým primárním úkolem, jakožto diplomanta, je prostudovat tuto doménu spadající do oblasti počítačové grafiky. Poté, co získám v této oblasti přehled o tom, jaké knihovny a programy jsou volně dostupné, vyberu ty z nich, které se budou z mého pohledu zdát nejpoužitelnější, a uskutečním testy, které prověří jejich základní funkcionalitu a efektivitu. Díky těmto testům budu moci následně rozhodnout, které z nich začlením do aplikace, která bude výsledkem mé diplomové práce. Základní funkcí této aplikace bude samozřejmě detekce několika typů objektů v obrázku. Tyto detekované objekty následně nějakým způsobem popíši (nejlépe pomocí obdélníkových oblastí kolem nich) a tento popis poslouží jako výsledek detekce. Kromě toho bude tato aplikace umožňovat detekci pomocí několika přednastavených profilů, které, jak doufám, vzniknou na základě provedených testů. Ideálně by tato aplikace měla fungovat v prostředí EJB[1], protože celá má diplomová práce vzniká pod záštitou skupiny lidí spolupracujících na projektu G.A.T.E, který je na EJB založený.

Projekt G.A.T.E., který je vyvíjený na Fakultě Informatiky Masarykovy Univerzity v Brně, se zaměřuje na následující cíle. Zaprvé se tento projekt zaměřuje na vývoj utilit pro snadnou anotaci obrázků. Zadruhé se snaží poskytnout nevidomým uživatelům podporu při „prohlížení“ obrázků a zatřetí vyvíjí systém, který generuje obrázky pomocí dialogu s uživatelem a umožňuje tak nevidomým vytvářet jednoduchou počítačovou grafiku.

Důležitým modulem projektu G.A.T.E. je Annotator. Tento modul zprostředkovává navigaci v obrázku a jeho základní úlohou je informovat uživatele o obsahu obrázku. Pro tento účel má dva různé prostředky, jak zprostředkovat obsah obrázku, a to slovně a zvukově. Stejně tak podporuje dva

[1] Enterprise Java Beans jsou serverové komponenty pro modulární tvorbu podnikových aplikací

základní nástroje pro podporu komunikace s uživatelem. Prvním z nich je What-Where Language (zkráceně WWL, umožňuje pokládat dotazy typu „Co je uprostřed obrázku“, nebo „Kde je na obrázku věž“ apod.). Druhým nástrojem je rekurzivní navigační mřížka, která rozděluje obrázek na devět oblastí podle čísel na numerické klávesnici a každou z těchto oblastí na dalších devět oblastí atd.

Pokud jde o získávání informací z obrázku, existují v projektu G.A.T.E. pro dvě výše zmíněné strategie reprezentované moduly Guide a Explorer. Modul Guide zajišťuje slovní informaci, kterou získá jednak díky anotacím v obrázku a jednak přímo z formátu obrázku. Modul Explorer zajišťuje komunikaci pomocí myši, digitizéru nebo numerické klávesnice. Informace, kterou uživatel získává, je založena na konkrétním bodě v obrázku, objektu nebo obdélníku, na který uživatel „ukáže“, a jako výstup je použit primárně zvuk. Hranice obrázku jsou signalizovány speciálním zvukem a jakákoliv informace o barvě jednotlivých oblastí je signalizována pomocí speciálního spojení vybraných zvuků a jejich intenzit pro základní barvy na RGB stupnici. [1]

V první části mé diplomové práce tedy popíšu spíše z teoretického hlediska určité principy, které jsou využitelné v oblasti detekce objektů v obrázcích a zmíním také konkrétní nalezené knihovny a programy, které pro svou práci využiji. V další části uvedu proceduru testování těchto vybraných aplikací společně s jejich testy a v posledních kapitolách mé diplomové práce popíši mnou navrženou aplikaci a její poskytovanou funkcionalitu.

2. Detekce objektů

Detekcí objektů je myšlena výpočetní úloha spadající do oblasti počítačové grafiky, využívající techniky a metody z dalších informatických oblastí jako je například statistika nebo umělá inteligence (většina metod pro detekci objektů využívá poznatky z oblasti umělé inteligence k tomu, aby se nejprve „vytrénovala“ na cvičné sadě několika set až tisíců pozitivních a negativních příkladů daného objektu a díky tomu pak bylo možné detekovat jakýkoliv objekt stejného typu s větší či menší pravděpodobností úspěchu).

Co se týče samotného přístupu k zachyceným objektům, je třeba rozlišovat dva základní přístupy, jakým způsobem chceme zvolený obrázek (obecně jakkoliv digitálně zpracovanou obrazovou informaci, tedy například i video) zkoumat. Jedním z těchto přístupů je detekce objektů (tou se budu zabývat ve své diplomové práci) a druhým pak rozpoznávání objektů (anglicky „object detection a object recognition“). Základním rozdílem mezi těmito přístupy je to, že v případě detekce objektů máme k dispozici nějakou obrazovou informaci a zkoumáme, zda se v ní vyskytuje objekt určitého typu. V případě rozpoznávání objektů máme tutéž obrazovou informaci a zkoumáme, zda se v ní vyskytuje konkrétní jedinečná instance objektu daného typu. Můžeme mít obrázek, na kterém je určitý počet osob, a chceme pomocí výpočetní techniky zjistit, kolik lidí se na obrázku nachází. Potom použijeme metodu detekce objektů, která je schopna na tuto otázku odpovědět a případně dodat ještě nějaké další informace, např. možný věk všech rozpoznaných lidí (i když jsou tyto informace většinou jen velice orientační a záleží na mnoha faktorech, jako je kvalita a rozlišení obrázku, pozice jednotlivých osob, osvětlení apod.). Pokud chceme zjistit, jestli se mezi těmito lidmi nachází jeden konkrétní člověk, použijeme metodu rozpoznávání objektů, která je schopna rozhodnout, jestli ano nebo ne.

Metoda rozpoznávání objektů je v mnoha směrech jednodušší, protože nepotřebujeme tolik referenčních obrázků konkrétního typu objektu, abychom co nejvíce pokryli všechny jeho možné varianty. Stačí nám referenční obrázky jednoho konkrétního objektu a i jeho vyhledávání v jiném obrázku může probíhat v kratším čase. Na druhou stranu po této metodě požadujeme daleko vyšší pravděpodobnost toho, že správně tento objekt rozpoznáme a nižší pravděpodobnost toho, že chybně detekujeme jiný objekt. Toho se pak využívá například při autorizaci a autentizaci pomocí biometrik.

Velice důležitým faktorem při detekci objektů v digitální obrazové informaci je to, jaký objekt jsme se rozhodli detekovat. Ve skutečnosti není tato oblast informatiky ještě na tak vyspělé úrovni, abychom byli schopni detekovat téměř jakýkoliv objekt. Je tomu spíš naopak. Existuje zatím jen velice malá množina typů objektů, které detekovat dokážeme. Je to dáno především tím, že takřka každý objekt v našem reálném světě je originální, a je velice těžké najít generalizaci, která by pokrývala všechny tyto originality a je vůbec otázka, zda se někdy vůbec něco takového podaří.

Přesto existuje několik skupin objektů, které jsme schopni detekovat a některé z nich dokonce i s docela vysokou přesností a pravděpodobností úspěchu. Mezi nejlépe detekovatelné objekty patří lidská tvář, a právě proto se detekcí obličejů v obrázcích budu ve své práci zabývat nejvíce.

2.1. Detekce tváře

Detekce tváře v obrázku je specializovanější úloha detekce objektů, která si bere za cíl detekovat, zda se ve vybraném obrázku vyskytuje lidská tvář, nebo ne. Jedním z úkolů mé diplomové práce bylo zmapovat existující knihovny podporující nějakou formu detekce objektů v obrázku, které jsou volně dostupné, a tudíž využitelné pro systém G.A.T.E. Pro tento účel jsem provedl jednoduchý pokus, kterým jsem si vytvořil základní představu o tom, kolik je takových knihoven na internetu dostupných a jakými typy detekce objektů se zabývají. Pokus spočíval v tom, že jsem zadával různá klíčová slova do internetového vyhledávače Google. Tato metoda je v jistém smyslu velice nepřesná, protože ve výsledcích zajisté byly zastoupeny odkazy, které se vyhledávanou problematikou vůbec nezabývají, nebo odkazy již neaktuální. Za předpokladu, že takovýchto odkazů se v každém výsledku vyskytovalo přibližně stejné procento, byla však tato metoda podle mého názoru dostačující. Následující tabulka shrnuje všechna spojení klíčových slov, která jsem při pokusu použil, společně s počtem výsledků vyhledávání.

image "object detection"	1 090 000
image "object detection" library	427 000
image "object detection" library "open source"	155 000
image "face detection"	9 790 000
image "face detection" library	3 280 000
image "face detection" library "open source"	1 460 000
image "face detection" "java library" "open source"	9 390
image "object detection" -"face" library	223 000
image "object detection" -"face" library "open source"	68 200

Před stručným vysvětlením, čeho jsem tímto pokusem dosáhl, bych chtěl ještě jednou zdůraznit, že tento pokus rozhodně nemůže sloužit jako důkaz toho, v jaké oblasti detekce objektů v obrázcích je dosaženo největšího pokroku, a tedy pro jakou oblast existuje nejvíce materiálů a knihoven.

Jak je z tabulky vidět, na internetu se vyskytuje zhruba milion článků zabývajících se detekcí objektů v obrázcích. Naproti tomu však existuje desetkrát tolik stránek, zabývajících se nikoliv detekcí objektů, ale přímo detekcí tváře. Při mých dalších zkoumáních v této oblasti se tento fakt více méně potvrdil a mohu tudíž říci, že najít nějakou knihovnu, která by byla využitelná pro systém G.A.T.E. a nezabývala se detekcí tváří, je velice obtížné. Přesto však existují některé výjimky, které proberu později. Co se týče detekce tváře, můžeme z tabulky vidět, že pouze 10% ze všech výsledků se zmiňuje o nějaké knihovně pro detekci.

Potřeba počítačového detekování tváří v obrázku se projevila například s nástupem biometrik. Předtím, než bylo možné aplikovat nějaký algoritmus pro rozpoznání tváře pro autentizaci pomocí biometrik, bylo nutné nejprve využít nějakou metodu detekce tváře k tomu, aby se zjistilo, zda se na obrázku vůbec nějaká tvář nachází. V minulosti bylo nutné vyřešit mnoho problémů jako např. různou velikost tváře, její pozici nebo osvětlení. V nedávné době přibyla také nutnost detekce tváří ve videu nebo v reálném čase snímaném záznamu digitálních kamerových systémů. Díky všem těmto problémům je však oblast rozpoznávání tváře na vskutku vysoké úrovni (nejvyšší, co se týče všech detekčních algoritmů) a i díky poznatkům v této oblasti se dosáhlo pokroku i u ostatních detekčních algoritmů nezabývajících se detekcí obličeje.

Většina detekčních metod v dnešní době využívá sadu tréninkových obrázků pro pozdější detekci obrázků jiných. Tyto tréninkové obrázky typicky zachycují obličej ve zvolené pozici (například zepředu, z profilu). Pro zmírnění efektu osvětlení na výsledek se také často využívá metoda vyrovnávání histogramu[2], nebo jiných metod standardizace. Z těchto obrázků je následně získána informace, která je zpracována počítačem nebo člověkem, a využita při detekci.

[2] z anglického „Histogram equalization“ - metoda upravující kontrast rastrového obrazu s využitím jeho histogramu (sloupkového diagramu).

Pro detekci tváří v obrázcích je v současné době možné využít mnoho technik a metod, založených na bázi pixelů, lokálních příznaků hran, Haarových vlnek, Haarovských příznaků (viz dále). Ať už zvolíme jakoukoliv techniku, největší vliv na úspěch detekce má správná volba tréninkových obrázků. Pozitivní příklady (tedy ty, na kterých se lidská tvář vyskytuje) musí být voleny skutečně pečlivě, aby bylo možné pro následnou detekci využít také zrcadlení, otáčení a měnění rozměru jednoho jediného obrázku. Volba příkladů negativních (jakékoliv jiné obrázky, na kterých se lidská tvář nevyskytuje) už tak pečlivá být nemusí. Přesto můžeme i touto volbou negativních příkladů ovlivnit výslednou efektivitu detekce (například tím, že vybereme do těchto negativních příkladů obrázky, které zachycují objekty, které se podobají lidským tvářím). [2]

Jednou z možností dalšího vývoje v oblasti detekce lidských tváří je nadále vylepšovat současné technologie, aby byly ještě efektivnější a dosahovaly lepších výsledků. „Další zajímavou myšlenkou je vylepšit výkon detekčních algoritmů pomocí dodatečné informace o kontextu obrázku. Lidská tvář je ve většině případů spojena s dalšími částmi lidského těla a právě tato informace by mohla posloužit jako výborné vodítko pro to, kde se může tvář nacházet. U obrázků, na kterých je relativně malá změna (díky špatnému odlišení pozadí a popředí, špatné ostrosti apod.), by mohla přinést znatelný pokrok adaptace[3]. Na rozdíl od ostatních oblastí, jako například rozpoznávání hlasu nebo rukopisu, adaptaci pro vizuální detekci objektů bylo zatím věnováno relativně málo pozornosti.“ [3]

[3] Adaptace je proces, při němž nějaký systém přizpůsobuje své chování jednotlivým uživatelům na základě informací, které o těchto uživatelích získal. [24]

2.1.1. Vlastní vektory a Eigenfaces

Vlastními vektory se v matematice označují nenulové vektory, které po určité transformaci (po vynásobení určitou maticí) zachovávají směr původního vektoru – mění se tedy pouze jejich délka. Každému vlastnímu vektoru proto přísluší vlastní číslo, které udává, jak se změnila právě tato velikost konkrétního vlastního vektoru. Vlastní vektory jsou někdy také označovány jako charakteristické vektory a jim příslušná charakteristická čísla.

Matematicky se tento vztah dá zapsat následovně. Máme danou matici A. Nenulový vektor v je vlastním vektorem matice A právě tehdy, když existuje skalární λ (lambda) taková, že platí:

$$Av = \lambda v$$

Lambda je v tomto případě vlastním číslem matice A závislosti na vektoru v. Dále můžeme definovat ještě vlastní prostor matice A jako množinu všech vlastních vektorů mající stejná vlastní čísla sjednocenou s nulovým vektorem (i když nulový vektor není vlastním vektorem).

Pro příklad, mějme matici $A = \begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix}$ a vektor $v = \begin{bmatrix} 3 \\ -3 \end{bmatrix}$, potom vektor v je vlastním vektorem identity matice A (s vlastním číslem 1).

$$Av = \begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix} \begin{bmatrix} 3 \\ -3 \end{bmatrix} = \begin{bmatrix} 2 \cdot 3 + 1 \cdot (-3) \\ 1 \cdot 3 + 2 \cdot (-3) \end{bmatrix} = \begin{bmatrix} 3 \\ -3 \end{bmatrix} = 1 \cdot \begin{bmatrix} 3 \\ -3 \end{bmatrix}$$

Naproti tomu, pokud budeme mít stejnou matici a vektor $v = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$, potom tento vektor není vlastním vektorem, protože:

$$Av = \begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 2 \cdot 0 + 1 \cdot 1 \\ 1 \cdot 0 + 2 \cdot 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \end{bmatrix} \neq \lambda \cdot \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

Eigenfaces je v podstatě množina vlastních vektorů (přičemž každý vektor reprezentuje jeden tréninkový obrázek – vektor vznikne konkatenací řádků pixelů původního obrázku), která se používá v počítačové grafice pro řešení problému detekování a rozpoznávání lidských tváří v obrázcích. Metoda založená na Eigenfaces je považována za vůbec první výrazněji použitelnou metodu pro rozpoznávání lidských obličejů. „Zmíněné vlastní vektory jsou odvozeny z matice kovariance rozdělení pravděpodobnosti mnohorozměrného vektorového prostoru možných lidských tváří“. [4]

Množina Eigenfaces je vytvořena pomocí matematického procesu analýzy hlavních komponent, známého pod zkratkou PCA (z anglického „principal component analysis“). Tato metoda slouží k dekorelaci dat a používá se například v případech, kdy máme velké množství nějakých dat a chceme tento počet co nejvíce zmenšit s co nejmenší ztrátou informace o původním souboru za pomocí statistických postupů. Takto můžeme například ze souboru vyloučit extrémní případy, které nemají na popis celého souboru velký vliv apod.

Po spuštění PCA na velkém množství obrázků zachycující lidské tváře dostaneme zredukovanou množinu, které se někdy říká množina „standardizovaných ingrediencí tváře“ (z anglického „standardized face ingredients“). Tento název tato množina nese z toho důvodu, že se pomocí jednotlivých prvků této množiny (jednotlivých eigenfaces) následně skládají aproximace jednotlivých obličejů. Jeden obličej tak může být složen z průměrné tváře celkového souboru, do které je přidáno např. 5% eigenface 1, 30% eigenface 2, nebo dokonce -20% eigenface 3. Tento postup je podobný jako při skládání určité barvy pomocí stupnice RGB, kdy také přidáváme nebo ubíráme sytosti jednotlivých barev RGB spektra a jejichž smícháním nám nakonec vznikne požadovaná barva (výborný příklad toho, kdy pomocí hodnot tří konkrétních prvků z široké množiny možností jsme schopni pokrýt celé jevové pole). I v případě lidských tváří se ukazuje, že pro aproximaci většiny lidských tváří je zapotřebí zkombinovat pouze poměrně malý počet jednotlivých eigenfaces.

Provádět PCA přímo na matici kovariance jednotlivých obrázků by bylo v mnoha případech výpočetně neproveditelné. Pokud bychom měli například obrázky ve stupních šedé o velikosti 100 x 100 pixelů, pak každý takový obrázek je bod v 10 000 rozměrném prostoru a matice kovariance tedy obsahuje 100 prvků. Tato matice je však ve většině případů omezena počtem tréninkových obrázků. Pokud máme n tréninkových obrázků a toto n je větší než rozměr těchto obrázků (tedy pokud v našem případě zvolíme alespoň 101 obrázků), pak jejich konkatenací vznikne nanejvýše n-1 vlastních vektorů s nenulovými vlastními čísly. V opačném případě (tedy když počet obrázků je menší než rozměr těchto obrázků) je možné hlavní komponenty identifikovat ještě jednodušeji. Díky této úvaze a pomocí úprav s násobením matic se například jsme schopni dostat do situace, kdy si při sadě 300 tréninkových obrázků vystačíme při PCA s maticí o velikosti 300 x 300 a nikoliv s úplnou maticí kovariance o velikosti 10 000 x 10 000. Přesný popis těchto úprav je možné nalézt na stránce v literatuře označené číslem [4].

Jak jsem již zmínil dříve, využití eigenfaces a vlastních vektorů bylo jednou z prvních použitelnějších metod pro detekování a rozpoznávání lidských tváří v digitálních obrázcích. Ve skutečnosti vznik a popis samotných eigenfaces byl motivován snahou o vytvoření metody pro detekování a rozpoznávání lidských tváří (nebylo to tedy tak, že eigenfaces existovaly již někdy dříve a byly pouze vybrány jako vhodný kandidát pro detekci). Konkrétně byly eigenfaces poprvé popsány v roce 1987 v publikaci Low-dimensional procedure for the characterization of human faces od autorů L. Siroviche a M. Kirbyho [5].

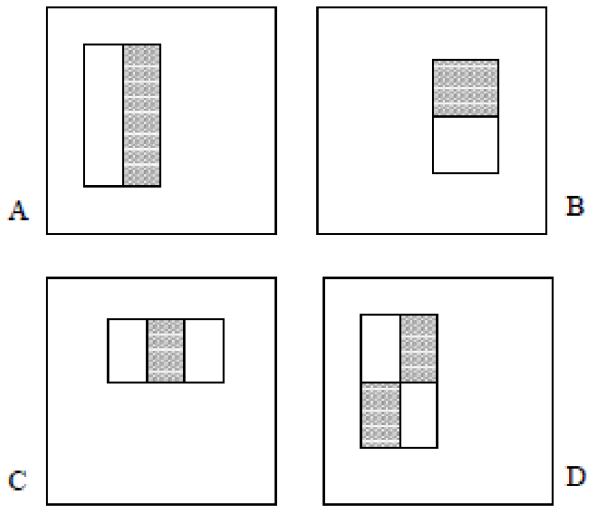
Mezi hlavní výhody této metody patří vysoká rychlost výpočtů a možnost práce nad mnoha tvářemi ve velice krátkém čase. Na druhou stranu však je již v dnešní době nedostačující v několika dalších charakteristikách, ve kterých dosahují jiné přístupy a metody daleko lepších výsledků, i když za cenu pomalejšího výpočtu. Mezi tyto hlavní nedostatky patří zejména detekce tváří při různorodých světelných podmínkách nebo různých úhlech pohledu na jednotlivé tváře. Pro

takovouto složitější detekci je potřebné využít daleko sofistikovanější mechanismy, než jednoduchou práci s konkatenací pixelů obrázku, vlastními vektory a násobením matic. Přesto jsou eigenfaces z historického hlediska velice důležité, protože dokázaly, že detekce lidských tváří v obrázcích je možná a lze s ní dosáhnout při volbě správných obrázků i relativně vysoké úspěšnosti detekce (až 96% při malých změnách v osvětlení). Bohužel v případě mé diplomové práce, pro systém G.A.T.E. si nemůžeme vybírat jen vhodné obrázky, ale naopak potřebujeme techniku, která vykazuje vysokou přesnost detekce nad jakýmkoliv obrázkem, a proto nepředpokládám, že použití eigenfaces bude tou správnou volbou. Přesto si myslím, že bylo vhodné popsat jeden z prvních přístupů k problematice detekce a rozpoznávání tváří, protože všechny ostatní metody byly touto metodou nějakým způsobem ovlivněny.

2.1.2. Haarovské příznaky

Haarovské příznaky (v originále „Haar-like features“; slovo feature, dále překládané jako příznak, označuje v oblasti zpracování obrazu část informace, která je použita k vyřešení určité výpočetní úlohy) jsou příznaky digitálního obrazu používané při rozpoznávání objektů. Haarovské příznaky byly poprvé popsány v roce 2001 v publikaci Rapid Object Detection using a Boosted Cascade of Simple Features [6] autory Paulem Violou a Michaelem Jonesem, kteří vylepšili tzv. Haarovu vlnku, což je první známá a nejjednodušší vlnka používaná jako vzorový příklad v oblasti teorie vlnek[4]. Základní nevýhodou Haarovy vlnky je, že funkce, která jí popisuje, není spojitá, a tudíž ji nelze derivovat.

Ve výše zmíněné publikaci se autoři zaměřují na problematiku rozpoznávání obličejů na základě hodnot jednoduchých příznaků v obrázku. Ve skutečnosti jsou při výpočtu použity tři typy příznaků podle toho, z kolika obdélníků se výpočet skládá (viz Obrázek 1).



Obrázek 1

[4] Oblast zabývající se rozkladem signálů a funkcí za pomocí vlnkové transformace, různými typy těchto transformací apod.

Nejprve je zvolena určitá oblast zkoumaného obrázku, ve které se poté počítají hodnoty jednotlivých příznaků. Hodnota dvou-obdélníkového příznaku se vypočítá jako rozdíl součtu intenzit pixelů v jednotlivých obdélnících, hodnota troj-obdélníkového příznaku jako součet intenzit pixelů ve dvou krajních obdélnících minus součet intenzit pixelů v prostředním obdélníku a hodnota čtyř-obdélníkového jako rozdíl součtů obdélníků na diagonálách. Tento výpočet tak slouží k formálnějšímu popisu obrázku a jeho částí právě na základě intenzit jednotlivých pixelů.

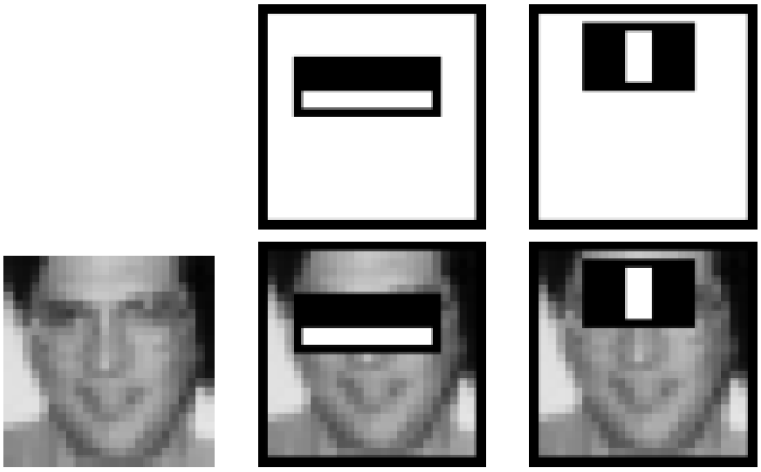
Pokud tedy známe pro určitý obrázek sadu příznaků a tréninkovou sadu vyhovujících a nevyhovujících obrázků, můžeme pomocí nějakého mechanizmu strojového učení [5] vytvořit klasifikační funkci, pomocí které lze poté určit vhodné příznaky pro rozpoznání obličeje. Problém je v tom, že pokud například rozdělíme obrázek do čtverců o velikosti 24x24 pixelů, vznikne pro každý takovýto čtverec sada čítající více než 180 000 příznaků. Proto je potřeba ještě dodatečný algoritmus pro výběr menší množiny příznaků. Metoda Haarovských příznaků využívá speciální typ klasifikátoru, tzv. kaskádovitý klasifikátor.

To, že je klasifikátor kaskádovitý, znamená, že výsledný klasifikátor je složen z několika jednodušších klasifikátorů. Díky tomu je možné dosáhnout jednak lepší výsledky detekce a jednak výrazně snížit čas výpočtu. Tento systém je založen na tom, že v prvním kroku výpočtu je použit jednoduchý klasifikátor, tedy takový, u kterého je zvolena velká hranice tolerance. V praxi se však ukazuje, že i takováto hranice je dostatečná k tomu, aby odfiltrovala převážnou většinu negativních případů ve velice krátkém čase. Pokud je v nějaké pozici část obrázku klasifikátorem přijata, přichází na řadu klasifikátor složitější, který má nastavenu nižší hranici tolerance. Takto, zjemňováním hranice, výpočet pokračuje do chvíle, než je tato část přijata a tedy rozpoznána, nebo odmítnuta.

[5] Podoblast umělé inteligence, zabývající se algoritmy a technikami, které umožňují počítačovému systému „učit se“. Učením v daném kontextu rozumíme takovou změnu vnitřního stavu systému, která zefektivní schopnost přizpůsobení se změnám okolního prostředí [25]

Odmítnutí jedním klasifikátorem samozřejmě znamená odmítnutí úplné a není na takovou část již aplikován klasifikátor další.[6]

Pro účely rozpoznávání obličejů na základě Haarovských příznaků byl zvolen algoritmus AdaBoost, díky kterému byly vybrány dva charakteristické typy příznaků (zdůvodnění toho, proč byly vybrány právě tyto dva příznaky společně s jejich testováním, lze nalézt ve výše zmíněné publikaci, v mé diplomové práci se již hlouběji tímto problémem zabývat nebudu). První příznak reflektuje skutečnost, že oblast očí je zpravidla na obrázku (převedeném do odstínů šedé) tmavší, než oblast tváří. Druhý příznak potom to, že opět oblast očí je tmavší, než kořen nosu (viz Obrázek 2). Samotné rozpoznávání funguje zjednodušeně tak, že je zvolena velikost referenčního čtverce, který je následně posunován po obrázku a v každé pozici se testuje, zda rozdíly intenzity pixelů v oblasti vymezené výše popsanými příznaky spadají do rozmezí určité stanovené hranice. Pokud ano, je tato oblast obrázku rozpoznána jako tvář, jejíž hranice jsou dány velikostí referenčního čtverce.



Obrázek 2

I když je tento přístup dozajista značně složitější, než je přístup za pomoci eigenfaces a vlastních vektorů, dosahuje stále velice vysoké efektivity výpočtu a přitom umožňuje detekci tváří, které by předchozí metoda jako tváře nerozpoznala.

Při prvních testováních Haarovských příznaků byl proveden pokus, který dokazuje efektivitu Haarovských příznaků, pokud jde o rychlost výpočtu. Klasifikátor o 38 vrstvách byl „naučen“ rozpoznávat tváře fotografované při pohledu čelně. Jako trénink pro klasifikátor byla zvolena sada 4916 obrázků lidských tváří. Tyto obrázky byly ručně označeny, zarovnány a zmenšeny na základní velikost 24 x 24 pixelů. Tyto obrázky byly náhodně staženy přímo z prostředí internetu. Stejným způsobem bylo získáno a označeno i 9544 obrázků neobsahující žádnou lidskou tvář.

Prvních pět úrovní klasifikátoru pracovalo postupně s 1, 10, 25, 25 a 50ti příznaky. Další vrstvy obsahují daleko vyšší počet příznaků, přičemž všechny vrstvy obsahují celkem 6061 příznaků. Každý klasifikátor byl trénován 9 832 obrázky obsahující tváře (4916 obrázků plus jejich vertikálně zrcadlené kopie) a 10 000 vybraných podoken z 9544 obrázků neobsahující tváře (celkem takovýchto podoken bylo přes 3,5 milionů).

Při testování bylo rozhodujícím faktorem, kolik příznaků muselo být následně aplikováno pro každé podokno jednotlivých testovacích obrázků. Výsledek byl takový, že v průměru je potřeba 10 příznaků z celkového počtu 6061 k vyhodnocení každého podokna. To je docíleno tím, že velká většina podoken může být automaticky zamítnuta aplikováním prvních dvou vrstev klasifikátoru. Další úrovně klasifikátoru se uplatní jen v tom případě, že bylo nalezeno vhodné podokno, které by mohlo obsahovat tvář a i z této malé množiny bylo zavrženo další množství podoken aplikováním několika následujících vrstev klasifikátoru.

V době, kdy byl tento test prezentován, byl prováděn na procesoru Pentium III s frekvencí 700 Mhz. Testem bylo dokázáno, že detekční proces nad obrázkem o velikosti 384 x 288 pixelů zabere v průměru 0,067 sekundy.

Datová množina testovacích obrázků zachycovala taktéž širokou škálu různých podmínek, za kterých mohou být fotografie pořizovány. Mezi některé z těchto podmínek patří například změna osvětlení, velikosti, pozice tváře apod. Samotní autoři přiznávají, že jejich záměrem nebylo prezentovat chování této metody pouze při pevně zvolených podmínkách, protože v reálném prostředí také jen zřídkakdy požadujeme algoritmus, který funguje například jen v jednom konkrétním prostředí.

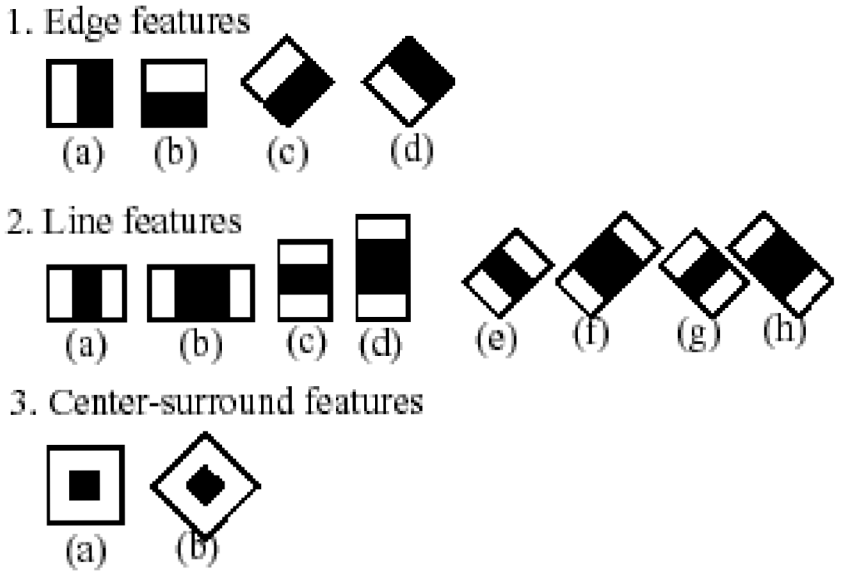
2.1.3. OpenCV

OpenCV (Open Source Computer Vision Library) je knihovna, která obsahuje několik stovek algoritmů zabývajících se počítačovým viděním (z anglického „computer vision“). Tato knihovna je volně dostupná jak pro akademické, tak komerční využití, přičemž spadá pod licenci BSD[6]. OpenCV má modulární strukturu, což znamená, že se skládá z několika menších sdílených statických knihoven. Základní částí je modul core, který definuje datové struktury a funkce používané všemi ostatními moduly. Mezi další moduly patří například modul video (sloužící k analýze videa), imgproc (podporující nelineární filtrování obrazu, obrazové transformace) nebo gpu (obsahující GPU algoritmy, které využívají ostatní moduly).

Pro potřeby mé diplomové práce je však nejdůležitější modul objdetect, který slouží k detekci objektů předem definované třídy [7]. Detekce objektů je prováděna za pomocí kaskádovitého klasifikátoru využívajícího Haarovské příznaky. V prvním kroku výpočtu je nejprve nutné „naučit“ klasifikátor rozpoznávat v obrázku určitý objekt (v našem případě lidskou tvář). Toto se děje za využití několika set cvičných obrázků stejné velikosti, a to jak těch, na kterých se zkoumaný objekt nachází, tak i těch, na kterých se nenachází. Stejně jako u Haarovských příznaků, tak i zde je pro výběr klasifikátoru použit algoritmus AdaBoost, nebo jeho varianta jako např. Real Adaboost (pracující s reálnými čísly). Poté je zvolena velikost rámce, který je posunován, opět podobně jako u Haarovských příznaků, po celém zkoumaném obrázku a v každé pozici se testuje, zda se v rámci nachází zkoumaný objekt. Klasifikátor je volen tak, aby bylo snadné ho podle potřeby zvětšovat a zmenšovat a hledat tak vyhovující objekty různé velikosti (tato metoda je rychlejší, než měnit velikost samotného zkoumaného obrázku). [8]

[6] Brekeley Software Distribution, nebo také Berkeley Unix je operační systém z rodiny Unix

V současné době knihovna OpenCV využívá následující Haarovské příznaky (Obrázek 3):



Obrázek 3

2.1.4. Betaface

Betaface Software je německá firma angažující se v oblasti rozpoznávání obličejů. „Cílem této organizace je nabídnout uživatelům nové možnosti navigace a kategorizace nepřeberného množství vizuálních dat, která se každý den objevují na internetu. Jako další cíle si klade eliminaci neefektivních přístupů při hledání neznámých i známých osob ve vizuálních datech, zvýšení prohledávací rychlosti a možnost předem zpracovat úložiště těchto dat pomocí automatizovaných procesů.“ [9]

Mezi základní služby firmy Beatface patří automatizované indexování obrázků a videa, díky čemuž je možné zrychlit následné vyhledávání v takto indexovaných datech, extrakci podob jednotlivých lidí ve vizuálních datech společně s jejich dalšími biometrickými údaji, rozpoznání řeči a textu a další. V budoucnu plánuje také návrh otevřeného webového rozhraní pro snadnější vytváření aplikací s podporou Betaface. Mimo jiné také nabízí možnost získání licence pro Betaface Face Detection and Recognition SDK[7].

„Betaface Face Detection and Recognition SDK je knihovna, obsahující sadu algoritmů vytrénovaných a optimalizovaných pro detekci lidských tváří a některých dalších rysů tváře ve statických obrázcích a videích. Tyto algoritmy nevyžadují k práci barevné vizuální data, protože vnitřně pracují s hodnotami pixelů v odstínech šedé.

SDK detekuje čelně vyobrazené tváře (s jistou tolerancí v rotaci hlavy) v jakémkoliv úhlu a vrací souřadnice všech tváří spolu se všemi detekovanými rysy. SDK také umožňuje vyříznout tváře z obrázku na základě detekovaných informací, tyto tváře porovnávat na základě jistého hodnocení, měnit jejich tvar, deformovat je nebo dokonce skládat.

[7] Z anglického „Software Development Kit“ – sada vývojových nástrojů umožňujících tvorbu aplikací pro určité omezené softwarové balíky (operační systémy, informační systémy apod.)

Jádro SDK tvoří chráněná DLL knihovna pro 32bitový systém Windows. Dále balík obsahuje nezbytné spustitelné soubory, rozhraní pro C++, COM a .NET, dokumentaci a příklady použití napsané v C++ a C#.“ [10]

Jak již bylo zmíněno výše, technologie Betaface neumožňuje pouze detekovat tvář v obrázku, ale můžeme pomocí ní získat ještě další informace o rysu tváře. Mezi tyto rysy patří například barva vlasů, zda má dotyčný člověk na obličeji brýle, zda se usmívá, umožňuje dokonce s jistou mírou pravděpodobnosti také určit, jaká je jeho etnická příslušnost a kolik je mu let. Všechny tyto informace jsou následně použitelné právě při porovnávání tváří a jejich kategorizaci.

Na oficiálních stránkách Betaface je v současné době dostupná online demoverze společně se zkušebním webovým klientem, který obsahuje zdrojový kód v C# a .Net 4.0. Tato demoverze detekuje tváře v obrázcích, umožňuje jejich vyřezání z obrázku, poskytuje základní informace k porovnávání a také některé z dodatečných informací o osobě na obrázku. Na těchto stránkách jsou také dostupné dvě sady licenčních klíčů pro nekomerční, zkušební a výukové produkty. Jedna sada slouží pro přístup k základním funkcím klienta a druhá pro přístup k funkcím rozšířeným, jako je zmíněná barva vlasů apod. Dvě sady klíčů jsou uvedeny zajisté především proto, že výpočetní čas detekce je v případě využití rozšiřujících funkcí vyšší, než pokud využijeme pouze funkce základní.

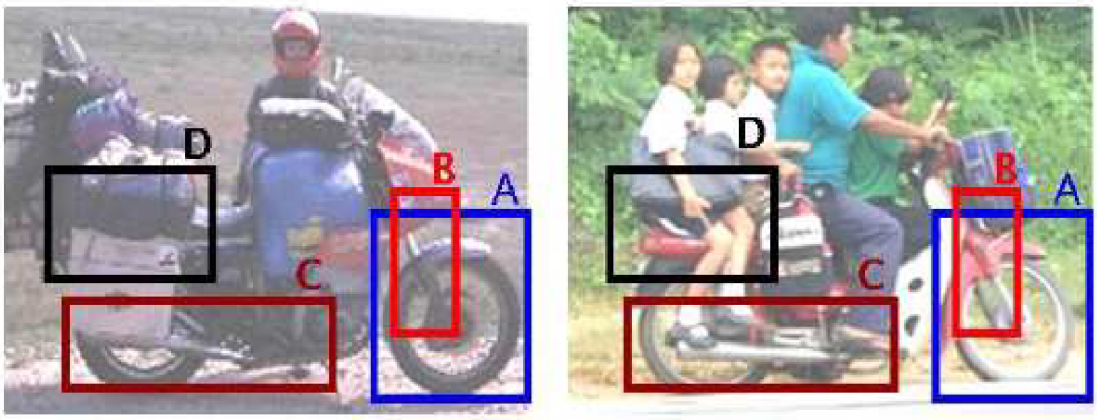
2.2. Detekce ostatních objektů

Jak jsem již zmínil dříve, najít knihovny pro detekci jiných objektů, než jsou lidské tváře, není jednoduché, zejména kvůli rozličnosti jednotlivých objektů spadajících do stejné kategorie (například v případě detekce lidské postavy může být jeden člověk vyšší, jeden nižší, mohou každý zaujímat jinou pózu na obrázku apod.). Pokud jsem již nějakou knihovnu objevil, musel jsem také přihlédnout k tomu, do jaké míry by byly využitelné pro systém G.A.T.E. Toto kritérium ještě dále omezilo již tak velice omezené možnosti výběru. Například jsem objevil knihovnu (respektive konkrétní detektor této knihovny) vytrénovanou na detekci hlav koček, která vykazovala při testování relativně vysokou přesnost, ale která zároveň bezesporu pro systém G.A.T.E. příliš využitelná není.

Pro mou další práci jsem nakonec zvolil open source implementaci algoritmu pro detekci objektů, kterou na svých stránkách prezentuje Ivan Laptev, který v současnosti pracuje ve výzkumném centru Inria Paris – Rocquencourt. Toto výzkumné centrum je součástí Francouzského národního institutu pro výzkum počítačových věd a řízení a v současné době se specializuje na sociální a ekonomický přínos v různorodých problémech dnešní doby, posiluje a rozšiřuje spolupráci s ústavy zabývajícími se jinými vědními obory a obchodními partnery ve Francii, zbytku Evropy, v USA a v dalších zemích, jako je například Čína, Indie a rozšiřuje působiště i v Jižní Americe a Africe. [11]

Práce Ivana Lapteva je motivována úspěšností metod, které pro reprezentaci obrázku používají jeho histogram. Díky histogramu totiž můžeme vypozorovat jistou shodu určitých částí zkoumaného typu objektu, i když konkrétní jednotlivé objekty budou mít tyto části každý poněkud odlišný. Popisná síla histogramů výrazně závisí na dvou faktorech. Je to jednak na tom, jaké hodnoty obrázku bude znázorňovat (jas, odstín, sytost, atd.) a jednak samozřejmě na tom, které části objektu budeme histogramem měřit. Co se týče problému měření hodnot, algoritmus Ivana Lapteva využívá

histogram orientovaných gradientů[8]. Pokud jde o to, jaké části objektu měřit, je nejkritičtější najít takové části, které se příliš nemění v širokém spektru konkrétních jednotlivých objektů (viz Obrázek 4).



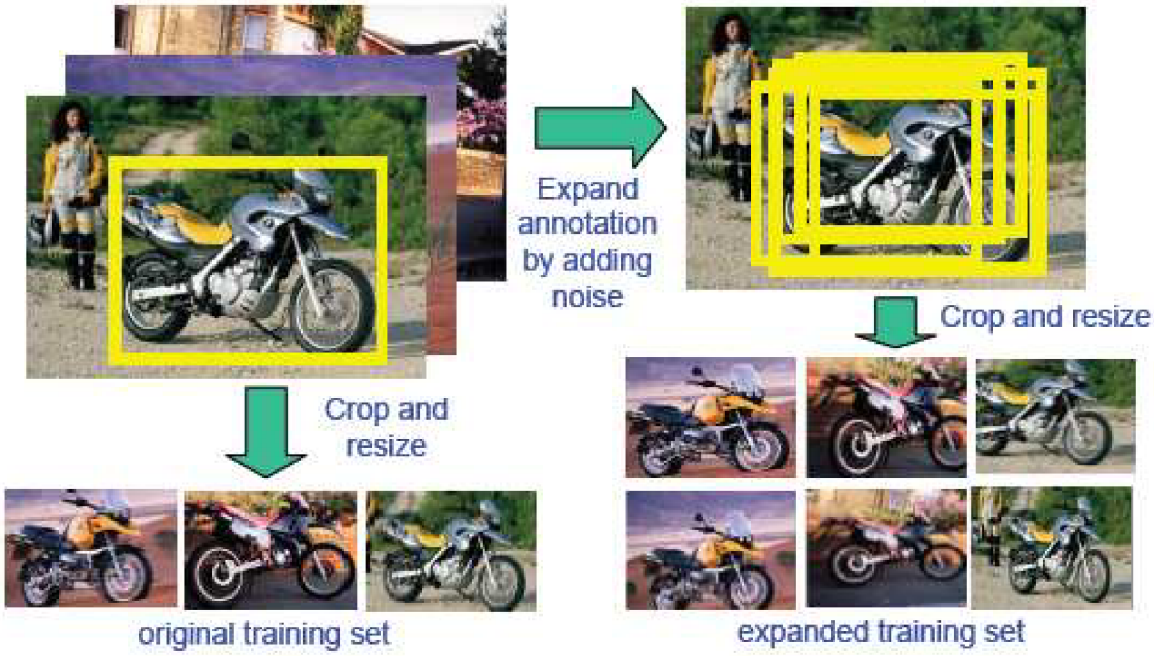
Obrázek 4

Obdélníky na obrázku jsou příklady možných pozic měření histogramu. Objekty v obdélnících A, B a C mají relativně podobný vzhled, a proto jsou dobrými kandidáty pro detekci motocyklu. Oblast D se ovšem nezdá být příliš spolehlivou pro rozpoznávání, protože se vzhled objektu v této oblasti výrazně liší. [12]

Při procesu tréninku algoritmu se předpokládalo, že byly použity vzorové obrázky, na kterých pozice jednotlivých částí zkoumaného objektu zhruba odpovídaly zvoleným obdélníkovým oblastem. U sady pozitivních případů byly použity normalizované (pozičně a velikostně) obrázky zvoleného objektu a poté pomocí metody procházení okna obrázku zkoumána podokna všech možných tvarů a velikostí. Kromě pevně zvolené sady negativních případů byly jako negativní případy v průběhu tréninku použity také ty, které vznikly chybným detekováním pozitivních případů. Navíc jeden pozitivní tréninkový obrázek byl pro zlepšení přesnosti ještě pomocí malých transformací využit

[8] Metoda počítá jednotlivé výskyty orientace gradientu (směru stoupání nebo klesání odstínů jednotlivých barev) v určité části obrázku

několikrát, čímž byla ještě rozšířena testovací množina. Mezi tyto transformace patřila například změna velikosti, nebo ořezání (viz Obrázek 5).



Obrázek 5

Tímto procesem vznikne za pomocí algoritmu prezentovaného Ivanem Laptevem detektor zvoleného typu objektů. Tento detektor je ve formě XML souboru a lze ho následně využít pomocí open source knihoven dostupných na webových stránkách Ivana Lapteva [13] k detekci těchto objektů v libovolném obrázku. V publikaci Improving object detection with booster histograms [12] jsou uváděny většinou příklady s detektory motocyklů, automobilů a lidských postav. Mimo tyto jsou na zmíněných stránkách dostupné ještě další detektory (šlapacích kol, koňů, koček), které však nepovažuji pro systém G.A.T.E. za příliš využitelné.

Ve své diplomové práci tedy využiji volně stažitelnou implementaci Laptevova algoritmu založeného na detekci pomocí histogramů ve spojení s detektorem lidských postav, otestuji jeho funkčnost a rozeberu jeho výhody a nedostatky.

2.3. Optické rozpoznávání znaků

„Optickým rozpoznáváním znaků, obvykle zkracované jako OCR (z anglického „optical character recognition“), se rozumí mechanický nebo elektronický převod naskenované obrazové informace zachycující ručně psaný nebo strojově tištěný text do elektronicky kódovaného textu.“ [14] Tyto metody již ve své podstatě, jak již samotný název napovídá, nespadají do dosud diskutované oblasti detekce objektů, ale do oblasti rozpoznávání přesně s těmi rozdíly, které byly popsány v úvodu mé práce. Při optickém rozpoznávání znaků se totiž ve většině případů již nesnažíme o detekci toho, zda se na obrázku vyskytuje nějaký text (i když i takové algoritmy mají své místo), ale spíše hledáme konkrétní výskyt konkrétního písmene v tomto textu.

Problémy, které jsou s touto oblastí spojeny, tkví především v tom, že v případě ručně psaného textu má samozřejmě každý člověk jiný rukopis. Tento problém se dá v současné době velice těžko překonávat jinak, než že se zvolený software určený k rozpoznávání opět nejprve „naučí“ rukopis uživatele a díky tomu je poté schopen rozpoznat text psaný jeho rukou. Pokud jde o strojově nebo počítačem sázený text, je problém v podstatě velice obdobný. V současné době se v každém textovém editoru dá zvolit nepřeberné množství různých fontů. I když je jich podstatně méně, než kolik je různých možných rukopisů, stejně je již velice obtížné, možná i nemožné, říct, jakou podobu mají jednotlivá písmena. Proto i systémy pro rozpoznávání takto sázeného textu potřebují jistou kalibraci, aby byly schopné rozpoznávat jednotlivé fonty. „V minulosti takové systémy musely být programovány pomocí obrázků jednotlivých písmen zvoleného fontu a pracovaly většinou jen s fontem jedním. Nynější systémy jsou již schopny rozpoznávat většinu fontů s docela vysokou mírou přesnosti. Některé systémy jsou schopné reprodukovat vstupní skenovaný dokument s takovou přesností, že dodrží i formátování textu, a to i netextových komponent, jako jsou například obrázky.“ [14] To se však samozřejmě týká pouze nejsofistikovanějších komerčních produktů.

Historie optického rozpoznávání znaků sahá do roku 1929, kdy německý vědec Gustav Tauschek obdržel na tento typ rozpoznávání patent. Jeho stroj byl čistě mechanický a využíval sadu vzorů jednotlivých písmen a detektor záření. I když od této doby uplynulo již více než osmdesát let a mohlo by se tedy zdát, že je tato oblast již velmi dobře prozkoumána, opak je pravdou. „Rozpoznávání textu psaného latinkou není stále stoprocentní, i když je vstupní obraz naprosto jasný a čistý. Studie založená na novinových stránkách z 19. a 20. století dokazuje, že rozpoznávání metodou písmeno po písmenu dosahuje přesnosti v rozmezí od 71% do 98% a že úplné přesnosti lze dosáhnout pouze po revizi člověkem. Oblasti jako je rozpoznávání ručně psaného textu nebo rozpoznávání jiných písem (zvláště Východní Asie) jsou stále předmětem aktivního výzkumu.“ [14]

Předpokládá se, že vyšší pravděpodobnost úspěchu rozpoznávání nebude do budoucna možná bez použití dodatečné gramatické informace, nebo informace o kontextu. S touto informací bude například možné dopředu určit, jaká písmena jsou možná pokračování právě zkoumaného slova, což výrazně omezí slovník, ze kterého je třeba zvolit to správné písmeno, a tudíž přispěje ke zvýšení rozpoznávacích schopností. To, že není optické rozpoznávání znaků stále na tak vysoké úrovni, na druhou stranu umožňuje použití některých bezpečnostních mechanismů, které by v případě stoprocentní úspěšnosti rozpoznávání textu v obrázku nemohly fungovat. Typickým příkladem tohoto faktu je Turingův test na rozlišení uživatelů od robotů CAPTCHA.

2.3.1. GOCR

„GOCR je program pro optické rozpoznávání znaků vyvíjený pod veřejnou licencí GNU[9]. Dokáže konvertovat naskenované obrázky textu zpět do textových souborů. Tento projekt spustil kolem roku 2000 Joerg Schulenburg, který v současnosti vede tým vývojářů. GOCR může být provozován na různých koncových zařízeních, což ulehčuje jeho přenos na jiné operační systémy a architektury.“ [15] GOCR může být použito jako plnohodnotná aplikace pro příkazový řádek, tak také jako podpůrný software pro jiné programy.

GOCR nativně pracuje s bezpatkovými písmy velikosti 20-60 pixelů. Bohužel má velké problémy s rozpoznáváním čehokoliv jiného. To zahrnuje jiná písma – patková, písma s překrývajícími se znaky, ručně psaný text, obrázky s vyšším stupněm šumu nebo, tak jako většina systémů zabývajících se optickým rozpoznáváním znaků, jinak než latinkou psaný text.

Základním formátem obrázků, který GOCR podporuje, je vlastně celá rodina formátů abstrahovaná zkratkou PNM. Tato zkratka je odvozena z anglického „Portable Any Map“, přičemž slovo Any je nahrazováno podle toho, jaký konkrétní formát z této rodiny využijeme. Mezi tyto formáty patří PBM (Portable Bit Map), PGM (Potrtable Gray Map), PPM (Portable Pixel Map) a PAM (Portable Arbitraty Map).

Všechny tyto formáty jsou používány především ve spojení s balíkem grafických knihoven a programů Netpbm. „Programy Netpbm jsou veskrze spouštěny z příkazové řádky, ale mohou být také využívány jinými programy. Jejich společnou charakteristikou je, že jsou to skutečně velice malé programy, stavební kameny.“ [16] Všechny formáty Netpbm jsou rastrové, to znamená, že popisují každý obrázek jako pixelovou matici. V případě PBM jsou tyto pixely černé a bílé, v případě PGM

[9] GNU je svobodný UNIX-like (podobající se Unixu, ale neobsahující žádný originální kód Unixu) operační systém

v odstínech šedé a v PPM v plném barevném spektru. Formát PAM umožňuje ukládat data jak černobílá, šedá, tak barevná.

Tuto metodu, stejně tak jako oblast optického rozpoznávání znaků, uvádím spíše pro doplnění toho, jaké jiné možnosti detekce a rozpoznávání jsou v současné době dostupné. Nepředpokládám však, že tuto metodu využiji pro mou výslednou aplikaci. Stojí však podle mého za zmínku a také je to minimálně jeden z kandidátů na možné vylepšení mnou navržené aplikace.

3. Java Beans

„Technologie JavaBeans je komponentová architektura pro Java 2 Platform, Standard Edition. Komponenty JavaBeans jsou znovupoužitelné softwarové programy, které jsou po vytvoření snadno propojitelné a slouží tak k vytváření sofistikovaných aplikací.“ [17] Odkaz na současnou specifikaci (verzi 1.01) je možné najít v seznamu použité literatury. [18]

Oficiální definice této komponenty je následující: „Java Bean je znovupoužitelná softwarová komponenta, se kterou lze vizuálně manipulovat ve vývojovém prostředí.“ [18] Tímto vývojovým prostředím může být myšlena široká paleta nástrojů od nástrojů pro webové stránky, grafické uživatelské rozhraní, serverové aplikace atd. Některé z těchto nástrojů mohou být výlučně založeny na grafickém uživatelském rozhraní, jiné mohou umožňovat skrze textové rozhraní manipulovat s jednotlivými beany.

I samotné komponenty Java Beans se mohou radikálně lišit svou podobou a nabízející funkcionalitou. Co se týče vzhledu, některé z nich mohou být jednoduché grafické elementy (tlačítka), některé mohou plnit daleko složitější funkce, jako například prohlížeče databáze. Některé z beanů nemusí mít vůbec žádnou grafickou podobu, ale přesto se s nimi dá jako s beany manipulovat skrze vývojové prostředí.

Jak bylo výše zmíněno, každá komponenta se může výrazně lišit i svou funkcionalitou, přesto by každá z nich měla mít určité typické znaky společné pro všechny komponenty. Mezi tyto znaky patří podpora:

- introspekce (dovoluje vývojovému nástroji zanalyzovat, jak komponenta pracuje)
- customizace (dovoluje programátorovi měnit vzhled a do jisté míry i ovlivňovat chování komponenty)

- událostí (jako nástroj pro komunikaci mezi jednotlivými komponentami)
- vlastností (každá komponenta má své nastavitelné vlastnosti, což se využívá jak při programování, tak při customizaci komponent)
- stálosti (vlastnosti komponenty mohou být nastaveny pomocí vývojového nástroje a poté uloženy pro pozdější použití)

Obecně by se dalo říci, že komponenta Java Bean je taková komponenta, se kterou může být vizuálně manipulováno a které lze nastavovat různé parametry k dosažení určitého efektu.

Ne všechny části aplikace využívající technologii Java Beans musí být nutně složeny z jednotlivých beanů. Typicky využíváme beany pro takovou část aplikace, která nese určitou funkcionalitu, a zároveň mají nějakou grafickou podobu. Pro všechny ostatní části aplikace, které pouze poskytují funkcionalitu, využíváme třídy, které známe z jednoduché Javy. Pro příklad při psaní aplikace pro získávání dat z databáze využijeme třídy pro algoritmy obsluhující databázi a bean, který uživateli umožní zadávat prostřednictvím uživatelského rozhraní dotazy a který tyto dotazy předá třídám.

Mezi tři základní kameny Java Beanu patří vlastnosti, metody a události. Vlastnosti jsou atributy, které mohou být čteny a měněny pomocí metod a které definují vzhled a vlastnosti jednotlivých beanů. Metody jsou obdobou metod, jak je známe z Javy. Mohou být volány z ostatních komponent. Mezi typické příklady metod jsou metody set a get, které získávají a nastavují hodnoty jednotlivých vlastností. Události slouží k upozornění ostatních komponent na důležité změny. To se děje za pomoci naslouchání, zda se v systému objevila nějaká událost, a pokud ano, vyvoláním příslušné metody.

Vzhledem k tomu, že systém G.A.T.E. by měl nevidomým poskytovat informace o vybraném obrázku přes webové rozhraní, je volba technologie Java Beans velice příznivá, protože její

architektura byla od počátku navrhována tak, aby efektivně zpracovávala požadavky v prostředí internetu a distribuovaných systémů. V současné době tkví největší problém ve zpoždění a dalších komunikačních problémech při volání vzdáleného serveru. Proto je nutné navrhovat rozhraní, které komunikují přes síť velice opatrně, tak aby co nejméně síť zatěžovala a dokonce se snažit o to, minimalizovat celkovou komunikaci mezi jednotlivými uzly v síti.

Pro přístup k síti Java Beans využívají zejména mechanismus Java RMI. „Java Remote Method Invocation umožňuje snadný vývoj distribuovaných aplikací. Umožňuje návrh rozhraní distribuovaných systémů pomocí programovacího jazyka Java a implementace klientů a serverů může být realizována právě na základě těchto rozhraní. Jakékoliv volání metod Java RMI je automaticky a transparentně doručeno od klienta k serveru. Očekává se, že použití tohoto mechanismu bude v budoucnu nejpoužívanějším přístupem při psaní programů využívající Java Beans v síťovém prostředí.“ [18]

Dalším použitelným mechanismem je Java IDL, který pracuje na základě standardu OMG CORBA, což umožňuje Java Beans klientům pracovat s IDL (Interface description language) servery[10]. Poslední možností je využití databázového rozhraní JDBC, díky kterým lze také komunikovat s SQL servery.

Použití Java Beans nyní demonstruji na jednoduchém přikladu. Tento příklad využívá beanů k vytvoření appletu[11] s prohlížečem databáze. Při psaní kódu appletu programátor nejprve vyvolá inicializační metodu, ve které vytvoří tlačítko (dave.Button) a prohlížeč databáze (dave.DatabaseViewer).

---

[10] Servery založené na jazyce IDL – tento jazyk umožňuje komunikaci mezi softwarovými komponentami, které nesdílí společný programovací jazyk, tedy například mezi komponentami navrženými v Javě a v C++

[11] Applet je softwarová komponenta, která běží v kontextu jiného programu. Slouží k plnění funkce v daném kontextu a nepředpokládá se, že bude používán jako samostatná aplikace. [26]

```
class MyApplet extends java.awt.Applet {

public void init() {
        try {
                ClassLoader cl = this.getClass().getClassLoader();
                btn = (Button)Beans.instantiate(cl, “dave.Button”);
                add(btn);
                dbViewer = (DatabaseViewer)Beans.instantiate(cl,“dave.DatabaseViewer”);
                add(dbViewer);
        } catch (Exception ex) {
                throw new Error(“Couldn’t create bean “ + ex);
        }
}
        private DatabaseViewer dbViewer;
        private Button btn;
}
```

Pokud se rozhodneme, že chceme změnit vzhled nebo chování některé z komponent, musíme zavolat některou z metod sloužících ke změně vlastností zvolené komponenty. V následující části kódu nejprve upravíme barvu pozadí tlačítka a poté zadáme cestu k databázi, kterou chceme zobrazit.

```
btn.setBackground(java.awt.Color.red);
dbViewer.setURL(“jdbc:msql://aardvark.eng/stuff”);
```

Při propojování jednotlivých komponent využijeme výše zmíněných událostí. Nejprve naprogramujeme, jakou činnost chceme, aby zvolená událost způsobila. V našem případě je touto činností obnovení databáze.

```
void actionPerformed(java.awt.event.ActionEvent) {
        dbViewer.update();// prod the DB viewer.
}
```

Následně spojíme tuto událost s tlačítkem, které událost vyvolá.

```
btn.addActionListener(this);
```

Tímto je příklad u konce. Po přidání appletu na html stránku je zavolána inicializační metoda. Tato metoda vytvoří tlačítko a prohlížeč databáze, zavolá metody, které nastaví vlastnosti jednotlivých komponent a začne naslouchat, zda se vyskytne událost, která by způsobila obnovení databáze. Pro úplnost následuje kompletní kód využívající Java Beans.

```
class MyApplet extends java.awt.Applet implements
java.awt.event.ActionListener {

// Inicializace appletu
public void init() {
        try {
                ClassLoader cl = this.getClass().getClassLoader();
                btn = (Button)Beans.instantiate(cl, “dave.Button”);
                add(btn);
                dbViewer = (DatabaseViewer)Beans.instantiate(cl,“dave.DatabaseViewer”);
                add(dbViewer);
```

```
        } catch (Exception ex) {
                throw new Error("Couldn't make bean " + ex);
        }

        btn.setBackground(java.awt.Color.Red);
        dbViewer.setURL("jdbc:msql://aardvark.eng/stuff");

        // Přiřazení události tlačítku
        btn.addActionListener(this);
        }

        // Metoda zachytávající události tlačítka
        void actionPerformed(java.awt.event.ActionEvent evt) {
        dbViewer.update(); // prod the DB viewer.
        }

        private DatabaseViewer dbViewer;
        private Button btn;
}
```

4. faint

Faint (The Face Annotation Interface) je framework[12] pro programovací jazyk Java. Projekt faint byl vytvořen a následně vyvíjen Maltem Mathiszigem na Oldenburské univerzitě v Německu. Faint se zabývá detekcí tváří a obecně technologiemi tohoto rozpoznávání, založených na různých typech pluginů a filtrů. Obsahuje také grafické rozhraní, které může být použito pro vytváření pipelin[13] vhodným kombinováním těchto pluginů a filtrů.

V současné verzi aplikace jsou dostupné následující moduly pro rozpoznávání: OpenCV-Haarclassifier-Detection (rozpoznávání pomocí grafické knihovny OpenCV tak, jak bylo popsáno výše), Betaface.com-Detection (rozpoznávání pomocí funkcí služby organizace Betaface), Skin-Color-Filter (využívající osmikilobytové tabulky obsahující různé odstíny a sytosti), Eigenface-Recognition (Java implementace Eigenfaces přístupu k rozpoznávání) a Simple-Context-Filter (filtr, který rozpoznává tváře a následně zabraňuje vícenásobnému výskytu jednoho člověka na jedné fotografii).

Detekované tváře jsou ukládány v lokální databázi, která může být modifikována přímo pomocí aplikace faint. Všechny anotace tváří mohou být navíc uloženy přímo do souborů obrázků ve formátu Adobe XMP.

Pro použití tohoto frameworku ve vyvíjených aplikacích stačí využít JAR soubor jako Java knihovnu. Tento soubor je volně dostupný na stránkách projektu [19]. Úplný zdrojový kód je dostupný skrze repositář Subversion[14] na stránkách SourceForge [20]. I když by bylo samozřejmě daleko jednodušší využít JAR soubor a z vyvíjeného programu pouze volat jeho metody, tak jako většina z takto volně dostupných modulů, s kterými jsem se za svou praxi setkal, i tento trpí jedním

[12] Framework je softwarová struktura, která slouží jako podpora při programování, vývoji a organizaci jiných softwarových projektů. Může obsahovat podpůrné programy, knihovny, atd.
[13] http pipelining je technika, při které je na server odesláno několik požadavků zároveň a nečeká se tudíž na odpovědi na jednotlivé požadavky
[14] Subversion je systém pro správu a verzování zdrojových kódů

zcela zásadním nedostatkem. Tímto nedostatkem je zcela chybějící dokumentace. Samotný JAR soubor obsahuje ve svém archivu pouze soubory typu class (soubory zkompilované pro virtuální stroj Javy), ze kterých samozřejmě nelze vyčíst, jak ve skutečnosti pracují a jak mezi sebou komunikují. Neobsahuje žádnou dokumentaci a při volání jeho metod z jiného programu jsou viditelné pouze názvy a typy jednotlivých parametrů těchto metod a případně výstupní typy. Z toho se bohužel dá jen velice těžko určit, jak mají být tyto metody správně používány a k čemu vlastně slouží.

Naštěstí je pro faint dostupný ještě výše zmíněný druhý zdroj na stránkách SourceForge. Tento zdroj obsahuje 318 MB velký adresář obsahující kompletní zdrojový kód faintu a právě nejdůležitější části tohoto zdrojového kódu pro detekci lidských tváří proberu v následující části mé práce.

4.1. Struktura faintu

Nejdůležitější součástí archivu aplikace faint je balík zdrojových souborů nacházejících se v adresáři de/offis/faint. Pro mě nejdůležitější součástí tohoto balíku jsou zdrojové soubory, které se nacházejí v podadresáři detection/plugins/opencv. V tomto adresáři se nachází soubory, které zprostředkovávají detekci tváří pomocí knihoven OpenCV a také několik různých klasifikátorů, které tuto detekci umožňují, každý s trochu jinými parametry. V následujících částech mé diplomové práce se proto zaměřím na tyto klasifikátory a otestuji jejich funkčnost.

Dále se v archivu nacházejí zmíněné pluginy pro detekci a filtry, třídy nastavující globální parametry detekce a třídy s grafickým uživatelským rozhraním. Pro mou práci je důležitá ještě třída Region, nacházející se v balíku de/offis/faint/model. Tuto třídu využívá detekce k uložení informací o jednotlivých detekovaných oblastech v obrázku. Já tuto třídu využiji nejen pro detekci tváří, ale budu si díky ní schopen ukládat informace i o ostatních detekovaných oblastech, jejichž detekci již pomocí faintu provádět nebudu. Zároveň také budu mít všechna data díky této třídě uložena jednotně v jedné datové struktuře.

Co se týče samotné třídy zprostředkovávající detekci, její hlavní součástí je metoda public Region[] detectFaces(String file, int scanWindowSize), která si vezme cestu k obrázku zadanou prostřednictvím jejího prvního parametru a na tomto obrázku spustí detekci a vrátí seznam detekovaných oblastí ve formě pole výše zmíněných objektů typu Region.

Pro fungování detekce je zapotřebí této detekční metodě zadat ještě jeden parametr. Tento parametr scanWindowSize je ve skutečnosti kromě změny klasifikátoru jedna ze dvou hodnot, díky které můžeme upravovat fungování samotné detekce a jejích výsledků. Druhá taková hodnota je uložena v atributu scaleFactor a lze ji změnit pomocí příslušných metod detekční třídy. Parametr

scanWindowSize udává, jaká bude počáteční velikost okna (neboli čtverce), které bude následně posouváno po obrázku a v každé jeho pozici se bude testovat, zda obsahuje, nebo neobsahuje tvář. Atribut scaleFactor (dále faktor zvětšování okna) obsahuje hodnotu ve formě reálného čísla a udává, o kolik procent se má zvětšit toto okno před dalším průchodem detekčním algoritmem. Pokud tedy například zvolíme velikost okna 10 a faktor zvětšování okna 1,1, pak v druhém průchodu bude mít okno velikost 11, v dalším 12 a tak dále, dokud nepřesáhne velikost obrázku, ve kterém provádíme detekci. Tyto dva parametry spolu s klasifikátorem budou také předmětem testování v následujících kapitolách mé diplomové práce.

4.2. Změny ve faintu

Ještě předtím, než uvedu testy a jejich výsledky, musím v krátkosti popsat změny, které jsem udělal v samotném faintu z důvodu funkčnosti (čistě stažený archiv faintu obsahoval některé chyby, například z důvodu neaktuálnosti adresy služby betaface.com apod.) nebo z důvodu přizpůsobení některých vlastností mým potřebám.

Pro bezchybné fungování faintu jsem byl nucen provést hlavně změny ve třídě de.offis.faint.model.ImageModel. Tato třída obsahovala mnoho importů (ze skupiny com.drew.imaging a com.drew.metadata), které hlásily chybu spojenou s tím, že tyto balíky neexistují. Tento problém jsem vyřešil jednoduchým smazáním těchto importů a také smazáním všech metod, které se o tyto importy opíraly. Mohl jsem si to dovolit, protože samotná detekce žádnou z těchto metod pro svou funkčnost nepotřebuje.

Další změny se týkaly už pouze přizpůsobení faintu mým potřebám a odstranění nepotřebných balíků a tříd. Jako první jsem musel změnit soubor modules.conf v adresáři data. Z tohoto souboru (potažmo tím i z jeho kopie, kterou si faint udržuje v adresáři dokumentů aktuálně přihlášeného uživatele) si faint bere informace o tom, jakými pluginy a filtry se má snažit provádět detekci. Protože mým zájmem bylo, aby prováděl detekci pouze na základě pluginu OpenCV, smazal jsem z tohoto konfiguračního souboru všechny ostatní pluginy a filtry, kromě detekce pomocí OpenCV. Důvodem toho bylo hlavně redukovat čas výpočtu, který by faint strávil detekcí ostatními nepotřebnými metodami.

Aby tímto smazáním nedocházelo k chybám ve výpočtu, musel jsem následně ve třídě de.offis.faint.controller.MainController, která inicializuje mnoho objektů používaných pro detekci, smazat všechny inicializace objektů, které pracovaly s v předchozím kroku odstraněnými detekčními

metodami. Tam, kde nebylo možné pouze odstranit řádky kódu, protože bych tím zanesl chyby do dalších částí detekce, jsem jednoduše nahradil všechny reference na smazané pluginy a filtry hodnotou null. Konkrétním příkladem může být řádek 140 ve zmíněné třídě, který jsem změnil do podoby (místo hodnoty null se zde vyskytoval parametr availableDetectionFilters):

```
this.detectionHotSpot = new HotSpotController<IDetectionPlugin,
IDetectionFilter>(availableDetectionPlugins, null);
```

Díky těmto úpravám jsem následně mohl smazat všechny balíky (potažmo adresáře), které se netýkaly detekce pomocí OpenCV (tedy všechny ostatní detekční metody, třídy vytvářející grafické uživatelské rozhraní apod.). Díky tomu jsem také omezil velikost celého faintu z 318 MB na pouhých 21 MB.

Poslední změnou bylo přidání soukromého atributu type do třídy de.offis.faint.model.Region, abych mohl následně rozhodnout, zda detekovaná oblast vznikla pomocí detekce tváří, nebo pomocí jiných detekčních metod (lidských těl, automobilů nebo motocyklů). Protože je tento atribut podle dobrých zvyků soukromý, musel jsem také přidat metody pro jeho nastavení a získání jeho hodnoty.

5. Betaface

Jak jsem zmínil již v předchozích částech své práce, další možností, jak přistoupit k problému detekování lidských tváří, je využití služeb firmy Betaface. V této části nejprve stručně vysvětlím princip volně dostupné služby, kterou tato firma poskytuje, poté pomocí ní otestuji stejná data, jako v případě detekce pomocí frameworku faint a následně tato měření porovnám s testy z minulé kapitoly a zmíním výhody a nevýhody tohoto přístupu.

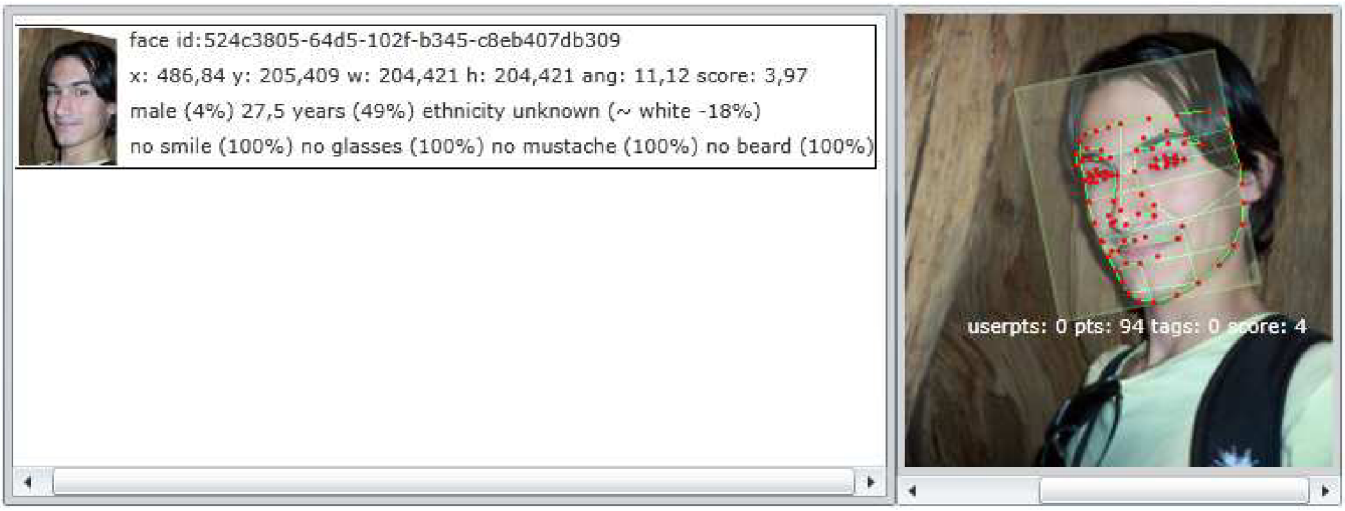
Na oficiálních stránkách Betaface.com je možné poskytovanou službu pro detekci (a také rozpoznávání) tváří využít dvojím způsobem. Zaprvé je možné si službu vyzkoušet online pomocí appletu, který načte soubor z disku a následně provede detekci a vrátí výsledky. Druhou možností (a řekl bych pro praktické využití vhodnější) je stáhnout si ukázkového klienta využívajícího poskytovanou službu. Jak online detekce, tak tento klient, je dostupný na stánkách v literatuře označených jako [23].

Tento klient pracuje v podstatě shodně s tím, jak pracuje jeho online obdoba, s tím rozdílem, že ho lze daleko jednodušeji začlenit do případných vyvíjených aplikací. Adresář s klientem obsahuje několik dll knihoven, soubory se zdrojovými kódy, soubory s nastavením a také jeden exe soubor, který tohoto klienta spouští a dva bat soubory, z nichž jeden spouští zmíněný exe soubor s parametry pro detekci tváří a jeden s parametry pro rozpoznávání určité tváře na jiných fotografiích. Pro mou diplomovou práci je samozřejmě důležitý pouze soubor řešící problém detekce. V podstatě tedy pro detekci tváří lze spustit exe soubor skrze bat soubor nebo skrze příkazový řádek zadáním následujícího příkazu:

BetafaceWebServiceTest.exe "http://208.109.122.177/BetafaceWebService.svc/" "d45fd466-51e2-4701-8da8-04351c872236" "171e8465-f548-401d-b63b-caf0dc28df5f" upload image.jpg

Prvním parametrem je adresa, na které je služba Betaface dostupná, druhým a třetím parametrem jsou licenční klíče, které firma Betaface publikovala pro nekomerční účely zdarma na svých stránkách, následuje příkaz „upload“, čímž říkáme, že chceme provádět detekci tváří (v případě rozpoznávání by následovalo klíčové slovo „compare“) a jako poslední parametr následuje výčet cest k obrázkům, na kterých chceme detekci provádět.

Po potvrzení tohoto příkazu jsou všechny obrázky specifikované posledním parametrem postupně odeslány na server Betaface, kde jsou zpracovány a následně jsou zpět odesílány soubory obsahující detekované tváře a pro každý obrázek také jeden xml soubor popisující každou oblast, ve které byla detekována tvář. Tento soubor obsahuje kromě rozměrů a pozice každé takové oblasti ještě dodatečné informace o detekované tváři – viz Obrázek 6. Tento obrázek zachycuje příklad detekce pomocí online verze klienta, stejné hodnoty však obsahuje zmíněný xml soubor, jen jsem pro přehlednost raději zvolil toto řešení.



Obrázek 6

Jen pro doplnění, kromě všech hodnot v rámečku vlevo obsahuje xml soubor také souřadnice všech bodů na obrázku vyznačených červeně.

6. Testy

V následující části představím nejprve část zdrojového kódu třídy sloužící k vytvoření testovacích dat, díky kterým následně vyberu nastavení, které vykazuje v detekci lidských tváří nejlepší výsledky. Toto nastavení je důležité vybrat proto, že ve výsledné aplikaci bude možnost zvolit si mezi několika různými profily, které budou k rozpoznávání každý přistupovat jiným způsobem (jeden bude rozpoznávat více tváří s rizikem vzniku více chybně rozpoznaných oblastí, jiný naopak apod.). Toto výše popsané nastavení přitom bude označeno jako ideální a použito jako základní způsob detekce.

Poté v krátkosti vysvětlím funkcionalitu testovací třídy, respektive její main metody. Pro sběr testovacích fotografií jsem využil stránek sociální sítě Facebook, přičemž jako hlavní testovací soubory jsem zvolil mé osobní fotky a pro další dodatečné testy fotografie mých přátel.

Kód testovací třídy:

```
public static void main(String[] args){
    String folderPath = "d:/Diplomka/TestPhotos/RadekStasta/Faces/";
    File folder = new File(folderPath);
    File[] files = folder.listFiles();

    int face = 1;

    OpenCVDetection detection = new OpenCVDetection();
    detection.setCascade("haarcascade_frontalface_default.xml");
    detection.setScale(1.1f);

        for(int i=0;i<files.length;i++){
```

```
                Region[] regions = detection.detectFaces(files[i].toString(),1);
                try {
                  for (Region region : regions){
                        System.out.println(region);
                  BufferedImage img = region.toThumbnail(region.getWidth(), region.getHeight());
                  File output = new File(folderPath + "ResultsDefaultF11S1/" + face + ".jpg");
                  ImageIO.write(img, "jpg", output);
                  face++;
              }
            } catch (NullPointerException ex) {
                System.out.println("No faces detected in current photo");
            } catch (IOException ex) {
                Logger.getLogger(OpenCVfaceTests.class.getName()).log(Level.SEVERE, null, ex);
            }
          }
        }
}
```

V úvodu této metody je nejprve nastavena cesta do adresáře, ve kterém se nacházejí soubory s testovacími fotografiemi. Tato cesta byla měněna pokaždé, když jsem dokončil testování fotografií konkrétního člověka. Do seznamu files typu File jsou následně načteny jména všech souborů, nacházejících se v adresáři zadaném v předchozím kroku. Dále je nastaven číselný atribut face, který bude následně použit pro očíslování jednotlivých nalezených tváří (a samozřejmě i částí fotografií, které byly chybně označeny jako tváře, i když žádné tváře neobsahují) ve zkoumaných fotkách. Toto číslo bylo při testování také použito jako jméno souboru pro uložení jednotlivých výřezů fotografií na disk.

Následuje inicializace detekční třídy s využitím třídy OpenCVDetection frameworku faint a nastavení použitého klasifikátoru a faktoru zvětšování okna na požadované hodnoty. Tyto hodnoty byly také při testech často měněny, abych mohl prozkoumat efektivitu jednotlivých klasifikátorů při různých faktorech zvětšování okna.

Poslední částí algoritmu je cyklus, který pro každou fotografii ve zvoleném adresáři vykoná následující akce. Vytvoří seznam oblastí, ve kterých byly pomocí metody faintu detectFaces detekovány lidské tváře (pro atribut minimální velikosti okna byly použity hodnoty 1, 10, 20, 30, 40 a 50 – vysvětlení níže). Pro každou takto nalezenou oblast byly na standardní výstup vypsány její informace a následně byla uložena na disk pod číslovkou právě uloženou v atributu face. Nakonec následuje ošetření některých výjimek, které mohly v průběhu výpočtu nastat.

Následující část diplomové práce se zabývá výsledky testů, které byly pořízeny na základě právě probraného algoritmu. Na konci následující části pak vyberu jedno nastavení detekčního algoritmu, které testy prošlo s největším úspěchem a uvedu také důvody, proč bylo vybráno právě toto nastavení.

6.1. Testování detekce tváří a jeho výsledky

Předtím, než uvedu konkrétní výsledky jednotlivých testů, bych rád popsal proces, který před testováním předcházel, i proces samotného testování a sběru jeho výsledků.

Jak jsem již zmínil, všechny fotografie, které jsem pro svou diplomovou práci získal, byly staženy z profilů sociální sítě Facebook. Pro stahování těchto fotografií jsem použil dva nástroje. První z nich byl program fbdownloader dostupný ze stránek uvedených v literatuře [21]. Jedná se o desktopovou aplikaci, která je po přihlášení stejnými přihlašovacími údaji jako pro Facebook schopna stáhnout všechny fotografie nacházející se na profilu přihlášeného a všech jeho přátel. Tento program bohužel nebyl ve všech případech stoprocentně spolehlivý, a proto jsem pro všechny ostatní fotografie, které nebylo možné stáhnout tímto programem, využil online aplikaci facebook2zip dostupnou na stánkách v literatuře uvedených pod číslem [22], který pracuje na velice podobném principu.

Ze všech stažených fotografií jsem v druhém kroku testování vybral takové, které obsahovaly tváře v různých měřítcích, úhlech a vyfocených při různých podmínkách (v přímém světle, ve tmě, v mlze apod.). Protože i na těchto vybraných fotografiích vzniklo v některých případech velké množství chybně detekovaných oblastí, nepovažuji za nutné provádět testy i na sadě obrázků, které tváře vůbec neobsahují, protože předpokládám, že procento chyb, které by vznikly na tomto typu obrázků, by bylo velmi podobné tomu, které vzniklo při testech na obrázcích s tvářemi.

Pro každého zvoleného uživatele sociální sítě tak vznikl soubor o velikosti od 100 do 150 fotografií. Na všech těchto fotografiích jsem následně manuálně spočítal všechny tváře a poté jsem tyto fotografie nechal zpracovávat výše popsanou testovací třídou. Výsledky (myšleno soubory vzniklé jako výstup testovací třídy) jsem opět ručně prošel a spočítal, kolik z nich obsahuje tváře a

kolik z nich bylo chybně jako tvář metodami faintu označeno (při použití různých parametrů minimální velikosti okna, faktoru zvětšování okna a použitého klasifikátoru).

Protože jsem při ručním počítání měl s některými fotografiemi poněkud problém v tom smyslu, že obsahovaly například tváře příliš vzdálené, takže i lidským okem byly téměř nerozeznatelné, musel jsem zavést některá omezení na to, co za rozeznatelnou tvář považuji, a podle kterých jsem při počítání postupoval a díky kterým jsem, jak doufám, omezil možné nejednoznačnosti. Tato omezení byla následující:

- tvář nesmí být příliš malá (musí mít jasně rozpoznatelné rysy)
- převážná většina tváře musí být zachycena v obrázku (nepočítal jsem tváře, které jsou oříznuty po okrajích fotografie)
- tvář musí být vidět alespoň z profilu (nepočítal jsem tváře, které jsou otočeny o více jak 90° směrem od fotoaparátu)
- pokud se tvář vyskytla v davu, počítal jsem jen ty, které byly z převážné většiny viditelné, nezakryté ostatními tvářemi

Jako první jsem pro testování zvolil fotografie vlastní. Celkem jsem použil 124 souborů obsahujících v součtu 331 tváří. V průběhu testů jsem použil všechny možné kombinace klasifikátorů (jmenovitě klasifikátory default, alt, alt2 a altTree), dvou hodnot faktoru zvětšování okna (faktor 1,1 a 1,05) a šesti hodnot minimální velikost okna (velikost 1, 10, 20, 30, 40 a 50). V průběhu několika prvních testů vycházelo, že na výsledek nemá vliv, zda byla použita velikost 1, 10 nebo 20, a proto jsem v následujících testech používal již pouze hodnoty 1, 30, 40 a 50. Takto vzniklo celkem 32 unikátních měření, přičemž u každého měření mě zajímaly tři věci. Jednak samozřejmě to, kolik procent tváří bylo zvoleným nastavením parametrů detekováno. Dále to, kolik procent z celkově vzniklých souborů tváře neobsahuje a bylo tudíž chybně jako tvář detekováno a v neposlední řadě

pak také to, jak dlouho trvala detekce v jednom testovacím obrázku. Z takto získaných dat jsem ještě vždy vypočítal čtvrtou hodnotu $Diff$, který dává do jistého poměru míru správně a chybně detekovaných tváří (o kolik procent se konkrétní nastavení liší od ideálního stavu 100% správně a 0% chybě detekovaných tváří). Tuto hodnotu jsem počítal pomocí následujícího jednoduchého vzorce:

$$Diff = (1 - CA) + FA,$$

$$kde\ CA\ je\ procento\ správně\ detekových\ tváří\ a\ FA\ procento\ chybně\ detekovaných\ tváří$$

Je jasné, že mou snahou bylo nalézt takové nastavení, které by mělo hodnotu $Diff$ co nejmenší (ideálně 0, protože právě to vystihuje situaci, kdy máme 100% správně a 0% chybně detekovaných tváří). Následující tabulka shrnuje všech 32 měření:

Klasifikátor	Zvětšování	Okno	CA	FA	Čas v s	Diff
Default	1,1	1	75,83%	42,30%	1,06	66,47%
Default	1,1	30	68,89%	43,00%	0,87	74,11%
Default	1,1	40	54,38%	42,49%	0,70	88,11%
Default	1,1	50	43,81%	42,00%	0,51	98,19%
Default	1,05	1	84,89%	60,97%	1,57	76,08%
Default	1,05	30	80,97%	59,08%	1,26	78,11%
Default	1,05	40	66,16%	57,22%	0,97	91,06%
Default	1,05	50	51,96%	54,86%	0,73	102,90%
Alt	1,1	1	72,21%	15,25%	0,77	43,04%
Alt	1,1	30	64,35%	14,11%	0,56	49,76%
Alt	1,1	40	51,66%	14,50%	0,41	62,84%
Alt	1,1	50	37,46%	13,89%	0,31	76,43%
Alt	1,05	1	77,34%	33,68%	1,31	56,34%
Alt	1,05	30	74,32%	33,15%	0,96	58,83%
Alt	1,05	40	61,33%	30,24%	0,69	68,91%
Alt	1,05	50	44,41%	30,33%	0,48	85,92%
Alt2	1,1	1	74,62%	19,28%	0,70	44,66%
Alt2	1,1	30	68,88%	18,57%	0,53	49,69%
Alt2	1,1	40	52,87%	16,27%	0,38	63,40%
Alt2	1,1	50	38,67%	16,33%	0,29	77,66%
Alt2	1,05	1	78,25%	43,82%	1,22	65,57%
Alt2	1,05	30	74,32%	42,66%	0,93	68,34%
Alt2	1,05	40	62,54%	40,35%	0,69	77,81%
Alt2	1,05	50	46,22%	37,04%	0,47	90,82%
AltTree	1,1	1	65,25%	11,46%	0,61	46,21%
AltTree	1,1	30	57,10%	12,50%	0,45	55,40%

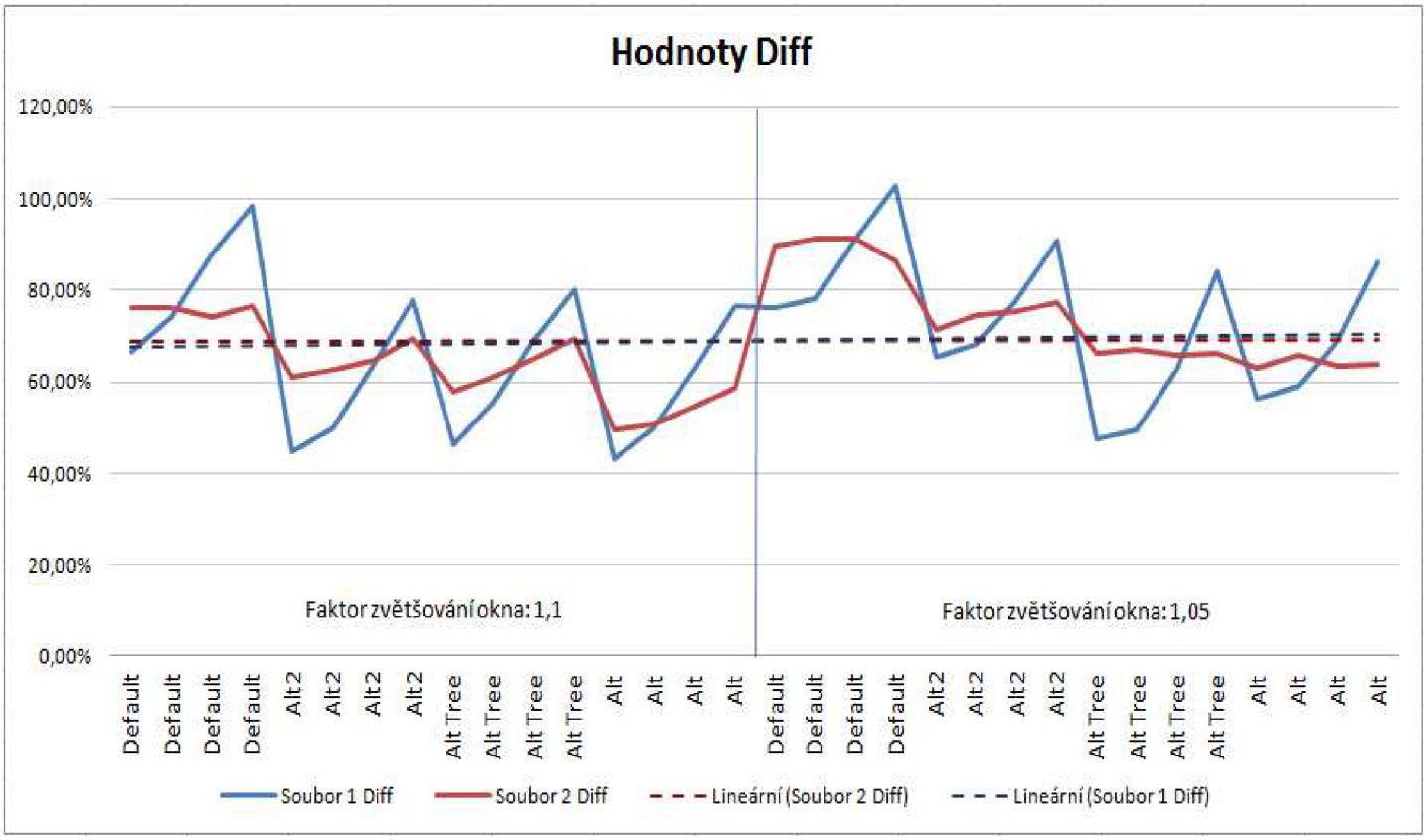
AltTree	1,1	40	42,90%	11,80%	0,34	68,90%
AltTree	1,1	50	31,12%	11,20%	0,27	80,08%
AltTree	1,05	1	72,81%	20,20%	0,98	47,39%
AltTree	1,05	30	69,49%	19,01%	0,75	49,52%
AltTree	1,05	40	55,29%	18,30%	0,56	63,01%
AltTree	1,05	50	36,86%	20,78%	0,38	83,92%

Z tabulky je vidět, že několik ze zvolených nastavení vykazuje vcelku dobré výsledky – sedm z nich dosahuje úrovně hodnoty $Diff$ menší, než 50%. Nejlepší efektivity dosahuje použití klasifikátoru Alt při faktoru zvětšování okna 1,1 a velikosti tohoto okna 1. Toto nastavení bylo schopno detekovat takřka tři čtvrtiny tváří a přitom označilo pouze 15,25% dalších míst na testovacích fotografiích, které tváře neobsahovaly. Další dvě nastavení nedosáhly již tak dobrého celkového výsledku, ale na druhou stranu dosáhly vždy lepšího výsledku v jedné ze dvou proměnných, které přispívají do hodnoty $Diff$. Klasifikátor Alt2 se stejným nastavením dokázal rozpoznat ještě o dvě procenta více tváří, ale bohužel vytvořil o více jak čtyři procenta více chybných souborů (s oblastmi, které žádný obličej neobsahují). Klasifikátor AltTree opět se stejným nastavením vytvořil těchto chybných souborů podstatně méně než klasifikátor Alt (pouze 11,46%, což je o čtyři procenta méně, než klasifikátor Alt), ale zase pokulhává, co se týče detekce tváří, kde dosáhl pouze 65-ti procentní úspěšnosti. Pro přesnější výsledky testů jsem testoval ještě jiný soubor fotografií, abych věděl, zda budou výsledky testů podobné. Tento soubor obsahoval celkem 150 fotografií, na kterých jsem ručně identifikoval 341 tváří. Následující tabulka shrnuje výsledky těchto testů. Vzhledem k tomu, že časy výpočtů nad tímto souborem byly více méně shodné, rozhodl jsem se v tabulce uvést místo těchto časů hodnoty pojmenované jako Rozdíl, které jsou v této chvíli podstatnější. Rozdíl zachycuje, jak se v tomto souboru liší hodnoty $Diff$ od korespondujících hodnot v předchozím souboru.

Klasifikátor	Zvětšování	Okno	CA	FA	Diff	Rozdíl
Default	1,1	1	70,97%	46,92%	75,95%	9,48%
Default	1,1	30	68,91%	45,22%	76,31%	2,20%
Default	1,1	40	64,52%	38,72%	74,20%	-13,91%
Default	1,1	50	58,65%	35,28%	76,63%	-21,56%
Default	1,05	1	76,25%	66,06%	89,81%	13,73%
Default	1,05	30	73,71%	64,74%	91,03%	12,92%
Default	1,05	40	69,21%	60,27%	91,06%	0,00%
Default	1,05	50	64,81%	51,11%	86,30%	-16,60%
Alt	1,1	1	66,86%	16,48%	49,62%	6,58%
Alt	1,1	30	64,52%	15,06%	50,54%	0,78%
Alt	1,1	40	59,53%	13,98%	54,45%	-8,39%
Alt	1,1	50	53,37%	12,08%	58,71%	-17,72%
Alt	1,05	1	70,97%	33,88%	62,91%	6,57%
Alt	1,05	30	68,04%	33,71%	65,67%	6,84%
Alt	1,05	40	65,10%	28,39%	63,29%	-5,62%
Alt	1,05	50	58,36%	22,27%	63,91%	-22,01%
Alt2	1,1	1	64,52%	25,42%	60,90%	16,24%
Alt2	1,1	30	61,88%	24,37%	62,49%	12,80%
Alt2	1,1	40	57,48%	21,91%	64,43%	1,03%
Alt2	1,1	50	51,03%	20,55%	69,52%	-8,14%
Alt2	1,05	1	72,14%	43,45%	71,31%	5,74%
Alt2	1,05	30	68,33%	42,75%	74,42%	6,08%
Alt2	1,05	40	64,81%	40,11%	75,30%	-2,51%
Alt2	1,05	50	57,18%	34,34%	77,16%	-13,66%
AltTree	1,1	1	52,20%	10,10%	57,90%	11,69%
AltTree	1,1	30	49,56%	10,58%	61,02%	5,62%
AltTree	1,1	40	44,87%	10,00%	65,13%	-3,77%
AltTree	1,1	50	39,00%	8,28%	69,28%	-10,80%
AltTree	1,05	1	58,36%	24,62%	66,26%	18,87%
AltTree	1,05	30	56,60%	23,41%	66,81%	17,29%
AltTree	1,05	40	53,08%	18,83%	65,75%	2,74%
AltTree	1,05	50	46,92%	13,04%	66,12%	-17,80%

Pokud se vyskytuje v této tabulce ve sloupci Rozdíl kladná hodnota, znamená to, že je hodnota $Diff$ v tomto souboru (označme ho Soubor 2) vyšší než v souboru předchozím (označme ho Soubor 1). Pokud se vyskytuje hodnota záporná, je tomu právě naopak. Při prvním zkoumání by se mohlo zdát, že si hodnoty pro jednotlivé soubory příliš neodpovídají a že z těchto testů nelze tedy vyvozovat závěry o tom, jak efektivní jsou jednotlivá nastavení. Pokud však výsledky nebudeme brát zvlášť pro jednotlivé řádky a podíváme se na ně z celkového statistického hlediska, zjistíme, že

výsledky nejsou zase tak odlišné, jak by se mohlo na první pohled zdát. Tento fakt nejlépe dokládá graf na Obrázku 7.



Obrázek 7

Hodnoty $Diff$ jsou v grafu seřazeny podle faktoru zvětšování okna a následně podle zvoleného klasifikátoru. Jak je vidět, jediný znatelný rozdíl v obou souborech je ten, že v Souboru 1 jsou hodnoty extrémnější než v Souboru 2. To však nemění nic na tom, že jednotlivá nastavení vykazují pro oba dva soubory stejné výsledky, liší se pouze proporcionálně v závislosti na vhodnosti jednotlivých souborů (myšleno tak, zda se v souboru vyskytují tváře, které jsou snadněji detekovatelné všemi nastaveními a naopak). Pro ještě bližší vysvětlení se například zdá, že klasifikátor Default s faktorem zvětšování okna 1,05 je ze všech nastavení nejhorší pro oba soubory a jediný rozdíl je v tom, že v Souboru 1 vykazuje daleko horších výsledků než v Souboru 2. Grafy pro oba soubory jsem navíc proložil regresní přímkou. Jak je vidět, tyto regresní přímky jsou takřka

totožné, což jen dokládá tvrzení o tom, že jednotlivá nastavení dosahují stejných výsledků, liší se pouze ve svých extrémech.

V této kapitole jsem tedy dokázal, že lze brát výše popsané testy za dostatečně směrodatné. Díky těmto poznatkům v následující kapitole vyberu jedno konkrétní nastavení, které vykazuje nejlepší výsledky, a uvedu také argumenty, které vedly k tomuto výběru.

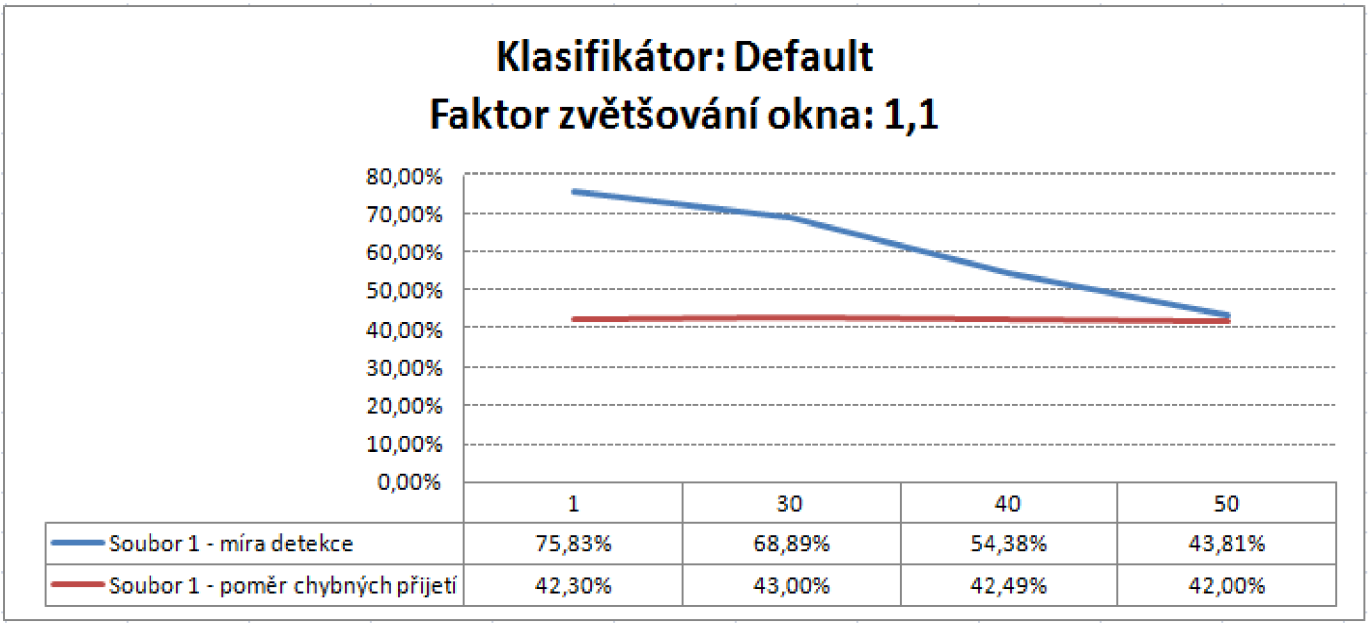
6.1.1. Výběr vhodného nastavení

Jak jsem zmínil v předchozí kapitole, nyní se zaměřím na to, vybrat co nejvhodnější nastavení jako kandidáta na základní ideální nastavení v implementaci pro systém G.A.T.E. Tento výběr provedu tak, že se pokusím uvést dostatečné množství argumentů pro vyloučení určité množiny nastavení, dokud nebude tato množina natolik malá, že bude možné vybrat jediné nastavení na základě jeho porovnání s ostatními nastaveními z takto redukované množiny.

Jak jsem také v předchozí kapitole dokázal, je v podstatě jedno, jaký testovací soubor k tomuto účelu použiji. Pro důkazy jsem zvolil Soubor 1, a to proto, že výsledky mají extrémnější charakter, a proto na něm budou důkazy zřetelnější.

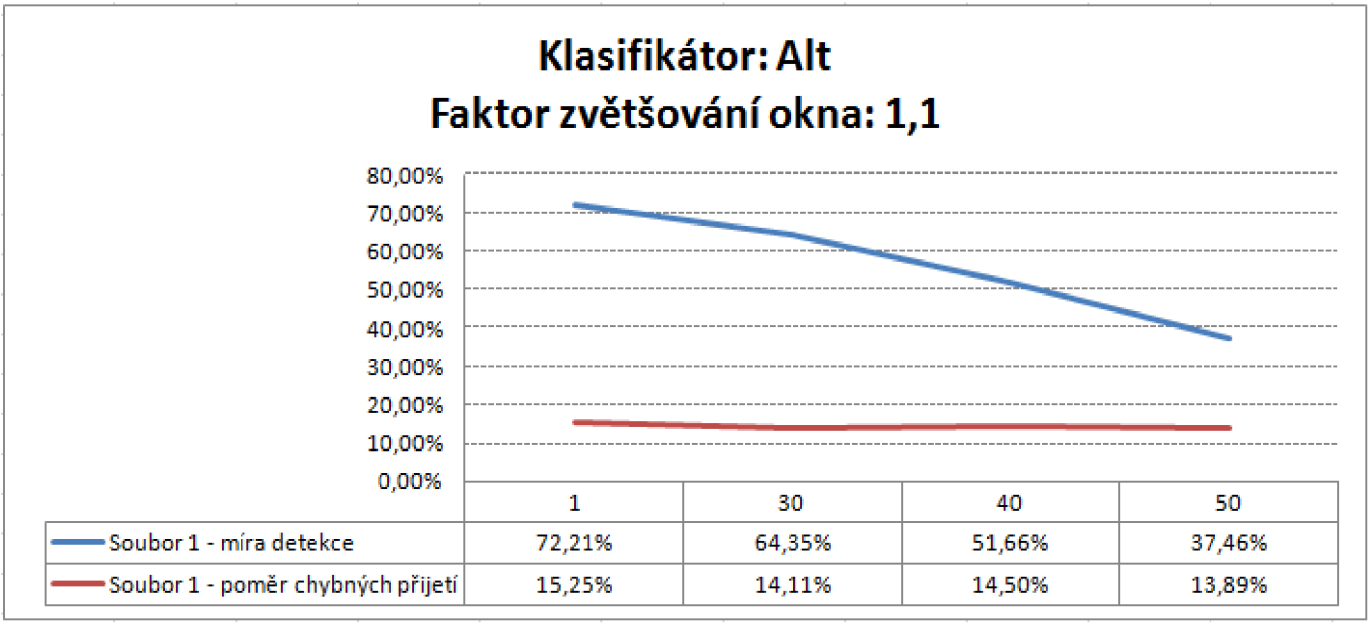
Jako první bych rád dokázal fakt, že není třeba hledat kandidáta v nastaveních, která mají hodnotu minimální velikosti okna větší než 1. Minimální velikost okna nám v podstatě říká, od jak velké čtvercové oblasti začneme provádět detekci. Mějme tedy tvrzení, že pokud převážná většina chybně detekovaných oblastí spadá do kategorie velice malých oblastí v obrázku, pak zvolením dostatečně velkého počátečního okna jsme schopni zredukovat počet těchto chybných detekcí se zachováním přibližně stejného procenta správných detekcí. K vyvrácení tohoto tvrzení využiji grafy na Obrázcích 8a – d.

Jak je na všech grafech jasně patrné, výše zmíněné tvrzení rozhodně neplatí. Se zvětšující se minimální velikostí okna se velice drasticky snižuje procento správně rozpoznaných tváří v obrázku, ale poměr chybných přijetí oblastí, na kterých se obličej nevyskytuje, zůstává veskrze konstantní. To samé platí i pro nastavení s faktorem zvětšování okna 1,05. Díky tomuto faktu můžeme tedy omezit množinu všech nastavení na ty, které mají minimální velikost okna 1 a vyřadit všechny ostatní, protože v nich zajisté nenajdeme takové nastavení, které má lepší vlastnosti.



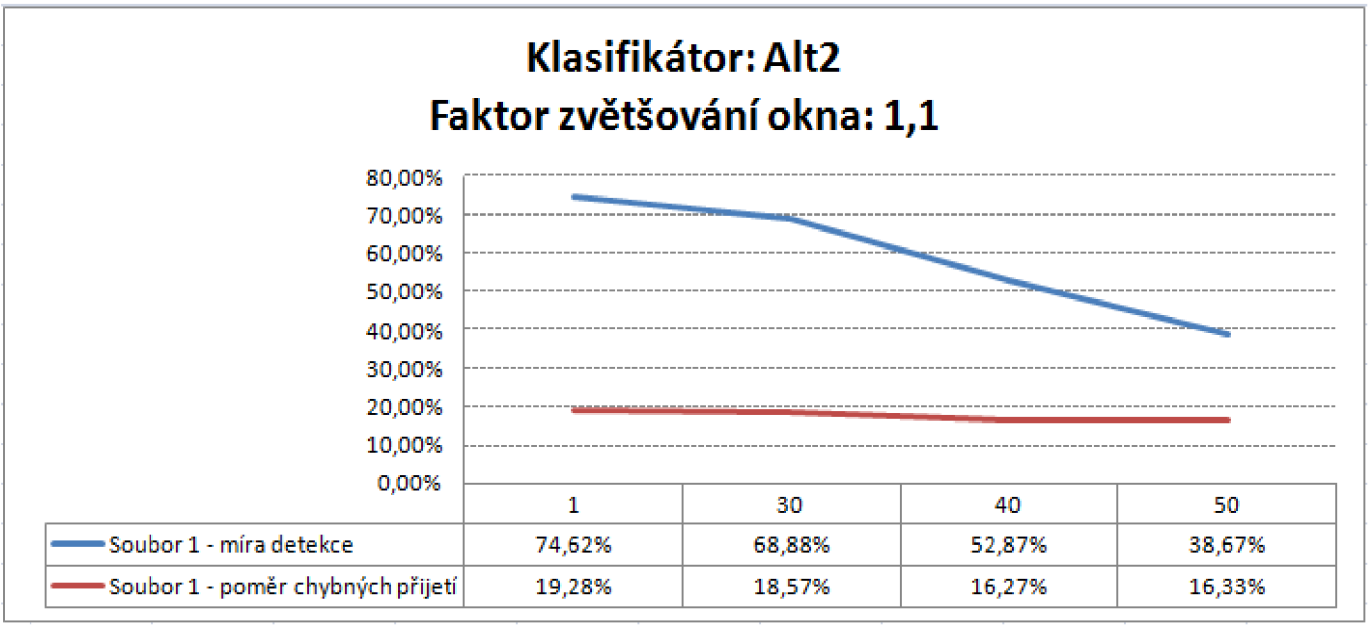
	1	30	40	50
Soubor 1 - míra detekce	75,83%	68,89%	54,38%	43,81%
Soubor 1 - poměr chybných přijetí	42,30%	43,00%	42,49%	42,00%

Obrázek 8a

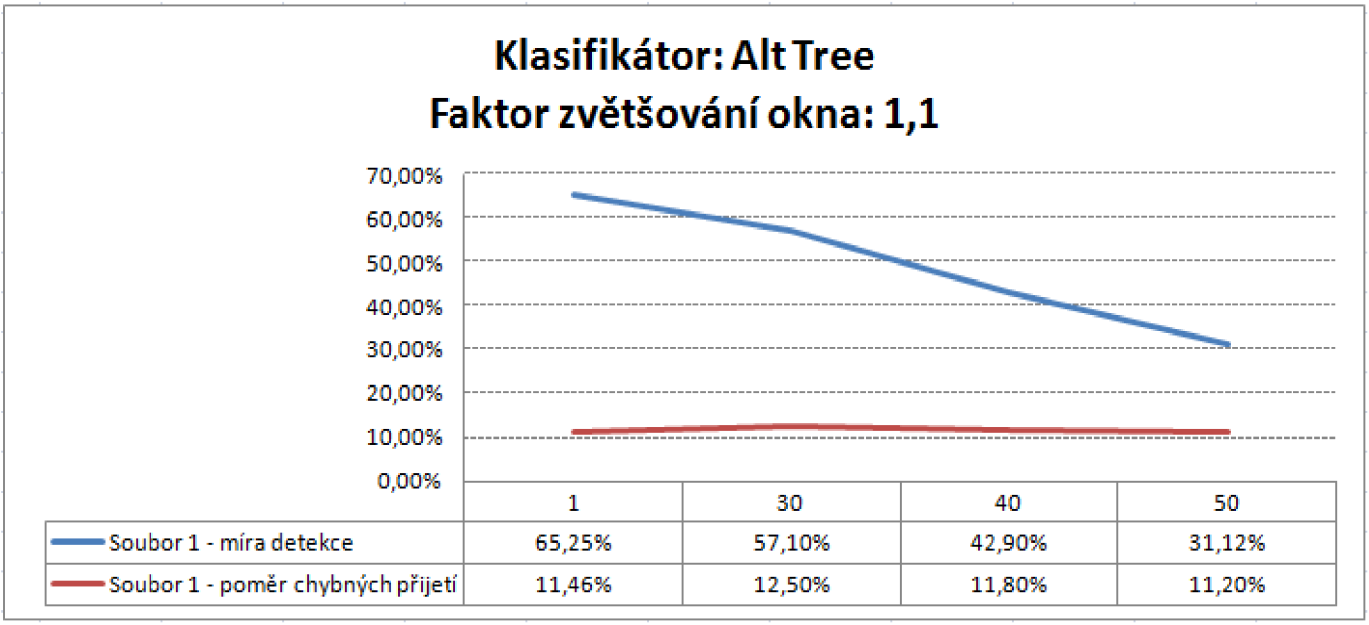


	1	30	40	50
Soubor 1 - míra detekce	72,21%	64,35%	51,66%	37,46%
Soubor 1 - poměr chybných přijetí	15,25%	14,11%	14,50%	13,89%

Obrázek 8b



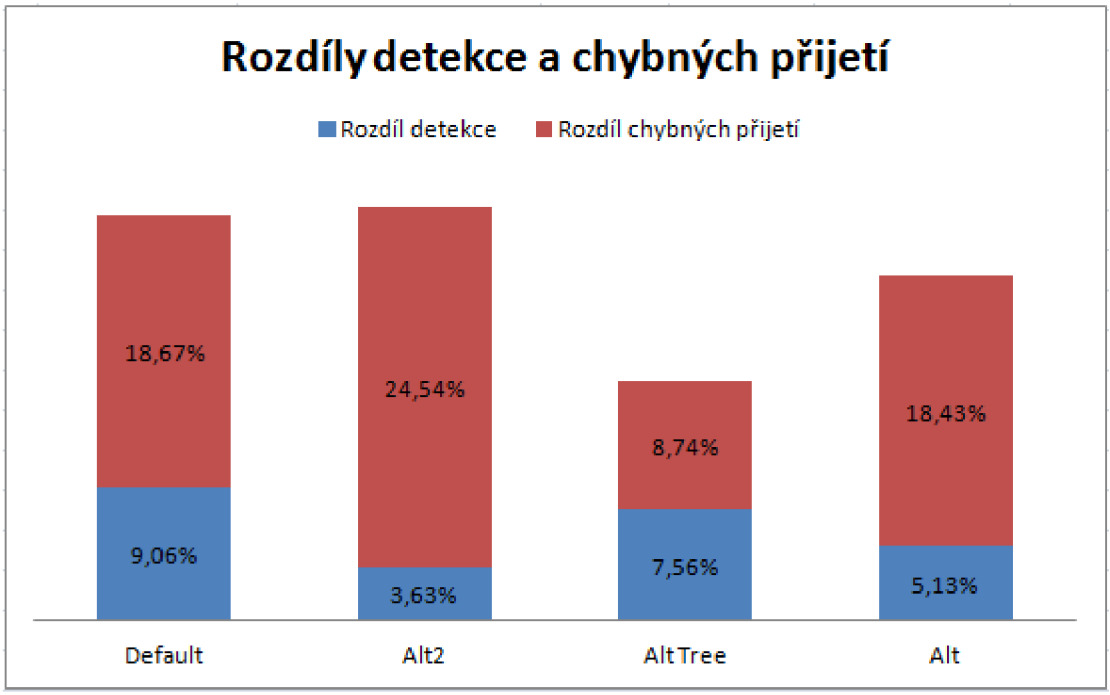
Obrázek 8c



Obrázek 8d

Tím jsme radikálně omezili množinu nastavení z třiceti dvou na osm. Dále se podívejme na to, zda je výhodnější použít faktor zvětšování okna 1,1 nebo 1,05. Tento faktor udává, o kolik procent zvětšíme velikost okna před opětovným prohledáváním obrázku. V našem případě zvětšujeme velikost okna o deset nebo pět procent. Pokud zvětšujeme velikost o pět procent, dosáhneme zajisté

lepší přesnosti a měli bychom tedy být schopni detekovat větší procento tváří. Nyní záleží na tom, zda tímto zjemněním nedosáhneme bohužel také toho, že zvýšíme poměr chybných detekcí. Pokud vezmeme všechny klasifikátory a vypočítáme rozdíly detekce a chybných přijetí mezi jednotlivými faktory (hodnoty faktoru 1,1 odečteme od hodnot faktoru 1,05), vznikne nám následující graf (obrázek 9):



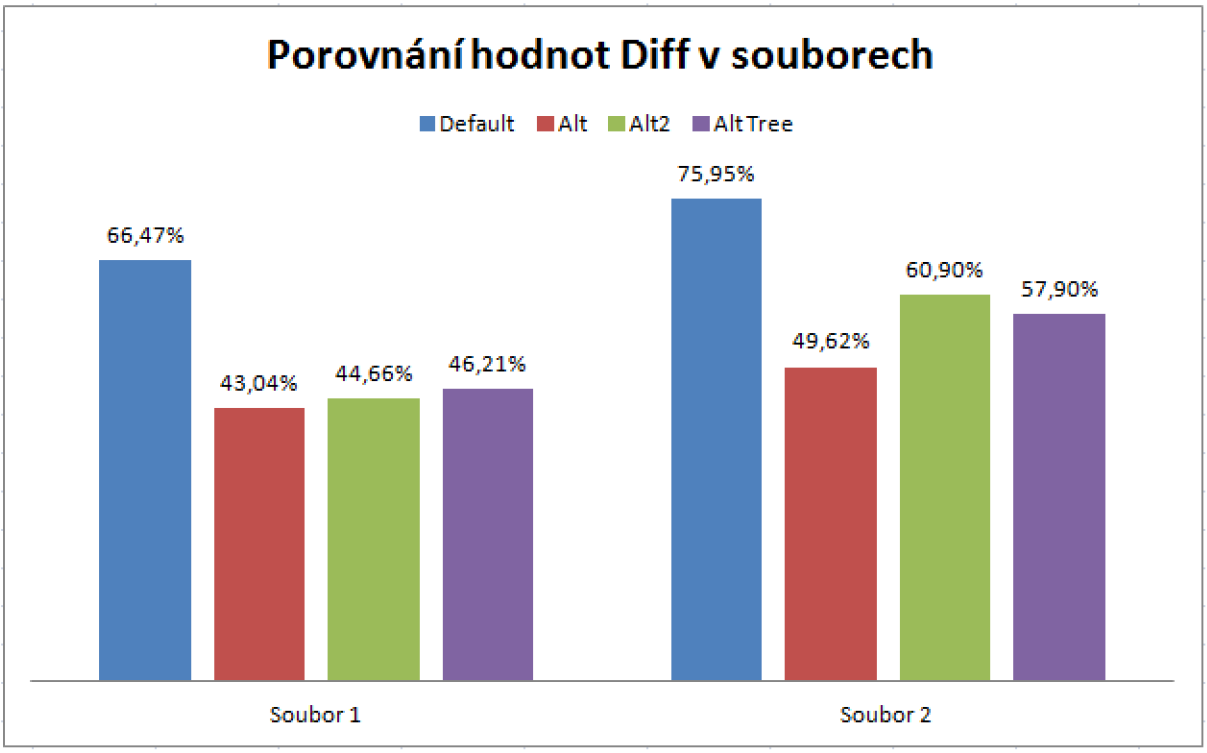
Obrázek 9

Jak je možné z grafů vidět, u všech klasifikátoru dojde k tomu, že se sice zvýší procento správně detekovaných tváří, ale také se zvýší procento chybných přijetí. Přitom platí následující vztah (názvy částí vzorce odpovídají názvům z předchozí kapitoly):

$$FA = k \cdot CA, kde\ k > 1$$

To znamená, že použitím faktoru zvětšování okna 1,05 vždy dosáhneme více chybných detekcí, než těch správných (v případě klasifikátoru Alt2 dokonce skoro sedmkrát více).

Díky tomuto důkazu můžeme množinu kandidátů omezit na čtyři, přičemž nás zajímají pouze ta nastavení, která mají faktor zvětšování okna 1,1 a hodnotu minimální velikosti okna 1. Nezbývá než tyto čtyři porovnat a na základě tohoto porovnání vybrat jedno finální nastavení. Pro tento výběr opět využijeme oba soubory a porovnáme hodnoty $Diff$ ve čtyřech zbývajících nastaveních – viz graf na Obrázku 10.



Obrázek 10

Jak je vidět z posledního grafu, v prvním souboru dosahují tři ze čtyř nastavení takřka shodných výsledků. V Souboru 2 je však již situace odlišná, a proto tím nastavením, které jsem vybral jako kandidáta pro ideální a základní nastavení aplikace pro systém G.A.T.E., je použití klasifikátoru Alt s faktorem zvětšování okna 1,1 a minimální velikostí okna 1. V obou souborech toto nastavení dosahuje 69,54% úspěšnosti detekce tváří při 15,87% chybných detekcí.

6.1.2. Dodatečné testy

Po provedení těchto výše popsaných testů, které vedly k výběru jednoho konkrétního nastavení a také ukázaly efektivitu všech ostatních nastavení, jsem rozšířil množinu testovacích souborů, abych mohl všechna předchozí tvrzení podložit testy na velkém množství dat. Pro tyto účely jsem použil celkem 1018 souborů obsahující v součtu 1710 tváří. Na těchto datech jsem samozřejmě neprováděl již tak důkladné testy. Jako první mne zajímaly výsledky nastavení zvoleného v minulé kapitole. Při testování na takto rozšířené množině dat detekovalo toto nastavení celkem 72,22% všech tváří při 17,86% chybných detekcí. Jak je vidět, i tato procenta vesměs odpovídají výsledkům z předchozí kapitoly.

Pro aplikaci, která je výsledkem mé bakalářské práce, však bylo potřeba zvolit ještě dvě další nastavení pro možnost profilování této aplikace konkrétním potřebám uživatelů. Jako nastavení, které je schopno detekovat co nejvíce tváří za rizika vyššího výskytu chybných detekcí, jsem zvolil klasifikátor Alt2 s faktorem zvětšování okna 1,05 a velikostí okna 1. Toto nastavení sice v předchozích testech nedosáhlo úplně nejvyššího procenta detekcí, ale z těch nejvýše hodnocených dosáhlo nejnižší hodnoty $Diff$ (v průměru z obou testovacích souborů z minulé kapitoly 75,20% detekovaných tváří při 43,64% chybných detekcí). Úplně nejlépe hodnocené nastavení v míře detekování tváří jsem nepoužil proto, že v průměru dosáhlo sice 80,57% detekovaných tváří, ale bohužel při 63,52% chybných detekcí. To v praxi znamená, že každá detekovaná oblast může být s podobnou pravděpodobností jak tvář, tak chyba, což není příliš příznivý výsledek a o zkoumaném obrázku bychom neřekli nic nového. Při testování na rozšířené množině dat detekoval klasifikátor Alt2 78,34% tváří při 46,24% chybných detekcí.

Jako poslední nastavení pro profilování aplikace jsem zvolil klasifikátor AltTree s faktorem zvětšování okna 1,1 a velikostí okna 1. Toto nastavení je přesným protikladem nastavení minulého.

To znamená, že minimalizuje počet chybných detekcí s nižším procentem správně detekovaných tváří. Toto nastavení opět nevykazovalo v testech v předchozí kapitole nejmenší hodnoty poměru chybných detekcí, ale kvůli podobným důvodům, jaké jsem uvedl v předchozím případě, jsem zvolil toto nastavení. Při testech na rozšířené množině testovacích dat bylo schopno detekovat 58,13% tváří při 9,76% chybných detekcí.

Tím nám vznikly tři možnosti, jak provádět detekci nad konkrétním obrázkem. Zaprvé se můžeme snažit detekovat co nejvíce tváří, ale musíme se smířit s velkým procentem chyb v případě klasifikátoru Alt2. Zadruhé si můžeme chtít být s velkou pravděpodobností jisti, že detekovaná oblast je skutečně tváří s tou nevýhodou, že velkou část tváří nebudeme schopni detekovat v případě klasifikátoru AltTree. A zatřetí můžeme chtít jistý kompromis mezi těmito nastaveními a poté použijeme nastavení popsané v předchozí kapitole. Všechny tyto tři možnosti budou v části implementace začleněny do výsledné aplikace.

6.1.3. Testování detekce Betaface a jeho výsledky

Testování služby firmy Betaface jsem pochopitelně prováděl na dvou souborech testovacích dat popsaných v kapitole 6.1. Následující tabulka shrnuje výsledky tohoto testování a zároveň je porovnává s výsledky detekce pomocí frameworku faint, konkrétně ideálně zvoleného nastavení:

Soubor	% detekovaných tváří	% chybných detekcí
	Betaface	
Soubor 1	80,97%	8,84%
Soubor 2	72,67%	11,62%
	Faint	
Soubor 1	72,21%	15,25%
Soubor 2	66,86%	16,48%

Jak je z tabulky vidět, detekce tváří pomocí služby firmy Betaface vyšlo pro oba soubory daleko lépe, jak co se týče procenta správně detekovaných tváří, tak procenta chybných detekcí (myšleno opět kdy byla nějaká oblast na obrázku označena jako tvář, i když se ve skutečnosti o tvář vůbec nejednalo).

Z výsledků těchto testů a také z toho, že pomocí Betaface detekce jsme s určitou pravděpodobností schopni pro jednotlivé tváře dodat další informace o tom, jaké je člověk na fotografii etnické příslušnosti, zda se usmívá, zda má či nemá brýle apod., by se mohlo zdát, že použití Betaface pro aplikaci pro systém G.A.T.E. je daleko vhodnější, než použití detekci založenou na frameworku faint. Pokud bychom se rozhodovali pouze podle těchto vlastností, pak by to samozřejmě byla pravda.

Detekce pomocí služby Betaface má bohužel však také některé negativní vlastnosti. Jednou z nich je nepochybně čas výpočtu. Pokud by se jednalo pouze o zpomalení z důvodu přenosu jednotlivých souborů na vzdálený server a následný přenos výsledků zpět, nejednalo by se zase o

takový problém. Nejzákladnější verze klienta však bohužel obsahuje i výpočty oněch dodatečných informací o jednotlivých tvářích, což výpočetní dobu ještě dále prodlužuje. Vezměme si jako příklad testovací data označené v kapitole 6.1. jako Soubor 1. Pokud provedeme testování nad celým souborem obsahujícím 124 obrázků, pak doba výpočtu pomocí frameworku faint a ideálního nastavení zabere jednu minutu třicet šest sekund. Pokud nad tou samou množinou provedeme testování pomocí služby Betaface, zabere nám to přibližně dvacet minut. Toto více jak třináctinásobné zpomalení jistě musíme brát do úvahy.

Další negativní vlastností tohoto způsobu detekce tváří je závislost na jiném zdroji. Pokud by byla služba zprostředkovávající detekci nějakou dobu mimo provoz, nebo pokud by byla dokonce tato služba zrušena, potom by samozřejmě přestala fungovat také detekce mé vyvíjené aplikace pro systém G.A.T.E. V průběhu vzniku této diplomové práce se také stalo, že firma Betaface přesunula jejich webovou službu na jinou adresu, což by znamenalo nutnost zásahu do zdrojového kódu mé aplikace, kde by bylo třeba nastavit novou cestu k webové službě.

Díky těmto zmíněným negativním vlastnostem jsem se rozhodl pro svou aplikaci zvolit detekci na základě frameworku faint. Ten sice nedosahuje takových výsledků, jako detekce Betaface, ale má nespornou výhodu, pokud jde o složitost a s ní spojený čas výpočtu a také ten fakt, že bude detekce fungovat vždy, protože nebude závislá na dostupnosti jiného zdroje. Je však dobré vědět, že existuje i jiný způsob, jak detekci provádět a jak získat pro jednotlivé detekované tváře další dodatečné informace.

Jako rozšíření mé vyvíjené aplikace je tedy například možné například do budoucna uvažovat o implementaci detekce na základě služby firmy Betaface, ale spíše tím způsobem, že by jako primární metodou detekce nadále zůstával framework faint a pomocí služby Betaface bych využil jako dodatečnou metodu pro zisk zmíněných dodatečných informací o jednotlivých tvářích a také pro

minimalizování chybných detekcí (pokud by framework faint chybně detekoval nějakou oblast v obrázku a Betaface by následně tuto oblast jako tvář nerozpoznal, nebyla by ve výsledku za tvář prohlášena ani mou aplikací). Tím by bylo zaručeno, že detekce tváří by byla funkční nezávisle na dostupnosti služby Betaface a pokud by tato služba dostupná nebyla, přišli bychom pouze o dodatečné informace o detekovaných tvářích, nikoliv o celý detekční systém. Jak jsem však již zmínil v úvodu tohoto odstavce, jedná se pouze o úvahu nad možným rozšířením, jako součást mé diplomové práce a s ní spojené aplikace toto rozšíření implementováno není.

6.2. Testování detekce ostatních objektů

Kromě detekce lidských tváří jsem pro účely mé diplomové práce dále zvolil implementaci algoritmu pro rozpoznávání objektů na obrázku pomocí histogramu tohoto obrázku. V následující části tedy popíšu, jak algoritmus funguje a následně ho otestuji při použití dostupných klasifikátorů pro detekci lidských postav, automobilů a motocyklů. Tyto tři klasifikátory také využiji v implementační části jako součást aplikace pro systém G.A.T.E.

Všechny potřebné knihovny, klasifikátory a zdrojové soubory jsou dostupné ke stažení ze stránek Ivana Lapteva [13], který algoritmus pro detekci výše zmíněných objektů navrhl. Samotný proces detekce se spouští jednoduše pomocí souboru objectdet.exe z kořenového adresáře. Stejně jako v případě použití klienta firmy Betaface, tak i v tomto případě o způsobech detekce rozhodují parametry, které tomuto spustitelnému souboru předáme. Adresář s touto detekční metodou také obsahuje krátký textový soubor, ve kterém je vysvětleno, jaké přepínače a jaké parametry lze pro detekci využít.

Následující seznam shrnuje ty nejdůležitější přepínače, které jsem využil jak pro testování, tak pro následnou implementaci:

Přepínač	Funkce
-i	Nastavení zdrojového souboru
-o	Nastavené výstupního adresáře pro xml soubor s výsledky detekce
-c	Nastavení klasifikátoru
-e	Nastavení nejvyššího rozlišení vstupního obrázku. Pokud je tato hodnota (expval) nastavena, je před samotnou detekcí změněna velikost vstupního obrázku na základě faktoru $(1/2)\hat{}expval$
-n	Nastavení počtu úrovní detekce. Při hodnotě 10 dostaneme přesnější výsledky v delším výpočetním čase, při hodnotě 5 kompromis mezi přesností a rychlostí, při hodnotě 2 nejmenší přesnost v nejkratším čase
-t	Nastavení hranice, při které ještě objekt považujeme za rozpoznaný. Výchozí hodnota je 5
-q	Zapnutí tichého módu bez textového a grafického výstupu výsledků

Pomocí přepínače –e lze výrazně zkrátit dobu výpočtu, pokud máme obrázky ve velkém rozlišení a předpokládáme, že objekty, které chceme detekovat, jsou na takovém obrázku velké. Potom díky tomuto přepínači jsme schopni obrázek zmenšit a provádět detekci na takto zmenšeném obrázku bez ztráty informace ve výrazně kratším čase. Nutnou podmínkou je však to, že objekty na obrázku musí být skutečně velké, ideálně zabírající převážnou většinu obrázku. Pokud bychom zmenšili obrázek ve velkém rozlišení, na kterém je detekovaný objekt malý, pak je velmi pravděpodobné, že ho společně s obrázkem zmenšíme tak, že ztratíme schopnost jeho detekce.

Kromě těchto přepínačů lze využít ještě další, kterými můžeme říct, že objekt na obrázku je nějakým způsobem pootočený nebo jinak graficky deformovaný, ale tyto parametry nejsou pro mé účely vhodné a použitelné.

Následující příklad tedy ukazuje spuštění detekce automobilů při deseti stupních detekce, hranice detekce 5, přičemž výsledek je uložen do souboru result.xml: objectdet.exe -i "images/000537.png" -o "result.xml" –c "classifiers/cabmodel_VOC06carside.xml" -n 10 -t 5.

6.2.1. Testování detekce postav

Pro testování detekce lidských postav jsem jako základní soubor dat pro získání prvních orientačních výsledků zvolil 128 obrázků, na kterých se nacházelo celkem 191 lidských postav. Stejně jako v případě tváří, tak i zde jsem musel zvolit jistá omezení pro to, co budu za lidskou postavu považovat. Bohužel, jak jsem uvedl v teoretické části, detekce lidských postav silně závisí na tom, jak je na obrázku zachycena, a proto jsem tyto obrázky musel volit také podle toho, co je vůbec v možnostech použitého detekčního algoritmu. Ve výsledku jsem tedy měl na fotografie zobrazující lidské postavy následující dvě nejdůležitější podmínky. Zaprvé, aby postavy na obrázcích byly ve vzpřímené poloze a zadruhé, aby se na obrázku nacházely z opravdu velké většiny celé (aby se tedy na fotografii nenacházely jenom do půl těla apod.). Jak jsem z testů zjistil, použitá metoda detekce má velké problémy s tím, pokud opravdu není na obrázku zachycena skutečně celá postava a to až do té míry, že pokud na obrázku celá postava je, tak je vysoká pravděpodobnost, že bude detekována a naopak, pokud na obrázku je postava jen trochu „oříznuta“, pak je vysoká pravděpodobnost, že detekována nebude.

Po několika testováních jsem také zjistil, že není nutné použití přepínače –e z velké části především proto, že jsem nepracoval s testovacími obrázky, které by byly v extrémně velkých rozlišeních, a proto nebylo potřeba tyto obrázky před samotnou fázi detekce nijak zmenšovat. Proto jsem provedl testování pro tři hodnoty úrovní detekce (konkrétně pro hodnoty 10, 8 a 5) a pro každou tuto hodnotu jsem následně volil z pěti hodnot hranice detekce (hodnoty od 1 do 5). Pro hodnoty úrovně detekce nižší než 5 byly výsledné testy natolik špatné, že není nutné s nimi nadále pracovat. Stejně tak jsem se rozhodl neuvádět výsledky pro hodnoty úrovně detekce 6, 7 a 9, protože se podobají hodnotám 10, 8 a 5. Následující tabulka shrnuje hodnoty všech použitelných testů, přičemž jednotlivé popisky sloupců odpovídají významem popiskům z kapitoly 6.1.

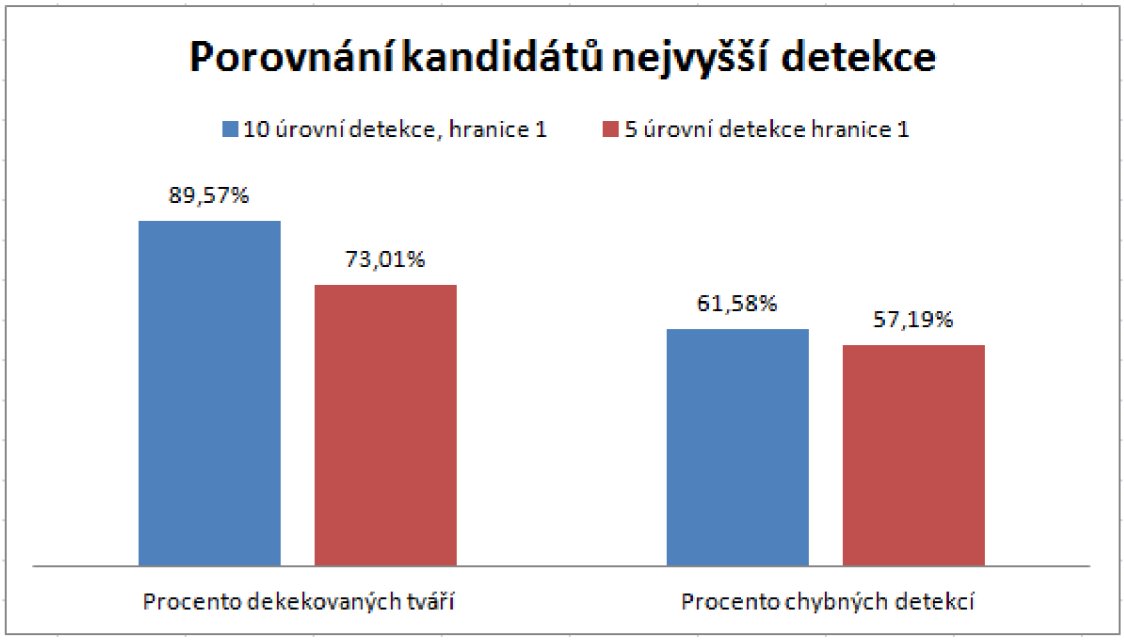
-n	-t	CA	FA	Čas v s	Diff
10	5	36,20%	9,23%	3,50	73,03%
10	4	40,50%	24,14%	3,43	83,64%
10	3	54,60%	27,64%	3,46	73,04%
10	2	69,94%	38,71%	3,45	68,77%
10	1	89,57%	61,58%	3,43	72,01%
8	5	30,67%	9,09%	2,82	78,42%
8	4	36,20%	15,71%	2,83	79,51%
8	3	47,85%	17,02%	2,86	69,17%
8	2	59,51%	38,22%	2,86	78,71%
8	1	82,21%	63,98%	2,85	81,77%
5	5	17,18%	3,45%	1,88	86,27%
5	4	20,25%	8,33%	1,89	88,08%
5	3	33,13%	12,90%	1,91	79,77%
5	2	44,17%	29,41%	1,92	85,24%
5	1	73,01%	57,19%	1,89	84,18%

Jako první je z testů na první pohled vidět, že skutečně s nižší hodnotou stupně detekce klesá čas detekce. Zároveň je také vidět, že tento čas nezávisí na hodnotě hranice rozpoznání, protože je stejně zapotřebí prohledat celý obrázek a tato hodnota pouze určuje, které oblasti budou považovány za rozpoznané.

Stejně tak, jako v případě testů rozpoznávání tváří, tak i pro rozpoznávání lidských postav (a následně také pro rozpoznávání automobilů a motocyklů) naleznu v následující části tři nastavení, která budou použita pro profilování výsledné aplikace.

Dvě z nastavení dosáhla více než osmdesátiprocentní pravděpodobnosti detekce lidské postavy. Z těchto dvou jsem jako kandidáta pro způsob detekce, kdy chceme odhalit co nejvíce lidských postav za ceny vysokého rizika chybného detekování oblastí, které lidské postavy neobsahují, vybral nastavení s hodnotou úrovní detekce 10 a hranicí rozpoznání 1. Do úvahy jsem však bral však ještě jako druhého kandidáta nastavení s hodnotou úrovní detekce 5 a hranicí rozpoznání 1. Toto nastavení sice dosáhlo nižší pravděpodobnosti detekce lidské postavy, ale na druhou stranu také

dosáhlo nižší pravděpodobnosti výskytu chyb. Následující graf (Obrázek 11) graficky shrnuje obě tato nastavení.

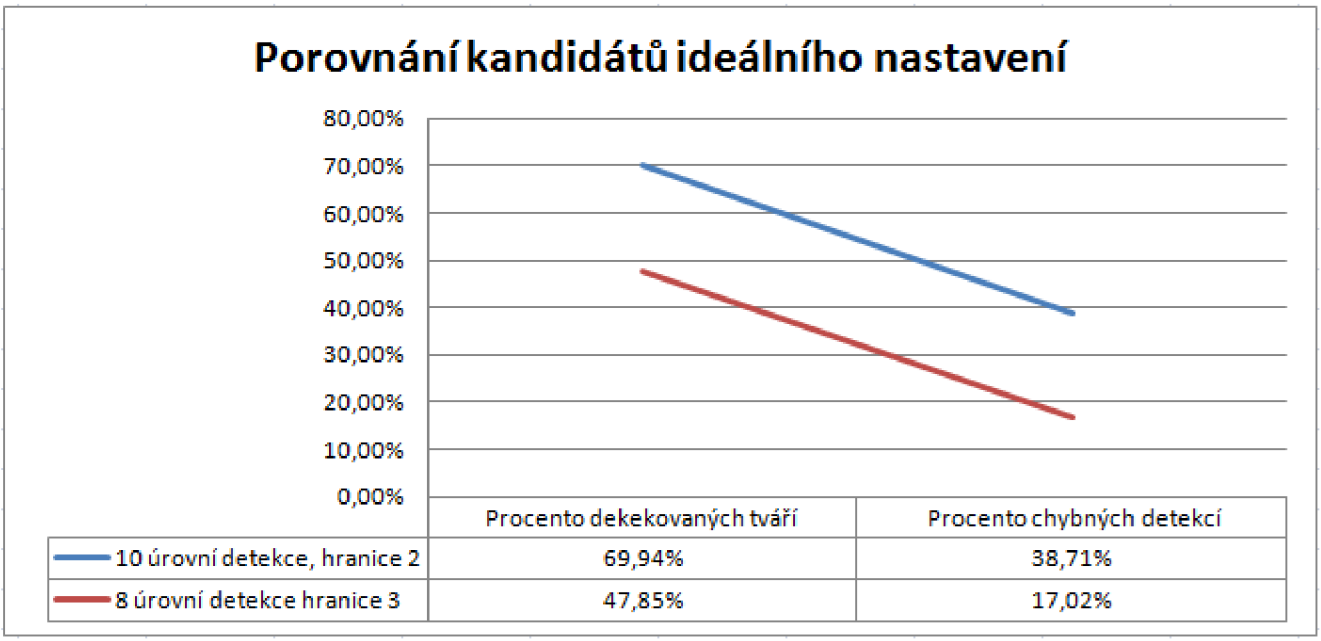


Obrázek 11

Na tomto obrázku je jasně vidět, že rozdíl v procentech správně detekovaných tváří je daleko vyšší, než rozdíl procenta chybných detekcí. Tento fakt byl hlavním důvodem pro zvolení prvního z prezentovaných nastavení pro implementaci do výsledné aplikace. Nebylo by totiž příliš efektivní kvůli zlepšení o pět procent v chybných detekcích ztratit šestnáct procent správně detekovaných lidských postav.

Pro profil, který bude minimalizovat počet chybných detekcí, což znamená, že aplikace sice označí menší procento lidských postav, ale pokud nějakou oblast jako lidskou postavu detekuje, budeme si moci být takřka jisti, že se o postavu jedná, jsem zvolil nastavení úrovně detekce 10 s hranicí rozpoznání 5. Toto nastavení sice detekuje podle testů pouze 36,20% postav, ale při 9,23% chybných detekcích.

Co se týče ideálního nastavení, které by dalo rozumně do poměru sílu detekce a míru chyb, zvolil jsem podle nejnižší hodnoty $Diff$ dva kandidáty. Následující graf (Obrázek 12) zachycuje obě tato nastavení společně s hodnotami procenta detekcí a procenta chyb.



Obrázek 12

Jak je z grafu vidět, obě přímky jsou takřka rovnoběžné, což dokazuje skutečnost, že i jejich hodnoty $Diff$ se příliš neliší. Při výběru vhodného nastavení tedy nastává problém. Tímto problémem je, zda by bylo lepší mít spíše schopnost v ideálním případě detekovat více postav za cenu vyššího procenta chyb, nebo se naopak snažit minimalizovat možný počet chyb na úkor detekční síly. Pro vyřešení tohoto problému by bylo v ideálním případě nejlepší provést široký výzkum mezi potenciálními uživateli mé navrhované aplikace a podle výsledků tohoto výzkumu rozhodnout, která z možností je pro tyto uživatele přijatelnější. Já se však tento problém pokusím vyřešit poněkud jiným způsobem, a to na základě principu, který popíši v následující kapitole. Díky tomuto principu jsem tedy zvolil jako ideální nastavení 10 úrovní detekce při hranici rozpoznání 2 vzhledem k tomu, že má vyšší úspěšnost, co se správných detekcí týče a také vzhledem k tomu, že

v následující kapitole popíšu způsob, jak je možné v některých případech snížit procento chybných detekcí.

Pro shrnutí jsem tedy jako ideální nastavení profileru zvolil výše popsané nastavení, jako to, ve kterém nám jde o to detekovat co nejvíce lidských postav na úkor vyšší pravděpodobnosti výskytu chyb jsem zvolil 10 úrovní detekce při hranici rozpoznání 1 a jako nastavení, které naopak co nejvíce minimalizuje počet chyb nastavení úrovně detekce 10 s hranicí rozpoznání 5.

6.2.1.1. Princip snížení počtu chybných detekcí

V této kapitole se pokusím vyřešit problém nastíněný v kapitole předcházející a omezit tak počet chybných detekcí lidských postav. Konkrétně je tento princip založen na tom, že se pokusím dokázat, že jsme u obrázků, které neobsahují žádné lidské postavy, schopni odstranit vysoké procento chybných detekcí na základě zanesení do detekčního procesu jisté informace o kontextu daného obrázku. Toto téma jsem již prezentoval v teoretické části své diplomové práce (konkrétně v kapitole 2.1.), kde jsem se zmiňoval o tom, že vnášení kontextuální informace do obrázků a využívání této informace pro následné detekce je jedna z možných cest, kterou by bylo vhodné se v budoucnosti vydat, pro ještě vyšší efektivitu a spolehlivost detekčních metod a algoritmů s nimi spojených.

Můj navržený princip je založen na dvou základních věcech. Zaprvé na předpokladu, že použitá metoda detekce lidských tváří je spolehlivější, než metoda detekce postav. Tento předpoklad považuji za správný díky tomu, co teoreticky víme o současných metodách detekce tváří a postav (že oblast detekce tváří je v současné době velmi pokročilá, zatímco v detekci postav žádná z metod nedosahuje zdaleka tak dobrých výsledků). Zároveň tento předpoklad potvrdily i mnou provedené testy. Zatímco nejlepší dosažená hodnota $Diff$ u detekce tváří byla 43,04%, u detekce postav byla nejlepší dosažená hodnota 68,77%, tedy o více jak dvacet procent horší. S tímto předpokladem také souvisí fakt, že detekce lidských postav dokáže jen ve velice málo procentech případů detekovat postavu, která je na obrázku otočená zády.

Zadruhé na tom, že do detekce zaneseme výše zmíněnou informaci o kontextu obrázku. Pokud budeme tedy předpokládat, že metoda detekce tváře je spolehlivější, než metoda detekce postav, pak pokud je na obrázku detekována tvář, se na tomto obrázku nemusí nacházet celá lidská postava (viz fotografie různých typů osobních legitimací apod.). Pokud však v obrázku tvář

detekována není, pak je velice nepravděpodobné, ne-li dokonce vyloučené, že se v tomto obrázku vyskytuje lidská postava.

Díky této kontextuální informaci můžeme tedy v případě, že nedetekujeme v obrázku žádnou tvář, rovnou přeskočit i detekci lidských postav. Samozřejmě, že v případě, kdy detekce obličejů selhala a nějaký obličej se na obrázku nachází, by mohla detekce postav ještě nějakou postavu objevit a ztratíme tak schopnost tuto postavu detekovat. Na druhou stranu s ohledem na budoucí uživatele musíme zvážit, zda by toto nebylo spíše na škodu. Pokud by totiž dostali informaci o tom, že se na obrázku nevyskytuje žádná tvář, ale vyskytuje se na něm lidská postava, pak by tuto detekovanou postavu podle mého názoru ve většině případů podle svých empirických zkušeností považovali spíše za chybu, než za správnou detekci.

Na druhou stranu si také myslím, že zavedením tohoto principu do mé aplikace odstraníme daleko více chybných detekcí, než těch správných. Řekněme, že v ideálním případě (myšleno v ideálním nastavení profileru) máme i pro obrázky, na kterých se nevyskytuje žádná lidská postava, pravděpodobnost 38,71%, že detekční metoda nějakou oblast za postavu prohlásí. Na tomto obrázku se samozřejmě nevyskytuje ani žádná tvář. Detekční metoda pro tváře (opět v ideálním případě) má však pravděpodobnost již pouze 15,25%, že nějakou oblast za tvář prohlásí. Díky tomu se tedy dostáváme pro detekci postav na hodnoty 69,94% správných detekcí pří 15,25% chybných detekcích, což jsou takřka shodná čísla, jako pro detekci tváří. Tento princip samozřejmě má tu nevýhodu, že funguje pouze u obrázků, které žádné tváře ani postavy neobsahují. Pro obrázky, které tváře a postavy obsahují, se dostáváme zpět na 38,71% pravděpodobnosti, že detekční metoda pro postavy prohlásí nějakou oblast v tomto obrázku za postavu, i když žádnou postavu obsahovat nebude. Také tím zároveň neumožníme detekci takových postav, které jsou na obrázku otočeny zády a pro které stále existuje jisté procento, že budou detekční metodou nalezeny. Přesto si však myslím, že

procento těchto případů je výrazně nižší, než téměř 24%, o které jsme snížili pravděpodobnost toho, že v obrázcích bez tváří a postav bude nějaká oblast chybně za postavu prohlášena.

Proto pokud po detekci tváří v mé navrhované aplikaci dostaneme nulový počet výsledků, přeskočím rovnou detekci lidských postav a aplikace se dále bude věnovat dalším detekcím. Tím se mimo jiné také zkrátí čas potřebný k detekci v konkrétním obrázku. V případě ideálního nastavení profileru se tento čas zkrátí v průměru přibližně o 3,5 vteřiny (takřka šestinásobek času stráveného detekcí tváří), což není zanedbatelné číslo v případě, kdy je v podstatě tento čas strávený detekcí, která bude mít v tom lepším případě stejný výsledek, jako kdyby byla rovnou přeskočena. V horším případě dokonce vyprodukuje s 38,71% pravděpodobností chybně detekovanou oblast.

6.2.2. Testování detekce automobilů a motocyklů

V poslední části předtím, než popíši principy fungování výsledné detekční aplikace, se v krátkosti zaměřím na detekci automobilů a motocyklů. Tyto dvě další detekční metody budou také ve výsledné aplikaci zahrnuty. V podstatě nejde o novou metodu detekce, pouze o záměnu klasifikátoru v detekčním algoritmu pro lidské postavy. Samotný detekční proces následně probíhá úplně stejně, jako v případě lidských postav, jen je specializovaný na detekci automobilů (v případě klasifikátoru cabmodel_VOC06carside.xml) a motocyklů (v případě klasifikátoru cabmodel_VOC05motorbikes.xml). Z toho důvodu, že detekce probíhá v podstatě stejně jako v kapitole 6.2. a 6.2.1., přijde mi zbytečné znovu tento princip popisovat a pouze zde prezentuji výsledky testů. Tyto testy byly v obou případech (jak v případě automobilů, tak v případě motocyklů) prováděny nad sadou zhruba 150 testovacích obrázků pro oba dva případy detekce. Je nutné zmínit, že oba dva klasifikátory jsou specializované výhradně na rozpoznávání automobilů a motocyklů na obrázku zachycených z boku. Jsou tedy schopni ještě s jistou pravděpodobnosti detekovat automobil nebo motocykl, který není striktně focen ze strany, ale pod takovým úhlem, kde je ještě bok detekovaného předmětu jasně viditelný. V žádném případě však nelze touto metodou detekovat automobily a motocykly fotografované zepředu a jakákoliv oblast, která pozitivně detekuje přední stranu zkoumaného objektu, musí být na základě vědomí toho, jak tento princip detekce funguje, označena jako chyba. Na základě toho jsem tedy volil takové testovací obrázky, které skutečně zachycovaly zkoumané objekty z boku, abych se nesnažil požadovat po detekční metodě něco, o čem vím, že toho není schopná.

Pro samotné testování jsem zvolil stejná nastavení, jako v případě detekce lidských postav, tedy tři různá nastavení úrovní detekce a pro každou pět hodnot hranice rozpoznání. Následující tabulka shrnuje výsledky testů pro detekci automobilů:

-n	-t	CA	FA	Čas v s	Diff
10	5	59,57%	54,84%	1,24	95,27%
10	4	68,09%	58,44%	1,26	90,35%
10	3	72,34%	65,66%	1,23	93,32%
10	2	85,10%	72,03%	1,25	86,93%
10	1	87,23%	83,06%	1,26	95,83%
8	5	51,02%	51,02%	0,98	99,96%
8	4	55,32%	50,00%	0,97	94,68%
8	3	65,96%	61,25%	0,98	99,29%
8	2	74,45%	70,83%	0,99	96,38%
8	1	85,11%	81,82%	0,96	96,71%
5	5	36,17%	46,88%	0,74	110,71%
5	4	48,97%	52,08%	0,76	103,11%
5	3	57,45%	55,00%	0,76	97,55%
5	2	76,60%	60,87%	0,75	84,27%
5	1	84,49%	76,11%	0,77	91,62%

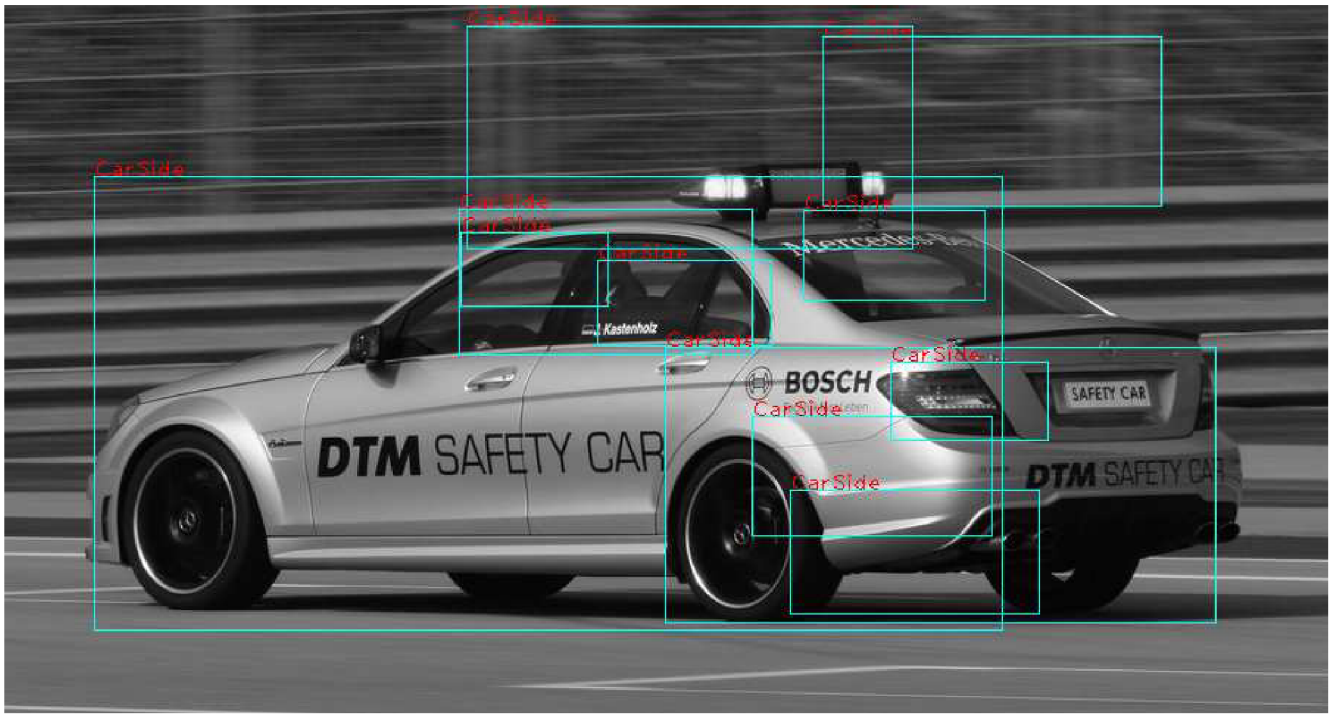
A stejným způsobem také vznikla následující tabulka shrnující výsledky v oblasti detekci motocyklů:

-n	-t	CA	FA	Čas v s	Diff
10	5	30,12%	11,76%	2,52	81,64%
10	4	32,21%	36,00%	2,50	103,79%
10	3	48,65%	44,19%	2,51	95,54%
10	2	62,13%	58,11%	2,53	95,98%
10	1	82,64%	73,89%	2,50	91,25%
8	5	20,33%	16,67%	2,04	96,34%
8	4	26,63%	18,75%	2,01	92,12%
8	3	36,02%	40,00%	2,05	103,98%
8	2	54,18%	55,74%	2,01	101,56%
8	1	78,34%	71,74%	2,00	93,40%
5	5	12,85%	14,29%	1,39	101,44%
5	4	22,64%	15,38%	1,42	92,74%
5	3	32,24%	33,34%	1,41	101,11%
5	2	42,75%	43,24%	1,39	100,49%
5	1	70,01%	69,30%	1,40	99,29%

Jak je z výsledků testů zřejmé, rozhodně nedosahují takových výsledků, jako detekce lidských postav a už vůbec ne, jako detekce lidských tváří. Přesto podle mého názoru i v této oblasti existuje

způsob, jak redukovat procento chybných detekcí při zachování stejného, nebo nanejvýše velice podobného procenta detekcí správných.

V průběhu testování jsem si všiml jistých společných negativních vlastností detekčního algoritmu pro velkou většinu obrázků. Přitom tyto negativní vlastnosti se podle mého názoru dají velice lehce odstranit a díky tomu zredukovat výrazně procento chybných detekcí. Konkrétně, pokud byl v obrázku správně detekovaný nějaký objekt, pak u většiny takto detekovaných objektů nebyla jako výsledek označena jen celá oblast obsahující tento objekt, ale také několik malých oblastí, které se nacházely buďto uvnitř zmíněné celé oblasti, nebo ji nějakým způsobem protínaly. Obrázek 13 zachycuje jeden z takovýchto případů.



Obrázek 13

Tento obrázek vznikl jako výsledek detekce automobilů s hodnotami úrovně detekce 10 a hranicí rozpoznání 1. Jak je na obrázku vidět, bylo detekováno celkem jedenáct různých oblastí, ve kterých se podle detekčního algoritmu nachází automobil. To je z velké většiny pravda, ale z velké

většiny zachycují jen nějakou část tohoto automobilu. Dále je možné si všimnout, že všechny detekované oblasti se buďto nacházejí uvnitř, nebo nějakým způsobem protínají největší oblast, která má svůj popisek na obrázku úplně vlevo a která jediná pokrývá skutečně tu část obrázku, ve které se z velké většiny nachází detekovaný automobil.

V současné situaci tedy máme na tomto obrázku poměr jedna správně detekovaná oblast ku deseti chybně detekovaným oblastem. Tento poměr se pokusím vylepšit v mé aplikaci pomocí dodatečného testování, zda se jednotlivé oblasti vzájemně nepřekrývají a pokud ano, pak do výsledku zahrnu pouze tu, která je ze všech překrývajících se oblastní největší. Na výše prezentovaném obrázku díky tomuto dodatečnému testování tak dostaneme pouze jednu správně detekovanou oblast, přičemž jsme desetkrát snížili počet chybných detekcí.

Zároveň se také podle mého názoru skutečně nijak zásadně nezmění procento správných detekcí, protože pokud se dvě oblasti překrývají, tak by to mohlo znamenat, že jsme správně detekovali dva za sebou stojící objekty. Schopnost detekovat částečně zakryté objekty však prezentovaná metoda detekce na základě histogramů nemá, což znamená, že pokud detekujeme dvě oblasti, které se překrývají, znamená to v drtivém procentu případů chybu detekce.

Jedinou negativní vlastnost tohoto přístupu vidím ve výběru největší oblasti jako tu, která bude zahrnuta ve výsledku. Může se totiž stát, že tato velká oblast bude chybou a správně detekovaný automobil nebo motocykl se bude vyskytovat v některé z menších detekovaných oblastí. Tím ale neztratíme schopnost tento objekt detekovat, pouze podáme chybnou informaci o jeho velikosti na obrázku. To je podle mého názoru negativum, které si můžeme dovolit do systému zanést, pokud díky tomu snížíme počet chybných detekcí v řádu desítek na jeden obrázek (jak jsem doložil ve výše uvedeném případě).

Díky tomuto principu omezení chybně detekovaných oblastí jsem tedy schopen i na základě nepříznivých testů rozhodnout, jaké nastavení použít. Jako to, které se snaží dát do vyváženého poměru počet správně a chybně detekovaných oblastí, jsem pro automobily zvolil 5 úrovní detekce s hranicí rozpoznání 2 a pro motocykly 8 úrovní detekce s hranicí rozpoznání 1. V případě detekce automobilů je to jednoduše nastavení, které má nejnižší hodnotu $Diff$. V případě motocyklů jsem stejný postup aplikovat nemohl, protože to nastavení, které má nejnižší hodnotu $Diff$,dosáhlo příliš nízkého procenta správných detekcí, a nastavení s druhou nejnižší hodnotou $Diff$ zase dosáhlo nejvyššího procenta správných detekcí (toto nastavení bude použito pro maximální detekci). Proto jsem pro motocykly zvolil až třetí nejlepší nastavení, co se týče výsledků testů.

Jako nastavení, které maximalizuje počet správných detekcí, jsem zvolil jak pro automobily, tak pro motocykly stejné nastavení, a to 10 úrovní detekce s hranicí rozpoznání 1. Tato nastavení dosáhla pro oba soubory nejvyšších hodnot, co se správných detekcí týče.

Poslední nastavení, které se snaží minimalizovat počet chyb, jsou pro automobily nastavení s hodnotou detekce 10 a hranicí rozpoznání 5 a pro motocykly nastavení s hodnotou detekce 8 a hranicí rozpoznání 4.

7. Implementace

V posledních kapitolách popíšu strukturu a funkčnost aplikace, která je výsledkem mé diplomové práce. Tato aplikace se skládá ze čtyř tříd, které využívají výše zmíněné detekční programy a knihovny. Tyto čtyři třídy se specializují každá na jiný typ detekovaných objektů (jedna na tváře, další na lidské postavy, automobily a motocykly). Dále má výsledná aplikace obsahuje EJB rozhraní a jeho implementaci, které spojuje všechny tyto třídy dohromady, poskytuje možnost profilovat detekci a získávat výsledky celé detekce nebo jejích částí pomocí textového výstupu a také výstupu v podobě pole objektů typu Region pro další možnou práci s detekovanými oblastmi. V následující části tedy popíšu jednotlivé třídy, jejich funkcionalitu a nakonec zmíněné rozhraní.

7.1. FaceDetection.java

Tato třída se, stejně jako ostatní třídy, nachází v balíku detectionLogic.src.detectiongate a je ze všech tříd nejjednodušší, protože přímo spolupracuje s třídami aplikace faint. Obsahuje následující atributy (všechny atributy jsou soukromé):

Atribut	Popis
boolean detect*	Rozhoduje, zda se bude provádět detekce tváře
int setProfile*	Uchovává hodnotu právě zvoleného profilu
String classifier	Uchovává název souboru klasifikátoru, který bude použit pro detekci
int windowSize	Uchovává hodnotu právě zvolené velikosti okna
float scaleFactor	Uchovává hodnotu právě zvoleného faktoru zvětšování okna
Region[] regions*	Do toho pole jsou uloženy výsledné detekované oblasti
int results*	Uchovává počet detekovaných oblastí

Atributy označené hvězdičkou se nacházejí také v ostatních mnou vytvořených třídách a pro každou třídu mají naprosto stejný význam, proto je už v seznamech u dalších tříd uvádět nebudu. Dále tato třída obsahuje následující metody (všechny metody jsou veřejné):

Metoda	Popis
void detectFaces(String imagePath)	Nejdůležitější metoda vykonávající detekci tváří. Tato metoda vezme obrázek zadaný parametrem imagePath, získá hodnoty atributů classifier, windowSize a scaleFactor, provede detekci a nastaví hodnoty atributů regions a results
boolean setProfile(int profileNumber)	Metoda pro nastavení profilu použitého pro detekci. Možné hodnoty parametru profileNumber jsou 0 (detekce vypnuta), 1 (detekce pomocí ideálního nastavení), 2 (detekce pomocí nastavení maximalizujícího počet detekovaných oblastí) a 3 (detekce pomocí nastavení minimalizujícího počet chyb). Tato metoda nastavuje hodnoty atributů detect, setProfile, classifier, windowSize a scaleFactor
String getProfile()*	Metoda vracející textovou informaci o tom, který detekční profil je aktuálně aktivní
int getNumberOfResults()*	Metoda vracející počet detekovaných oblastí (resp. hodnotu atributu results)
Region[] getRegions()*	Metoda vracející pole detekovaných oblastí (resp. hodnotu atributu regions)
String toString()*	Metoda, která vypisuje informace o výsledcích detekce

Stejně tak jako u atributů, tak i zde se metody označené hvězdičkou nacházejí i v ostatních detekčních třídách a mají naprosto stejný smysl. I ostatní metody této třídy jsou velmi podobné těm, které se nacházejí v ostatních detekčních třídách, liší se pouze v tom, jaké atributy nastavují a jaké hodnoty získávají (viz níže).

Jak je vidět, tato třída má z velké většiny hodně společného s ostatními třídami a dá se tedy říct, že ostatní třídy z ní vychází a pouze upravují některé její vlastnosti a funkce.

7.2. BodyDetection.java

Tato metoda zprostředkovává detekci lidských postav a je založena na detekčním programu Ivana Lapteva uloženém v adresáři src/java/detectionLogic/src/obejctdet. Kromě výše zmíněných atributů obsahuje pouze jeden dodatečný atribut typu int s názvem treshold, který uchovává informaci o tom, s jakou hranicí rozpoznání bude detekční algoritmu pracovat. Stejně tak kromě výše zmíněných společných metod obsahuje další následující metody:

Metoda	Popis
void detectBodies(String imagePath)	Nejdůležitější metoda zprostředkovávající detekci lidských postav. Tato metoda vezme obrázek zadaný parametrem imagePath, sestaví dávkový soubor pomocí metody assembleBatchFile a následně tento soubor spustí a pomocí metody parseResult uloží detekované oblasti do atributu regions
boolean setProfile(int profileNumber)	Metoda pro nastavení profilu použitého pro detekci. Možné hodnoty parametru profileNumber jsou 0 (detekce vypnuta), 1 (detekce pomocí ideálního nastavení), 2 (detekce pomocí nastavení maximalizujícího počet detekovaných oblastí) a 3 (detekce pomocí nastavení minimalizujícího počet chyb). Tato metoda nastavuje hodnoty atributů detect a setProfile
boolean assembleBatchFile(String imagePath, int profileNumber)	Metoda sloužící k sestavení dávkového souboru, do kterého jsou uloženy informace o cestě k detekovanému obrázku zadané parametrem imagePath, informace o zvoleném profilu zadaného parametrem profileNumber a informace o hranici rozpoznání zadané atributem treshold. Tento dávkový soubor spouští program Ivana Lapteva, který provede samotnou detekci
ArrayList<Integer> parseResult()	Metoda sloužící k získání dat detekce uložené v xml souboru, který je výstupem programu Ivana Lapteva. Nastavuje hodnotu atributu results a vrací seznam všech hodnot nalezených v tomto xml souboru.

Všechny tyto metody jsou součástí také následujících tříd (liší se pouze názvy a těmi částmi, ve kterých se namísto slova „body“ objevují pro detekci automobilů slova „car“ a pro detekci

motocyklů slova „mbike“), protože detekce ostatních objektů probíhá úplně stejným způsobem, pouze se změní záznam v dávkovém souboru o tom, jaký klasifikátor má být pro detekci použit.

Za zmínku stojí ještě metoda parseResult, protože je v ní asi nejsložitější logika z celé této třídy. Xml soubor, který je výsledkem detekce, je totiž sestaven tak nešťastně, že obsahuje element „data“, ve kterém jsou uložena čísla oddělené tečkou a mezerou, přičemž některá z těchto čísel označují souřadnice jednoho referenčního bodu detekované oblasti, některá označují šířku a výšku této oblasti a některá hodnotu, s jakou jistotou je možné detekovanou oblast považovat za lidskou postavu. Následující příklad demonstruje vzhled jednoho takového xml souboru:

```
<?xml version="1.0"?>
<opencv_storage>
<detrect type_id="opencv-matrix">
  <rows>12</rows>
  <cols>5</cols>
  <dt>f</dt>
  <data>
    595. 285. 155. 77. 12. 492. 179. 104. 52. 2. 140. 120. 616. 308. 32.
</data></detrect>
</opencv_storage>
```

Tento soubor ve skutečnosti říká, že byly detekovány následující oblasti:

Souřadnice x	Souřadnice y	Šířka	Výška	Jistota
595	285	155	77	12
492	179	104	52	2
140	120	616	308	32

V tomto případě se domnívám, že takový xml soubor naprosto ztrácí svůj smysl a bylo by dokonce jednodušší, kdyby tato čísla byla uložena jednoduše pouze v textovém souboru a mohla tak být jednoduše pouze přečtena a zařazena podle toho, jaký smysl mají, než zbytečně zavádět parser xml souboru, kterým tyto hodnoty získáme. Poté stejně musíme získané hodnoty zařadit, jako kdyby se o textový soubor jednalo. Aby tato situace byla ještě horší, pokud není detekována žádná oblast, není element „data“ prázdný, jak by se zdálo logické, ale obsahuje jeden řádek obsahující nulová, záporná a reálná čísla.

7.3. CarDetection.java a MbikeDetection.java

Tyto dvě třídy jsou ve skutečnosti takřka totožné, pouze do dávkového souboru zapisují jiný soubor klasifikátoru pro detekci automobilů a motocyklů. Kromě výše zmíněných metod obsahují dodatečné dvě metody, které se snaží redukovat pravděpodobně chybně detekované oblasti. Těmito metodami jsou:

Metoda	Popis
Region[] sortRegions(Region[] detectedRegions)	Metoda, která má za úkol seřadit detekované oblasti podle velikosti jejich obsahu (výška oblasti násobená šířkou oblasti). Vrací pole obsahující seřazené oblasti (od nejmenšího po největší)
Region[] removeOverlaps(Region[] detectedRegions)	Metoda, která vezme pole seřazených oblastí podle velikosti jejich obsahu zadané parametrem detectedRegions a postupně prochází toto pole a odstraňuje všechny oblasti, které se vzájemně překrývají. Vrací redukovaný seznam detekovaných oblastí

Oblasti jsou těmito metodami redukovány tak, že se vždy vezme konkrétní oblast v tomto poli a testuje se, zda předchozí (menší) oblasti v tomto poli nějak nepřekrývají tuto zvolenou oblast. Pokud ano, pak jsou všechny takové menší oblasti odstraněny. Tato metoda není stoprocentní, protože samotné rozhodování o tom, zda se dva obdélníky překrývají, není vůbec triviální záležitost. Má verze pracuje tak, že se vždy vezme referenční bod jedné oblasti (nacházející se uprostřed této oblasti) a testuje se, zda se tento bod nachází v jiném obdélníku jiné oblasti. Může se tedy stát, že střed nějaké oblasti se nenachází v jiné oblasti, a přesto se tyto oblasti překrývají. Většina takovýchto redundantních oblastí však vznikne tak, že je detekována pouze část objektu a tato vzniklá oblast se většinou nachází celá, nebo skoro celá v detekované oblasti pokrývající celý objekt (viz Obrázek 13) v kapitole 6.2.2. V takovýchto oblastech se překryv odhalí mnou navrženou metodou vždy.

Jak je vidět, všechny výše zmíněné třídy toho mají vskutku hodně společného, a proto se naskýtá otázka, zda by nebylo lepší tyto třídy spojit do jedné, jejíž logika by rozhodovala o tom, jakou detekci chceme právě provádět. Já jsem použil čtyři třídy namísto jedné zejména z důvodu toho, aby každá z tříd mohla být použita jako nezávislá komponenta, pokud bychom chtěli z mé aplikace použít pouze některý typ detekce.

7.4. EJB rozhraní DetectionGateLocal.java

Toto rozhraní spojuje všechny typy detekce do jednoho celku. Metody, které toto rozhraní poskytuje, jsou následující:

Metoda	Popis
void runDetection(String uri)	Metoda provádějící detekci všemi čtyřmi implementovanými třídami v obrázku zadaném parametrem uri. Parametr uri může být buďto cesta k souboru na disku nebo url obrázku na internetu
boolean setProfiles(int face, int body, int car, int mbike)	Metoda nastavující profily všech čtyř detekčních tříd
boolean setFaceDetectionProfile(int profile) boolean setBodyDetectionProfile(int profile) boolean setCarDetectionProfile(int profile) boolean setMbikeDetectionProfile(int profile)	Metody nastavující profily ne všech čtyř detekčních tříd, ale každé detekční třídě zvlášť
Region[] getDetectedFaces() Region[] getDetectedBodies() Region[] getDetectedCars() Region[] getDetectedMbikes()	Metody vracející pole detekovaných oblastí jednotlivými detekčními třídami
int getNumberOfFaces() int getNumberOfBodies() int getNumberOfCars() int getNumberOfMbikes()	Metody vracející počet detekovaných oblastí jednotlivými detekčními třídami
Region[] getAllRegions()	Metoda vracející pole všech detekovaných oblastí všemi detekčními třídami. Typy těchto oblastí je možné rozlišit pomocí mnou dodaného atributu type ve třídě Region
int getAllResults()	Metoda vracející počet všech detekovaných oblastí všemi detekčními třídami
String wholeDetectionStringOutput()	Metoda vracející textové informace o zvolených detekčních profilech, detekovaných oblastech a počtu těchto oblastí

Toto rozhraní společně s jeho implementací se nachází v balíku detection. V tomto balíku je také dostupná ukázková třída, která předvádí možnou práci s detekčním rozhraním.

V metodě runDetection se také testuje, zda parametr uri je cesta k souboru na disku, nebo zda se jedná o url k souboru na internetu. To se provádí pomocí jednoduchého testu. Nejprve se otestuje, zda existuje na disku soubor zadaný tímto parametrem. Pokud ne, tak je možné, že zadaný parametr bylo url obrázku na internetu. Pokud ano, pak se tento soubor uloží do adresáře mé detekční aplikace a následně se detekce provádí nad tímto obrázkem, protože jak faint, tak algoritmus pro detekci ostatních objektů umí pracovat pouze s obrázky uloženými na disku.

Na závěr bych chtěl zmínit, že všechny výše popsané metody mají přímo ve zdrojových kódech okomentovány popisy, které jsou v takovém formátu, že se z nich dá jednoduše vygenerovat Javadoc, který je ostatně součástí i mé výsledné aplikace (k nalezení v adresáři dist/javadoc).

8. Závěr

Na závěr své diplomové práce bych rád shrnul a zhodnotil dosažené výsledky, uvedl některé pozitivní, ale i negativní vlastnosti mého řešení a také prezentoval některá další rozšíření nebo pozměnění mého výsledného programu, o která by bylo možné můj výsledný program v budoucnu vylepšit.

Zaprvé bych chtěl zmínit, že se v mé diplomové práci více méně potvrdily teorie, které jsem prezentoval v prvních kapitolách své práce. Těmito teoriemi myslím zejména to, že oblast detekce lidských tváří je prozkoumána ze všech ostatních oblastí detekce zaručeně nejvíce. To potvrzuje nejenom to, že na internetu existuje několikanásobně větší množství knihoven a programů, které se zabývají detekcí lidských tváří než všech ostatních objektů, ale především výsledky mých testů. Nejlepší nastavení detekující lidské tváře dosáhlo 72% správně detekovaných tváří při 15% chybných detekcích. Naproti tomu nejlepší nastavení detekce ostatních objektů (konkrétně lidských postav), které jsem ve své diplomové práci využil, dosáhlo sice 70% správně detekovaných objektů, ale bohužel při 39% chybných detekcích. Toto procento není příliš lichotivé, protože to v praxi znamená, že v průměru více jak každá třetí detekovaná oblast je chybou. V případě detekce automobilů a motocyklů je situace ještě daleko horší. S těmito vlastnostmi bohužel nejsme v současné době schopni nic udělat a záleží na budoucnosti oblasti detekce objektů, tedy zejména na tom, zda se objeví nějaký nový způsob detekce, který posune hodnoty výše zmíněných procent do daleko přijatelnějších mezí.

Jako hlavní přínos mé detekční aplikace považuji to, že se snaží skloubit detekci několika různých objektů do jednoho celku. V průběhu sběru podkladů pro mou diplomovou práci jsem sice narazil na několik využitelných knihoven a programů (ať už těch, které jsem uvedl, nebo i některé

další), které řešily problém detekce objektů v obrázcích, ale všechny byly v zásadě úzce specializované na jediný typ detekce. Vskutku tedy považuji moje řešení za jeden z prvních pokusů skloubit několik typů detekce s tím, že dokonce jeden typ detekce závisí na detekci jiného typu, což do celého procesu zanáší jistou (i když ne příliš významnou) informaci o kontextu obrázku. Jako další přínos vidím to, že jsem se snažil nějakým způsobem vylepšit již stávající použité detekční metody, což se mně podle mého názoru podařilo především u detekce automobilů a motocyklů zavedením principu, který několikanásobně redukuje počet chyb odstraněním překrývajících se detekovaných oblastí.

Jako hlavní negativum pak vidím to, že i přes všechny snahy je detekce automobilů a motocyklů nedokonalá (princip odstranění překrývajících se oblastí neřeší případ, kdy se na obrázku nevyskytuje žádný automobil nebo motocykl a přesto je nějaká oblast jako automobil nebo motocykl označena). Proto bych doporučoval považovat informace získané pomocí detekce automobilů a motocyklů spíše jako orientační. Stejně tak, pokud nám nejde konkrétně o detekci těchto typů objektů, nebo pokud nemáme jisté povědomí o tom, že se na obrázků nějaký takový objekt vyskytuje, bych spíše doporučoval detekci automobilů a motocyklů provádět pomocí profilu minimalizující chyby nebo dokonce tuto detekci úplně vypnout.

Dále jsem ve svém řešení příliš nezohledňoval čas výpočtu, který může v některých případech vystoupat až do řádu několika desítek sekund. Pokud však vezmeme na vědomí, v jakém prostředí bude potenciálně má aplikace nasazena (tedy jako podpůrný prostředek pro nevidomé uživatele, který jim umožní získat informace o obrázku při prohlížení internetových stránek), nepovažuji tento atribut za kritický. Pokud například uživatel prohlíží nějaký článek na internetu a nechá si prostřednictvím mé aplikace detekovat obrázek, který je součástí tohoto článku, může v průběhu detekce pokračovat ve čtení zmíněného článku a příliš nezáleží na tom, zda mu výsledky detekce přijdou do několika milisekund nebo sekund.

Pokud jde o vylepšení mé aplikace, pak bych za jednu z možností považoval vylepšení principu odstranění překrývajících se oblastí tak, aby byl schopen odstranit skutečně všechny případy. Otázkou je, zda by to nebylo na úkor ještě vyššího výpočetního času. Dalším možným vylepšením by mohlo být začlenění detekce lidských tváří pomocí služby betaface.com, díky které bychom byli schopni získat další informace o detekovaných lidských tvářích. Na druhou stranu si myslím, že detekce lidských tváří není tou prioritní oblastí, která by potřebovala dále vylepšovat, spíše by bylo zapotřebí nějakým způsobem vylepšit detekci ostatních objektů, která v současné době snižuje efektivitu mé aplikace. Za další (a poslední mnou zmíněné) možné vylepšení považuji uzpůsobení mé aplikace tak, aby mohla fungovat na jiných platformách, než je operační systém Windows. To je způsobeno především závislostí na detekčním programu, který je pro operační systém Windows vytvořen.

I když mé řešení není zdaleka dokonalé a má některé nepříjemné negativní vlastnosti (které z velké většiny nelze pomocí současně dostupných technologií rozumně odstranit), přesto si myslím, že patří ve své oblasti za přinejmenším zajímavé a do jisté míry také unikátní. Snaží se zanášet jistou informaci o kontextu obrázku do detekce a snaží se detekovat oblasti nejen pomocí algoritmu založenému na informacích získaných z numerické reprezentace obrázku, ale také pomocí empirických znalostí, které dále omezují chybné oblasti, pokud je z kontextu obrázku a rozložení oblastí jasné, že některé z nich musí být nutně chybné.

9. Literatura

[1]: *Lsd.fi.muni.cz* [online]. 2007 - 2010 [cit. 2011-09-14]. About GATE project. Dostupné z WWW: http://lsd.fi.muni.cz/gate/.

[2]: YANG, Ming-Hsuan. *Face Detection* [online]. [s.l.], 2008. 6 s. Esej. University of California. Dostupné z WWW: http://faculty.ucmerced.edu/mhyang/papers/face-detection-chapter.pdf.

[3]: ZHANG, Cha; ZHANG, Zhengyou. A Survey of Recent Advances in Face Detection. [online]. Redmond : Microsoft Research, červen 2010 [cit. 2011-09-18]. Dostupné z WWW: http://research.microsoft.com/pubs/132077/facedetsurvey.pdf.

[4]: Eigenface. In *Wikipedia : the free encyclopedia* [online]. St. Petersburg (Florida) : Wikipedia Foundation, 2003-10-24, last modified on 2011-3-07 [cit. 2011-9-19]. Dostupné z WWW: http://en.wikipedia.org/wiki/Eigenface.

[5]: SIROVICH, L.; KIRBY, M. Low-dimensional procedure for the characterization of human faces. *Journal of the Optical Society of America* . 1987, 3, s. 519-524.

[6]: VIOLA, Paul; JONES, Michael. Rapid Object Detection using a Boosted Cascade of Simple Features. In *Accepted Conference on Computer Vision and Pattern Recognition* [online]. [s.l.] : [s.n.], 2001 [cit. 2011-10-02]. Dostupné z WWW: http://research.microsoft.com/en-us/um/people/viola/Pubs/Detect/violaJones_CVPR2001.pdf.

[7]: *Opencv.itseez.com* [online]. 2011, 2011-8-17 [cit. 2011-10-05]. OpenCV API Reference. Dostupné z WWW: http://opencv.itseez.com/modules/core/doc/intro.html.

[8]: *Opencv.itseez.com* [online]. 2011, 2011-8-17 [cit. 2011-10-05]. Cascade Classification. Dostupné z WWW: http://opencv.itseez.com/modules/objdetect/doc/cascade_classification.html.

[9]: Www.betaface.com [online]. 2005 - 2011 [cit. 2011-10-09]. Solution. Dostupné z WWW: http://www.betaface.com/Solution.aspx.

[10]: Introduction. In *Betaface Face Detection and Recognition SDK* [online]. verze 1.73. [s.l.] : [s.n.], 2011 [cit. 2011-10-09]. Dostupné z WWW: http://www.betaface.com/Betaface%20Face%20Detection%20and%20Recognition%20SDK.pdf

[11]: *Www.inria.fr* [online]. 2011 [cit. 2011-10-10]. History of Inria. Dostupné z WWW: http://www.inria.fr/en/institute/inria-in-brief/history-of-inria.

[12]: LAPTEV, Ivan. Improving object detection with boosted histograms. In [online]. Rennes : INRIA Rennes - Bretagne Atlantique, 2008-08-06 [cit. 2011-10-15]. Dostupné z WWW: http://www.irisa.fr/vista/Equipe/People/Laptev/objectdetection/ivc07draft.pdf.

[13]: LAPTEV, Ivan. *www.irisa.fr* [online]. 2007 [cit. 2011-10-18]. Object detection. Dostupné z WWW: http://www.irisa.fr/vista/Equipe/People/Laptev/download.html#objectdetection.

[14]: Optical character recognition. In *Wikipedia : the free encyclopedia* [online]. St. Petersburg (Florida) : Wikipedia Foundation, 2002-04-14, last modified on 2011-12-10 [cit. 2011-12-14]. Dostupné z WWW: http://en.wikipedia.org/wiki/Optical_character_recognition.

[15]: *Jocr.sourceforge.net* [online]. 2010, 2010-09-24 [cit. 2011-10-19]. GOCR. Dostupné z WWW: http://jocr.sourceforge.net/index.html.

[16]: *Netpbm.sourceforge.net* [online]. 2009, 2009-02-22 [cit. 2011-10-18]. The Netpbm Programs. Dostupné z WWW: http://netpbm.sourceforge.net/doc/index.html#programs.

[17]: *Www.oracle.com* [online]. 2011 [cit. 2011-10-21]. Java SE Desktop Technologies. Dostupné z WWW: http://www.oracle.com/technetwork/java/javase/tech/index-jsp-138795.html.

[18]: *Download.oracle.com* [online]. Verze 1.01. 2011 [cit. 2011-10-21]. JavaBeans(TM) Specification 1.01 Final Release. Dostupné z WWW: http://download.oracle.com/otndocs/jcp/7224-javabeans-1.01-fr-spec-oth-JSpec/.

[19]: *Faint.sourceforge.net* [online]. 2007 [cit. 2011-10-22]. Faint - The Face Annotation Interface. Dostupné z WWW: http://faint.sourceforge.net/.

[20]: *Sourceforge.net* [online]. 2009 [cit. 2011-10-22]. Faint - The Face Annotation Interface. Dostupné z WWW: http://sourceforge.net/projects/faint/.

[21]: *Www.fbdownloader.com* [online]. 2011 [cit. 2011-11-02]. FbDownloader. Dostupné z WWW: http://www.fbdownloader.com/.

[22]: *Www.facebook2zip.com* [online]. 2011 [cit. 2011-11-02]. Facebook2zip. Dostupné z WWW: http://www.facebook2zip.com/.

[23]: *Www.betaface.com* [online]. 2005 - 2011 [cit. 2011-11-10]. Betaface - Online Demo. Dostupné z WWW: http://www.betaface.com/Demo.aspx.

[24]: Adaptation (computer science). In *Wikipedia : the free encyclopedia* [online]. St. Petersburg (Florida) : Wikipedia Foundation, 2009-02-25, last modified on 2011-07-02 [cit. 2011-12-14]. Dostupné z WWW: http://en.wikipedia.org/wiki/Adaptation_(computer_science).

[25]: Strojové učení. In *Wikipedia : the free encyclopedia*[online]. St. Petersburg (Florida) : Wikipedia Foundation, 2006-06-10, last modified on 2011-10-30 [cit. 2011-12-14]. Dostupné z WWW: http://cs.wikipedia.org/wiki/Strojov%C3%A9_u%C4%8Den%C3%AD.

[26]: Applet. In *Wikipedia : the free encyclopedia* [online]. St. Petersburg (Florida) : Wikipedia Foundation, 2007-08-18, last modified on 2011-05-23 [cit. 2011-12-14]. Dostupné z WWW: http://cs.wikipedia.org/wiki/Applet.